



UiO :

UNIVERSITY OF OSLO

Composing Software Product Lines with Machine Learning Components

SEBASTIAN SCHATUM NOMME & JØRGEN BORGERSEN

MASTER THESIS IN

INFORMATICS: PROGRAMMING AND SYSTEM ARCHITECTURE - SOFTWARE

60 CREDITS

DEPARTMENT OF INFORMATICS

THE FACULTY OF MATHEMATICS AND NATURAL SCIENCES

Spring 2020

Abstract

Background. A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment. The most considerable benefit of using a software product line is the ability of large-scale reuse. Currently, machine learning models lack reproducibility and suffer from inconsistent deployment. There is a disconnect in machine learning engineering and traditional software that can cause issues when including machine learning models in a software product line.

Aim. The study aims to outline an approach to address the problem allowing stakeholders better to weight their options in regards to how successfully include machine learning components in their software product line.

Method. In the thesis, we developed a prototype and conducted interviews to gain insights into the topic.

Results. Findings suggest that automatic product derivation with machine learning components has a few drawbacks. Manual effort is, in most cases, necessary. By having taken into account all the restrictions and constraints of software product line engineering and machine learning engineering, a composition-based approach is a viable option to architect software product lines.

Conclusion. Utilising a composition-based approach with a component-based system will enable to retain the many benefits of a software product line while including machine learning components.

Keywords: *software product lines, machine learning.*

Preface

First and foremost, we would like to thank everyone who has participated and contributed to the work in our thesis. It would have been hard to complete the thesis without your help and support. Thank you, Snapper Net Solutions for providing experience, employees, and hospitality throughout this process.

We would like to give a massive thank you to our supervisor, Antonio Martini. Thank you for your excellent supervision and guidance throughout a long and complicated period of work. Thank you for always helping us on the right track and having faith in the work that we did. Thank you for being critical and for always asking the right questions.

Finally, thank you for all the support from our friends and family. Through all these years of study, you have shown patience and encouragement towards our work. This is highly appreciated, and we could not have done this without you!

Sebastian Schartum Nomme & Jørgen Borgersen

University of Oslo, June 2020

Table of Contents

Table of Contents	ix
List of Tables	x
List of Figures	xiii
List of Equations	xiv
1 Introduction	1
1.1 Experience and background	2
1.2 Personal motivation	2
1.3 Snapper Net Solutions	3
1.4 Target group	3
1.5 Presentation of the thesis layout	4
2 The case	6
2.1 Problem description	6
2.2 Terms and concepts	7
2.3 Research questions	7
3 Background	11
3.1 Software product lines	11
3.1.1 Motivation / Awareness of problem	11
3.1.2 Fundamental approach	13
3.1.3 Product definition strategy	14
3.1.4 Variability management	15
3.1.5 Process	17
3.2 Machine learning	22
3.2.1 The use of machine learning	23
3.2.2 Methods of machine learning	24
3.2.3 Possible approaches	25
3.3 Recommender systems	32
3.3.1 How do recommender systems work?	33
3.3.2 Filtering methods	33

3.3.3	Top-N recommenders	36
3.3.4	Designing and evaluating recommender systems	38
3.3.5	Accuracy measures	38
3.3.6	Evaluating recommender systems	41
3.3.7	Economical considerations	43
3.3.8	Motivation	45
4	Research process	48
4.1	Conducting research	48
4.1.1	Foundation for the right research process	48
4.2	Our process	49
4.2.1	Define problem case	50
4.2.2	Research theory	50
4.2.3	Initial approach	51
4.2.4	Implementation of second protoype	51
4.2.5	Evaluation	52
4.2.6	Reflection and conclusion	52
4.3	Design science research	53
4.4	DSR framework	54
4.5	DSR Guidelines	56
4.5.1	Guideline 1: Design as an artefact	56
4.5.2	Guideline 2: Problem relevance	57
4.5.3	Guideline 3: Design evaluation	57
4.5.4	Guideline 4: Research contributions	60
4.5.5	Guideline 5: Research rigor	60
4.5.6	Guideline 6: Design as a search	61
4.5.7	Guideline 7: Communication of research	62
4.6	Methods for collecting data	62
4.6.1	Case study	63
4.6.2	Survey	67
4.6.3	Methods for data analysis	68

5	Product foundation	73
5.1	Data access and gathering	73
5.1.1	Explicit data	73
5.1.2	Implicit data	74
5.2	Company A data access	76
5.3	Generating mock data	77
5.4	Foundation	78
5.4.1	Architecture	78
5.5	Requirements elicitation in SPL	80
6	Initial approach	84
6.1	First prototype architecture	84
6.1.1	A tweak of K-Nearest Neighbours	86
6.2	Distance metrics	87
6.2.1	Euclidean distance	88
6.2.2	Pearson correlation coefficient	90
6.2.3	Recommending items	93
6.2.4	Evaluating other distance metrics	95
6.3	Prototype with SPL implementation	96
6.3.1	Architecture	97
6.3.2	Generic code	99
6.3.3	Layout of components and views	101
6.4	Process from first prototype to SPL prototype	103
6.5	Estimates of implementing R2	104
6.6	Considerations for the future	105
7	Implementation	108
7.1	Overall architecture	109
7.1.1	Architecture of the recommender system	111
7.1.2	Architectural flow with recommender system	112
7.2	Implementation of a recommender system in a SPL	113
7.2.1	Pre-processing	114
7.2.2	Model training	119
7.2.3	Model serving	126

7.2.4	Testing, experimentation and evaluation	126
8	Results from evaluation	130
8.1	RQ1: How possible is it to create machine learning components that work for multiple products in a software product line?	130
8.2	RQ2: How reusable are machine learning components in a software product line?	132
8.3	RQ3: How feasible is it to create and consume reusable machine learning models in a software product line?	133
8.4	RQ4: How can you support a Software Product Line Evolution containing Machine Learning Components?	135
8.5	RQ5: How does a software product line affect the quality of its recommender systems?	137
9	Lessons learned	142
9.1	Reactive and feature-oriented approach	142
9.1.1	Composition-based approach	143
9.1.2	Reactive approach	144
9.1.3	Feature-orientation	144
9.1.4	Software composition with mapping to SPL	145
9.2	Variability mechanisms	148
9.2.1	Components and services	148
9.2.2	Parameters	151
9.3	Version-control systems in software product lines	152
9.4	Recommender Systems in SPL	156
9.4.1	ML and SPL	157
9.4.2	Phases of recommendation process	157
9.4.3	Composing components in an SPL ML pipeline	160
9.4.4	SPL feature interaction	161
9.5	Summary	163
10	Discussion	167
10.1	Theoretical contributions	167
10.2	Practical contributions	171
10.3	Ethical Considerations	172

10.4 Threats validity	172
10.5 Related work	175
11 Conclusion	177
12 References	180
A Product requirements	193
A.1 Company A - product requirements	193
A.2 Company B - product requirements	197
B Interview templates	201
B.1 Architecture interview template	201
B.2 Business and process interview template	206
C Survey - Google Form	213
D Code extract from the prototype	218
D.1 Engine	218
D.2 CSV Loader	222

List of Tables

1	Probability of each terminal node	27
2	Course completion on employees of Company A	29
3	Explanations of data labels from mock data in csv file	78
4	Three parameters for <i>type</i> variable	86
5	Similarity ranking among users on mock data.	93
6	Collected data from implementation of first prototype	103
7	Collected data from implementation of SPL prototype	104
8	Predicted data from implementation of R_2 recommender system for Company B	105
9	Requirements for CSV loader	116
10	Description of the hyper parameters we use	125
11	Values for the hyper parameters we used	125
12	Main types of validity threats. Table taken from Feldt and Magazinius (2010), p. 376.	173

List of Figures

1	Economics of SPL engineering. Figure taken from van der Linden, Schmid and Rommes (2007), p. 4.	12
2	Relation of Different Types of Variability. Figure taken from van der Linden, Schmid and Rommes (2007), p. 9.	17
3	Data dependencies in ML Systems. Figure taken from Sculley et al. (2015), p. 4.	22
4	Decision tree for Company A employees showing mock data	26
5	KNN where $k = 3$. Figure taken from Srivastava (2018).	30
6	Logistic regression algorithm. Figure taken from Gupta (2018).	31
7	Collaborative filtering	34
8	User-item matrix	35
9	Content-based filtering	36
10	Rating frequency distribution. Figure taken from Liao (2018).	45
11	Our research process throughout the thesis	50
12	Model:knowledge is generated and accumulated though action	53
13	Design Science Framework	55
14	Design Evaluation Methods. Figure taken from Hevner et al. (2004), p.86	59
15	Rigor cycle	61
16	BAPO model. Figure taken from Bosch (2017).	64
17	Color codes mapped to research questions	69
18	Example of how we analyse interviews	70
19	Example of pie chart from survey	71

20	Explicit mock data	74
21	Implicit mock data	75
22	Excel view of the mock data generated in a <i>csv</i> file	77
23	Three basic techniques for realising variability in an architecture. Figure taken from van der Linden, Schmid and Rommes (2007), pp. 41.	79
24	Product requirements for Company A and Company B	81
25	Architecture of first prototype	85
26	Euclidean distance (JavaScript) example code	88
27	User-feedback diagram	89
28	Pearson correlation (JavaScript) code	91
29	Architecture of SPL components	98
30	<i>EuclideanDistance</i> components module constructor	99
31	<i>PearsonCorrelation</i> components module constructor	100
32	Generic recommender (R_C component) modules constructor input	100
33	First part of SPL prototype	101
34	Second part of SPL prototype, recommending courses	102
35	The three axes of change in an ML application: data, model and code, and reasons for them to change	108
36	Product A architecture	110
37	Process architecture	111
38	Sequence diagram of how the recommender system work	112
39	Recommender system pipeline	113

40	ML pipeline for the recommender system	120
41	<code>LogisticRegression</code> class diagram	122
42	Diagram showing the degree people think recommender systems make them do decisions more effectively	137
43	Diagram showing if people lose trust in recommender systems if bad recommendations are responded	138
44	Diagram showing the distribution of different categories of services where people generally like to receive recommendations from	139
45	Function composition diagram	146
46	Merging feature branches	153
47	Merging feature branches with custom modifications	153
48	Revision control. Figure taken from Apel et al. (2013).	154
49	ML consists of code and data. Figure taken from Breuel (2020).	157
50	ML Pipelines connect data and code to produce models and predictions. Figure taken from Breuel (2020).	158

List of equations

1	Bayes Theorem	28
2	Bayes Theorem with n conditions	29
3	Cross entropy loss metric	40
4	Dice coefficient	44
5	Euclidean distance between two points in a two dimensional space	88
6	Euclidean distance between two points in a n dimensional space	89
7	Pearson correlation coefficient formula	90
8	Gradient descent formula	123
9	Sigmoid equation	124
10	Accuracy formula	124

1 Introduction

"A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (Northrop, 2010, p. 521). Software product lines support large-scale reuse, in which, enable order-of-magnitude improvements in time to market, cost, productivity, quality and other business drivers.

There is a fundamental difference between machine learning and traditional software: Machine learning is not just code; it is code plus data. A machine learning model (the artefact deployed to production), is created by applying an algorithm to a mass of training data, which will affect the behaviour of the model in production (Breuel, 2020). The behaviour of the model also depends on the input data received at prediction time, unknown in advance. This is an issue at the root of the problem, causing a disconnect that has to be addressed before trying to deploy an ML model in production successfully. A few issues related to the production and deployment of machine learning models are:

- Lack of reproducibility
- Performance reduction
- Slow, brittle and inconsistent deployment

The lack of reproducibility is — framed in another context — an issue of reusability when creating machine learning components in a software product line. In this thesis, we will discuss how to compose software product lines in with machine learning components. We are very interested in the quality of reusability and explore the topic through the development of Recommender Systems, which is a subclass of machine learning algorithms. Recommender systems filters and recommends content to users based on discovered patterns in prediction of their ratings or preferences they have given in the past.

We want to add context to term *components* in *machine learning components* as components in a component-based system may provide and require multiple services, whereby each service is described by a service specification. A component can provide a specific service must declare to do so by implementing the interface specified by the service specification. An approach of "programming against interfaces" enables low coupling and flexible designs that are malleable

(Eichberg et al., 2010). Utilising a composition-based approach with a component-based system allows us to retain the many benefits of a software product line while including machine learning components.

In this thesis, we will discuss Software Product Lines Engineering, Machine Learning Engineering and the composition of those two fields.

In this chapter, we present the introduction to our master thesis. We start by presenting ourselves, with the experience and background we have as software developers. Then we present our personal motivation for choosing to conduct this thesis. Then we present the company that we are cooperating with. Then we present the target group for the thesis. And finally, we present the entire layout of the thesis.

1.1 Experience and background

We are two students studying Informatics: Programming and System Architecture, with a specialisation in Software at the University of Oslo. Both have a background in software development, where we have conducted the majority of our courses on the topic of web development. We have had other courses covering a broad diversity of areas within the field of informatics as well, including topics such as databases, software architecture, cloud computing, and so forth.

The most relevant experience we have is by working for different companies through internships or part-time job. Learning from experienced developers, we have gained professional experience in how real production-ready software is being developed and deployed for customers.

1.2 Personal motivation

During our studies, we have had multiple courses on software development and engineering but merely heard about what software product lines are. Discovering and learning about this topic has been a great motivation for choosing this area to research on.

We wanted to get an overarching and holistic understanding of how software products are being developed and how this affects and allows for scaling of companies. We also wanted to combine this with machine learning, because we saw this as an interesting topic to dive deeper into. This motivated us because we have not seen any similar work being done on the same topics with the same approach that we have chosen. We also wanted to learn more about different machine

learning algorithms, as learning about machine learning is more accessible now and has a rapidly growing community.

This is first and foremost, a theoretical thesis. However, we wanted to do a practical part where we could conduct research through the development and implementation of some products in the form of prototypes and proof of concepts. Since we have a background in software development, we found the case to be very interesting and exciting to be part of our research process.

1.3 Snapper Net Solutions

Snapper is a small Oslo-based company delivering e-learning solutions to medium- and large-sized companies with expertise on nano learning. Snapper provided us with the initial case, in which we used to derive our topic of research and problem description. The company was founded in the year 2000, but due to massive technical debt accumulated and tough competition, it has been hard to scale the company. After a very successful launch of an e-learning application called Product A for the consumer goods store chain Company A, they wanted to sell the same product to other customers. This was the birth of their new software product line which radically changed their focus and company structure. Snapper, in that case, has been very interesting to work with as could better understand the needs and difficulties of implementing a software product line as we researched in our thesis.

1.4 Target group

When conducting a master thesis for a company, we see them as a major stakeholder, and that they have a high interest in the results that we provide from the research. We rely on resources and competence from them and see them as an important target group for our research.

The target group is mainly companies that range from small to medium size in the number of employees, similar to the characteristics of Snapper. These companies should be working with a client base with diversity among the customers. The companies should either provide a software product line for the customers or have the possibilities and benefits of doing so. The product line contains products with product-specific configurations and commonalities among the products. It should be relevant and interests in using machine learning algorithms in the product line, that are shared among several of the products.

1.5 Presentation of the thesis layout

Chapter 2 is a presentation of the *case* in the thesis. Here we present the problem description that we want to solve, some frequently used terms and concepts we use throughout the thesis, and finally, the research questions are presented.

Chapter 3 is a complete presentation of the *background* literature and material we use as a foundation when solving our thesis.

Chapter 4 is a presentation of the *research process* that we use as our methodology for conducting the thesis. We present a framework that we use in our process, and finally, we present the chosen evaluation methods we will conduct.

Chapter 5 is a presentation of the *product foundation* that we base the development of the prototypes on. Here we present the different data sets that we work with and what limitations they have, we also present different approaches to designing the architecture, and finally how we handle the different requirements from several companies.

Chapter 6 is a presentation of the *initial approach* we had to solve the case. This was our first experience in making a prototype for solving the problem description.

Chapter 7 is a presentation of the final *implementation* (prototype) we made as a solution to the case. When we became more experienced with the concepts, we saw a better way to solve the problem description and made a new prototype.

Chapter 8 is a presentation of the *evaluation results* we received after conducting the evaluation methods based on the prototype we made. We did a case study and a survey, and present the results in this chapter.

Chapter 9 is a presentation of our *lessons learned* after researching the different topics, developing two prototypes and conducting the evaluation methods.

Chapter 10 is a *discussion* based on the findings and the lessons learned. We argue the outcome of the research questions and discuss the validity and limitations of our research.

Chapter 11 is a *conclusion* on the findings from the evaluation methods and the discussion based on the research questions. We summarize our contributions and suggest the possibility of future work.

2 The case

In this chapter, we start by presenting the problem description for the thesis. Then we present some frequently used terms and concepts in the thesis. And finally, we present the research questions we want to research and find answers for.

2.1 Problem description

In recent years, learning environments have shown increasing importance, playing a fundamental role in teaching and training activities in both academic and business settings. A few of the primary motivations for e-learning is the impact of technological advancements, such as intelligent interfaces, contextual modelling applications, and progress in the field of wireless communication — which altogether has provided numerous new and innovative perspectives for technology users.

Snapper would like to deliver their e-learning system, both as a mobile application and as a web application to a variety of clients. Each product is very similar but has a few unique features adapted to each unique client. In recent times, Snapper has ventured into mobile learning development as it introduces flexibility to the learning process since the access, creation and exchange to information occur naturally due to the omnipresence of mobile devices. Users can decide, when, how and where they feel more comfortable to learn. Due to a large number of mobile devices available in the market, the production of content for these devices becomes strongly dependent on issues such as the manufacturer and operating system.

The introduction of component-based development and service-oriented development has attracted the interest of the software community to the benefits and opportunities of code reuse. The success of before-mentioned initiatives has spurred the reuse in several stages of the software development process, including artefacts such as documents, and models, further increasing the perspective of cost reduction and return on investment (ROI).

The evolution of those ideas has led to the concept of the software product line, which represents a paradigm change in regard to traditional software development. Rather than developing software "project-to-project", businesses should now concentrate their efforts on creating and maintaining core assets, which would be the foundation for the construction of specific products for a given domain.

In the sense of this, a software product line could yield significant benefits for Snapper in the perspective of cost reduction and ROI. Snapper has shown interest in adding functionality to recommend courses based on the individual preferences to each user. For a company with 5-8 employees and an extensive portfolio of customers to manage, using machine learning to recommend courses for a particular client's product deemed too expensive. Snapper questioned whether it would be possible to build a recommender system which could serve as a core asset in a software product line for multiple customers.

Motivated by this scenario, in this thesis, we will examine the benefits of systematic reuse of an SPL in the context of a recommender system. The goal is to promote overall quality, domain comprehension, and reduction of time spent in the development and maintenance of building software product lines with machine learning components.

2.2 Terms and concepts

SPL is an acronym for **Software Product Lines**. The concept of a software product line is used to describe an approach where common components and services are used to satisfy specific requirements of a market segment shared by multiple products developed by the same company.

ML is an acronym for **Machine Learning**. Machine learning provides systems or algorithms that learn and improve automatically through experience based on data provided by users.

Recommender systems are a subclass of machine learning algorithms. It filters and recommends content to users based on discovered patterns in their ratings or preferences they have given in the past.

These terms and concepts are further described in chapter 3, and are frequently used throughout the thesis.

2.3 Research questions

The overarching topic of the thesis is whether machine learning and software product lines work together. By this we mean that we want to implement machine learning components into the generalised components of the software product line, so each instance of the product line can use the machine learning components as any other component. By researching through implementation, we want to create new knowledge about machine learning models and software

product line theory to clarify whether the case can be solved or not. We have investigated the following research question with sub-questions in this thesis:

RQ1: How possible is it to create machine learning components that work for multiple products in a software product line?

The first research question (RQ1) is the main question we are researching. We investigate whether it is possible to accommodate machine learning components into a software product line or not. A lot of articles have been published on similar topics with machine learning and software reusability. For example: Di Stefano and Menzies (2002), Morisio et al. (2002) and Camillieri et al. (2016). But they all focus on software reuse and evolution within a specific system or product, which is a slightly different approach than to have a software product line with generalised components. Their approach is to reuse and evolve some parts of the code, rather than entire components which we are researching. We use these articles as support for our research, but will further research the theory of software product lines rather than software reuse.

RQ2: How reusable are machine learning components in a software product line?

This sub-question (RQ2) focuses on reusability, which is a big part of our thesis. The research we conduct is to understand to what degree this is possible. We are interested in knowing how notable a percentage of the machine learning components code that is reusable and can be used to create generalised components. Another aspect that we want to research is what parts of the machine learning components that are reusable.

RQ3: How feasible is it to create and consume reusable machine learning models in a software product line?

We have been researching the development (consummation, with the framework TensorFlow) to build machine learning models to give users recommendations. TensorFlow is an architecture for executing graphs of numerical data. TensorFlow figures out how to distribute processing across the various GPU cores of your computer, or across various machines on a network, and allows for massive computing problems in a distributed manner (Cardoza, 2018). Our goal is to create a prototype with a machine learning component that can be used for multiple products and devices serviced through an interface and evaluate how to manage and evolve a software product line containing these machine learning components. **This sub-question (RQ3) focuses not only**

on the possibilities of reusability but the costs of doing it. We want to see if it is cost-beneficial to reuse machine learning components.

RQ4: How can we support a software product line evolution containing machine learning components?

Modern software systems tend to be long-living and, therefore, have to undergo continuous evolution to cope with new, and initially unforeseen, user requirements and application contexts. In practice, the necessary changes applied to design-, implementation-, and quality-assurance artefacts are often performed in ad hoc — conducted in a manual manner — thus lacking proper documentation, consistency checks among related artefacts, and systematic quality-assurance strategies. These issues become even more challenging in case of variant-rich software systems such as software product lines; even a small change may (erroneously) affect a large number of similar product variants simultaneously. **This sub-question (RQ4) is to research how to develop and evolve machine learning components in an evolving software product line.**

RQ5: How does a software product line affect the quality of its recommender system?

When multiple products share the same machine learning models, the quality of the predictions it provides can be affected. We have been researching whether this is the case, and if this can be an issue of having such components in a software product line. We also want to see how much it affects the quality and if this is a problem for the end-users. **This sub-question (RQ5) is to research if shared machine learning components satisfy the end-users requirements and needs.**

3 Background

In this chapter, we research the different topics relevant to our research, with the purpose of giving background and fundamental understanding of what we research and use as a knowledge base in our thesis.

First, we give a thorough background on software product lines and different topics within this area. Then we present the main concepts of machine learning theory, to then go deeper into different machine learning approaches and algorithms. And finally, we present the concept of recommender systems.

3.1 Software product lines

Software increasingly becomes an important asset for modern, competitive products. Simple or complex, small or large, there is barely any product without software. Software product lines (SPL) have gained attention in recent years due to its quality, cost and time to market concerns. Companies prize software reuse to capture more value from their investments.

3.1.1 Motivation / Awareness of problem

To embark on a software product line approach is down to different reasons – ranging from process oriented-aspects as cost and time to end-user aspects as interface consistency. The move towards software product lines is usually based on economic considerations: the approach supports large scale reuse during development. As opposed to traditional reuse approaches, this can be as much as 90% of the total software. Reuse are more cost-effective than development by orders of magnitude. Cost and time to market are heavily correlated in software product line engineering.

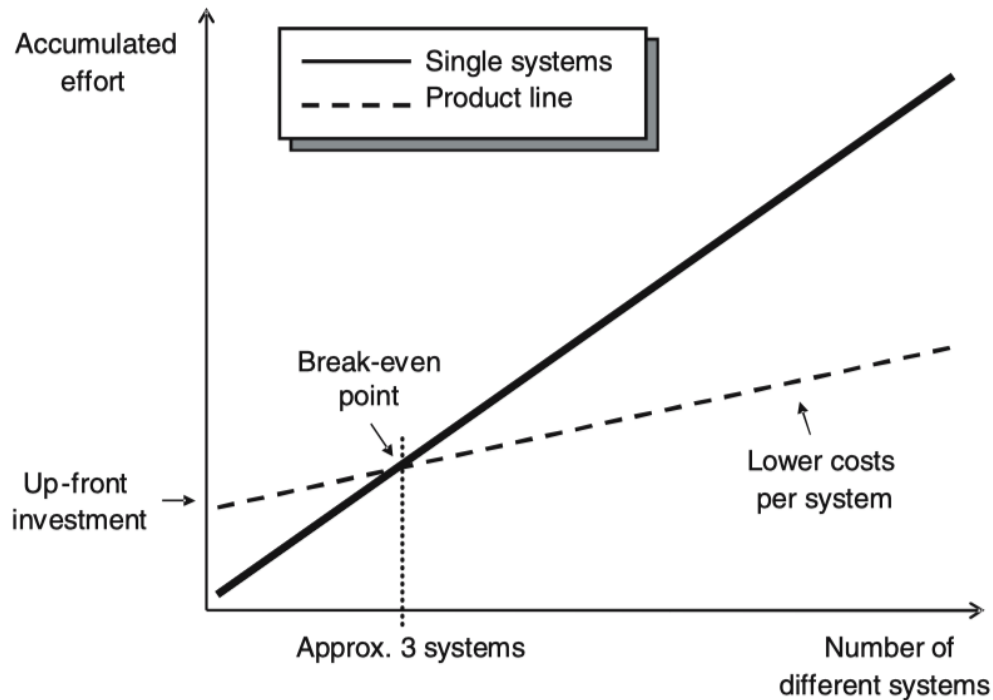


Figure 1: Economics of SPL engineering. Figure taken from van der Linden, Schmid and Rommes (2007), p. 4.

Thus, both development costs and time to market can be dramatically reduced by a software product line approach. Other benefits include the improvement of qualities in the resulting product as the reliability, ease of use, and decrease in product risk (Ferguson, 2018). Unfortunately, these benefits does not come for free but requires some extra initial investment — needed for building reusable assets, transforming the organisation, etc. Various approaches exist to make this investment, such as incremental strategies or the big bang adoption (instant changeover). Regardless, the need for underlying set-up remains. Break-even happens after about three products (as shown in figure 1), along with a reduction in maintenance costs, i.e. the overall amount of code and documentation that needs to be maintained is reduced along with project size and risk.

Software product line engineering has a strong impact on the quality of the resulting software. New applications will then consist of a large extent of matured and proven components, which leads to more reliable and secure systems because the defect density can be presumed to be lower than products that are developed anew. Process qualities such as quality assurance are

supported in software product line engineering by regarding a product and its simulation as two variants. When both variants are derived from the same code; simulations can be used as a foundation for analysing the quality of the end product. Thus, enabling extensive testing that would not be possible otherwise. While arguments of costs typically dominate the product line engineering debate, the ability to produce higher quality is for many organisations (especially in safety-critical domains) the primary reason to expend major efforts into software product line engineering.

Beyond process qualities, software product line engineering impacts product aspects like the usability of the final product by among things improving the consistency of the user interface. This can be achieved by using the same building blocks for implementing the same kind of user interaction — usually as a part of a design system. It is taking advantage of having a single component for user registration or product rating for a whole set of products instead of having a specific one for each product. In some cases, demand for this kind of unification has been the basis for the introduction of a product line approach in the first case.

3.1.2 Fundamental approach

Software product lines require a shift of focus: from the individual system to the product line — implying a change in strategy from the ad-hoc next-contract vision to a strategic view of a field of business.

Software product lines rely on a fundamental distinction of *development for reuse* and *development with reuse*.

Domain engineering (development for reuse) provides a basis for the actual development of individual products. Product line infrastructure encompasses all assets that are relevant through the software development life-cycle instead of a narrow view on code assets common in traditional approaches. Thus, the pooling of all assets is defining for the product line infrastructure. A key distinction of software product line engineering from other reuse approaches that the various assets themselves contain explicit variability. For example, a representation of the requirements may include an explicit description of specific requirements that apply only for a certain subset of products. Individual assets in the product line infrastructure are linked together, just like assets in software development.

Application Engineering (development with reuse) builds the final products on top of the product

line infrastructure. Application engineering is animated by the product line infrastructure, which contains most of the functionality required for a new product. Variability explicitly modelled and added in the product line infrastructure provides a foundation to derive individual products. In other words, when a new product is developed, an accompanying project is set-up. Then requirements are gathered and categorised as a part of the product line (i.e. a commonality or variability) or product-specific. After that, the various assets (e.g. architecture, implementation, etc.) may be instantiated right away, leading to an initial product version. Depending on the product line, the majority of the product should be available from reuse; only a small portion must be developed in further steps.

The developed product platform determines the capability of the company to perform business in the market; consequently, there are considerable ties with how an organisation does business and its overall market.

There exist a few characteristics relevant to the discussion about product lines. We can categorise them into:

- Product definition strategy
- Market strategy
- Product line life-cycle
- The relation of product line strategy and product line engineering

3.1.3 Product definition strategy

Product definition strategy illustrates how new products are defined. There are two main divisions within the product definition strategy: customer-driven and producer-driven. In a customer-driven situation, the specific product is mapped and determined based on demands from existing and future customers. The end product is individualised til each customer's desires — mass customisation — which proposes that there exist many different customer needs and the requirements for each product is hard to define in advance. The product line platform must support flexible extensibility in the further development of products.

On the other hand, in a producer-driven strategy, the producer is responsible for the design and development of the product line that defines the product(s). This approach is common when the

product is developed for mass-markets; when each variant is sold to a large number of different customers.

The producer-driven strategy can be further divided into market-oriented and technology-oriented strategies. In a market-oriented strategy, the products in the product line portfolio are accepted based on an analysis of potential market segments. New products are defined mainly to satisfy new market segments or changes in established segments. As opposed to, a technology-oriented strategy where the growth opportunity is influenced by the technological capabilities and opportunities developed by the company, delivered to the market. Product definition strategy has importance when deciding the product portfolio; offered by the company.

In practice, the product definition strategy is usually a mixture of the examples above. Product line engineering can support all of these approaches, but its relative advantage varies til relation to the strategy used.

Some essential questions to answer are:

- Should a product line be started at all?
- Which product shall we develop as a part of a product line?
- What shall be the characteristics or features of these products?
- Which functionality shall be developed as individual functionality?
- What functionality shall be developed as apart of the product line, based on the platform?
- How shall we evolve the product line over time?

In this thesis, we used the product definition strategy in the development process of the prototypes made, by considering the questions as mentioned above, while developing and planning functionalities for the prototypes.

3.1.4 Variability management

Software product line engineering aims to support a range of products; supporting individual and different customers or address entirely different market segments. Variability is a key concept in this regard. Instead of understanding each system by itself — software product

line engineering looks at the product line as a whole and the variation among the individual systems. This variability must be managed throughout the process. Variability management covers the entire life-cycle and starts with the early steps of scoping, including implementation and testing, eventually going into evolution. As aforementioned, variability is relevant to all assets throughout software development.

Types of variability

1. *Commonality*: a characteristic (functional or non-functional) can be common to all products in the product line. The commonality is implemented as a part of the platform.
2. *Variability*: a characteristic that is common to some of the products, but not all. Variability must be explicitly modelled as a possible variability and implemented in a way that allows having it in selected products.
3. *Product-specific*: a characteristic may be part of only one product — at least for the foreseeable future. These types of specialities are not required by the market per se but are due to concerns of individual customers. While these variabilities will not be included in the platform, the platform has to be able to support them.

A specific variability may change in type during the life-cycle of the product line. Product-specific characteristics may become a variability or even a commonality — should a decision be made about supporting an alternative characteristic. Thus, extending the platform beyond the initial scope of the product line.

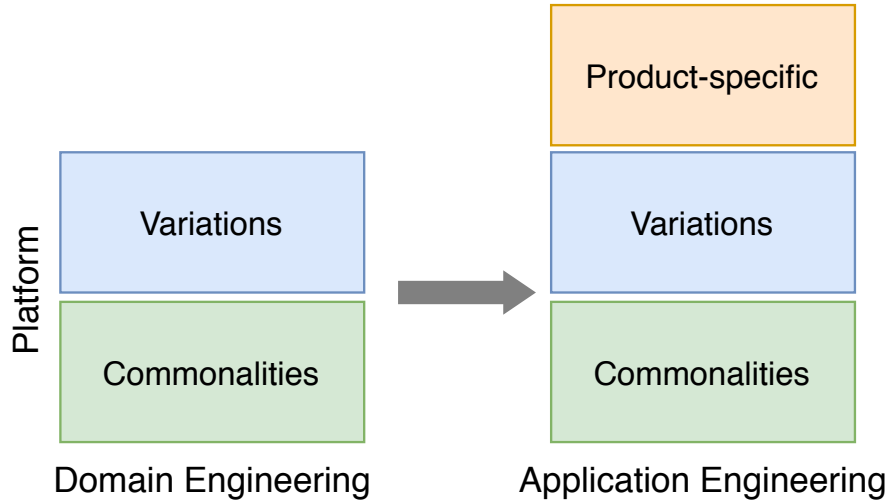


Figure 2: Relation of Different Types of Variability. Figure taken from van der Linden, Schmid and Rommes (2007), p. 9.

Commonalities and variabilities are handled regularly in domain engineering, and product-specific parts are exclusively handled in application engineering.

The different concepts and types of variability management are used in our thesis to map out the functionalities (or features) for the various products that each company requests. We address these requests in our study and implementation of prototype components for the SPL.

3.1.5 Process

The product line infrastructure is not a goal in itself. Its ultimate goal is its utilisation during application engineering — also called the instantiation of the variability.

When new requirements are defined during application engineering, the future of each requirement in the life-cycle must be considered: should it be a part of the platform or as a part of the product development?

In the simplest case — when the product line infrastructure supports the requirement — it is a question of binding of time. A variant can be seen as a binding of time (compile-time, start-up time, etc.).

Though when the product line infrastructure does not support the requirement, there are three options: Either try to renegotiate or cancel the requirement. In the context of a product line

— every supported variability increase the complexity of evolving the product line further. Integrate the new requirement with the product line infrastructure. This can usually be done with a systematic scoping process. Integrate the new requirement on an application-specific basis.

Both the second and third case usually occurs during the same system development. The second case leads to a hand-over to domain engineering and while the third case leads to a *per se* development cycle in the application engineering.

Business-centric

Product line engineering addresses the market as a whole, whereas traditional software tends to focus on the individual system. For product line engineering to remain successful in the long term; the product line infrastructure has to be an adequate tool to field new products onto the market efficiently. A holistic relationship between the development choices of an individual product and the product line has to be managed from an economic standpoint.

Because of the relationship between the individual product and the product line — larger business objectives must be well understood. Previously the goals have been addressed as time-to-market reduction, effort (and cost) reduction, usability and reliability improvement. The intent of usability improvement inherently supports user interface consistency. Thus, these goals provide a basis for a product line engineering effort. Moreover, the choice of whether it is implemented in full or on a *per se* application-specific basis. Either way, taking a business-centric approach to product line engineering means that key choices about the inclusion and realisation are based on a systematic financial decision. Thus, the break-even of three product implementations is a rule of thumb in deciding the costs of adding functionality as a part of domain engineering. A scoping analysis is a common tactic to inform about the different available options:

- Product portfolio planning
- Domain potential analysis
- Asset scoping

Product portfolio planning is used to capture the products that will be a part of the product line and to identify their main requirements — based on the commonalities and variabilities

required. Product portfolio planning is the first step at which optimisation can (and should) occur. Though this activity is business-centric — because of product costs — technical aspects must be taken into account as well.

Domain potential analysis has a strong focus on an area of functionality to determine whether an investment into a software product line should be made. This is usually done with a top-down approach with a holistic view of the product line; some strategies focus on the individual areas of the product line. The overall result of this activity corresponds to an assessment grounding in the question about where reuse investments should be focused.

Asset scoping aim to define the individual components that must be built for reuse. Two viewpoints (business and architectural) must be brought together to identify these components adequately.

During the life-cycle of a product line, a team is usually responsible for managing the initial set-up and evolution of the product line.

Architecture-centric

A common product line architecture (also called reference architecture) is crucial to the success of the product line engineering approach compared to other reuse approaches. The reference architecture is designed in domain engineering to provide a coherent overview of the various components that shall be used. Having a single environment for all components used in the individual products ensures that there is no need to develop multiple components that address similar functionality and differ only concerning their environment. The reference architecture is used in each application engineering cycle to derive a new product instantiation; both, for assignment of work in the development process and for determining the modification of assets to support product-specific requirements. In a few exceptional cases, product lines have been set up without significant investments in software architecture. Though it is safe to say that a robust product line architecture assumes the overall success.

Two-life-cycle approach

Software product line engineering consists of domain engineering and application engineering. In the ideal case — these two types of engineering are only loosely coupled and synchronised by software releases. This is a key characteristic of a product line as it allows for them to be conducted based on different life-cycle models.

Domain engineering focuses on the development of reusable assets that can provide a necessary range of variability. The underlying software development approach depends on being able to handle long-term, very complex system development. Domain engineering activities include:

- Product management
- Domain requirements
- Domain design
- Domain realisation
- Domain testing

Product management aims to identify the commonalities and variances among the products — defining the products that will constitute the product line. Furthermore, it encompasses the product portfolio planning and the economic analysis of the products in the product line. The output of product management is usually a product roadmap.

Domain requirements engineering begins with the product roadmap. It has an end goal of outputting a comprehensive list of requirements for the various products in the product line with an initial variability model.

Domain design is an activity for developing the reference architecture. It provides the basis for all future instantiations of the product line.

Domain realisation encompasses the detailed design and implementation of reusable software components — planned variability, which has been expressed as a requirement, must be realised with adequate implementation mechanisms.

Domain testing is used to validate the generic reusable components that were implemented as a result of the domain realisation. This is especially hard because the implemented variability must be taken into account, and there is no specific product which provides an integration context. Though everything is not wrong, the activity of domain testing offers a lot of groundwork for application testing by generating reusable test assets that can be used in application testing.

As a result, domain engineering provides a common product line infrastructure with all the required variability.

On the other hand, application engineering consists of the following activities:

- Application requirements engineering
- Application design
- Application realisation
- Application testing

As opposed to single system approaches — groundwork has been completed during the domain engineering phase — staying consistent with the reference architecture enables plug-and-play reuse.

Application requirements engineering is used to identify the requirements for an individual product to stay as close as possible to the existing product line infrastructure.

Application design is the activity that derives an instance of the reference architecture and adapts it to the requirements from the requirements specification. During this design phase; the product-specific adaptations are built.

Application realisation is the final implementation of the product that is developed — including configuration and reuse of existing components as well as building new components corresponding to the product-specific functionality.

Application testing is the last step, and it is when the product is validated against the application requirements. There is a lot of readily available reusable assets from the corresponding domain engineering activity.

While the integration of domain engineering and application engineering largely situate on the context: it is essential to separate these activities as they are carried out with different objectives and criteria of quality in mind. This is especially true when both life-cycles are enacted by the same people, which is often the case in small businesses. The aspect of the process and the two-cycle approach is relevant to discuss because it has to be addressed throughout the entire production of the software product line and is a crucial aspect to consider when deciding on what kind of SPL architecture to use.

3.2 Machine learning

Demands for machine learning (ML) capabilities in software products are increasing. Businesses seek to combine advanced analytics in predicting system outcomes without being explicitly programmed — with the prime purpose to allow computers to learn without human interference and adjust accordingly.

Machine learning capabilities can be used to improve decision-making and support critical business strategies. Machine learning components have to be continuously measured and monitored, in order to understand their behaviour. Changes in the product or external user conditions can have cascading consequences on the machine learning models, and the models will become less accurate and robust, that can result in issues that are hard to correct. Gartner predicts that 60% of big data projects *“will fail to go beyond piloting and experimentation, and will be abandoned”* (Goasduff, 2015). Therefore, the entry barrier can be very high from a perspective of product development because of the difficulty in building and maintaining machine learning products.

Thus, machine learning is hard to put into production. Software products are divided into components or layers that communicate with each other. The machine learning component of the product is often integrated through an API. Challenges occur when yet a simple machine learning component bring too many auxiliary components along with it, in order to function. Sculley et al. state that only a small fraction of real-world ML systems are composed of the machine learning code (Sculley et al., 2015), which figure 3 illustrates.

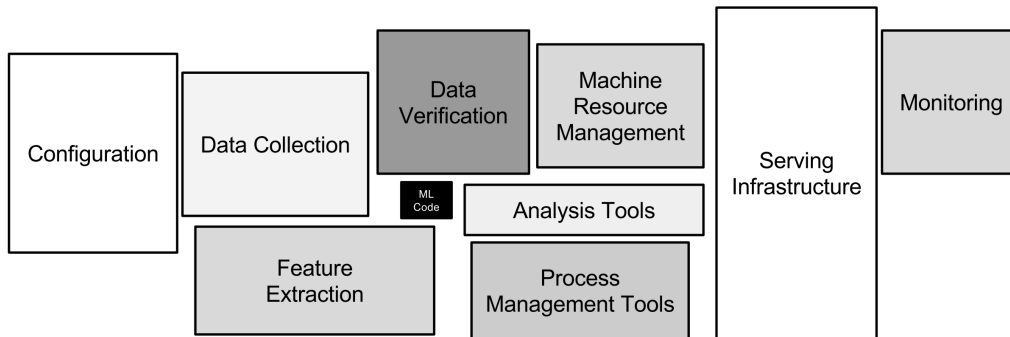


Figure 3: Data dependencies in ML Systems. Figure taken from Sculley et al. (2015), p. 4.

Surrounding but the necessary infrastructure is vast and complex. Machine learning products require spending significant effort to make additional dependencies work. Given the complexity

of machine learning products, the development strategy can be more important than acquiring the right tools.

3.2.1 The use of machine learning

Machine learning is a technique of data analysis that automates analytical model building. It is a branch of artificial intelligence based on the idea that machines should be able to learn and modify through experience. The process of learning begins with observations or data to look for patterns in the data and make better decisions based on the data we provide.

Two operational phases characterise machine learning models; the *training* (or learning) phase and the *testing* (or prediction) phase. In the training phase, a model is trained by explicitly feeding it data that has the correct answer attached (historical data). This training data is used to find patterns in the data and connect them to the right answer. Once trained this way, a model can be supplied with new data (typically unseen at training time) to generate run-time predictions (i.e. to compute the learned map on new data). These two phases are not always disjoint: incremental learning approaches exist that allow adapting the parameters of an ML model continuously and thus, predictions respond to new input data.

Data used in these phases can be divided into three portions: *training data*, *cross-validation* (or dev) *data* and *testing data* (Bajo, 2020). The training data is used to let the model recognise patterns in the data (adjust the parameters of the model), to reduce bias and the predictions (i.e. to fit the data). The cross-validation data is used to ensure better accuracy and efficiency of the algorithm used to train the model. The validation data is not seen by the model during training and has the aim to reduce variance (i.e. eliminate over-fit). Lastly, the test data is used to provide an unbiased evaluation of the final model. Nor this data is seen by the model during training. Furthermore, test and cross-validation data should come from the same distribution, to reduce data mismatch (Assawiel, 2018).

The final quality of the machine learning model predictions is influenced by the quality of the training data and the adequacy of the learning model for the specific computational learning task.

3.2.2 Methods of machine learning

There are various ways an algorithm can model a problem based on its interaction with the experience or the environment. Machine learning has a broad research field encompassing several paradigms, e.g. neural-inspired, probabilistic, kernel-based approaches and addressing an array of computational learning task types (Heller, 2019). For the purpose of this thesis, we will focus on machine learning models and algorithms targeted at solving *supervised* and *unsupervised learning* tasks, and merely touch upon *semi-supervised learning*.

Machine learning algorithms and models require the use of libraries because it relies so heavily on mathematics. These libraries are functions and routines that make it easier to do complex tasks, without having to write multiple lines of code. We use a library called *TensorFlow.js*. How we use it and why we chose this library over other is further explained in chapter 7.

Supervised learning

Supervised learning refers to a specific class of machine learning problems related to the learning of an unknown map between input information and output prediction (Bacciu et al., 2015, p. 75). After adequate training, the system will be able to provide targets for any new input. Also, it can compare its output with the correct intended output and find errors to modify and customise the model accordingly. Common supervised learning techniques include regression and classification. In a regression model, the value of the labels belongs to a continuous set (boundary values). On the other hand, in a classification model, the value of labels belong to a discrete set and can have as many categories as reasonable.

The input data is called or defined as the training data. All data fields are assigned (labelled) with a category. Both the categories and the assigned category to a data field are selected by people, and the data can, therefore, be biased, meaning external factors affect the model. After the data has been labelled manually, the model is prepared through a training process where it predicts categories for the data fields. In this process, the model has to predict labels, and are corrected when the predicted labels are wrong. This training process continues until the model has achieved an expected level of accuracy, where a certain amount of the predictions done by the model are correct (Brownlee, 2020).

Unsupervised learning

Unsupervised learning is used when the information used to learn is neither classified nor labelled. Instead of responding to feedback, unsupervised learning identifies commonalities in the

data and responds based on the presence or absence of such commonalities in each new piece of data (Soni, 2020). In other words, unsupervised learning can be used for discovering the underlying structure of the data. Some applications of unsupervised machine learning techniques include clustering, anomaly detection, association mining and latent variable models.

The input data or any data are labelled and have, therefore, no known results. A model is prepared by deducing structures that occur in the input data, and out of this may some general rules be extracted (Brownlee, 2020). For example, the data may be organized by similarity rules.

Semi-supervised learning

Semi-supervised learning is an approach where the some of the input data is labeled, and some data is not. This is an approach where the model has to learn how to label the unlabeled data based on the labeled data. For some cases it might increase the accuracy, or even save a lot of time and cost for the model to first learn from the labeled, and then to predict the unlabeled data fields (Gupta, 2019). It is a combination of supervised and unsupervised learning that we choose not to focus on, because we find it less relevant to our thesis.

3.2.3 Possible approaches

We have been researching different machine learning algorithms that can solve our problem. In the following section, we will describe the most relevant algorithms we found, and how applicable and feasible they are to solving our problem.

Some of these approaches use different distance metrics to calculate distances between a set of coordinates or data points. How these distance metrics work and how they are implemented are further described in chapter 6.2. We describe the ones we used, and some other metrics we considered using but found less applicable.

Decision trees

Decision trees are a supervised learning algorithm that is most commonly used for classification problems (Ray, 2017). The goal of the algorithm is that the training model can predict a class or variables by learning decision rules from the training data (Chauhan, 2019). The algorithm starts in the root node and then traverse down the tree. Values from the input data are compared with attributes that are stored in the nodes (called "*decision nodes*") that make forks in the tree structure. These forks divide the data into sub-nodes based on attributes. The branch that

matches the values from the input data is then continued on. We continue traversing through these forks in the tree until a “terminal node” is reached, a decision for the input is then made or predicted (Brownlee, 2020).

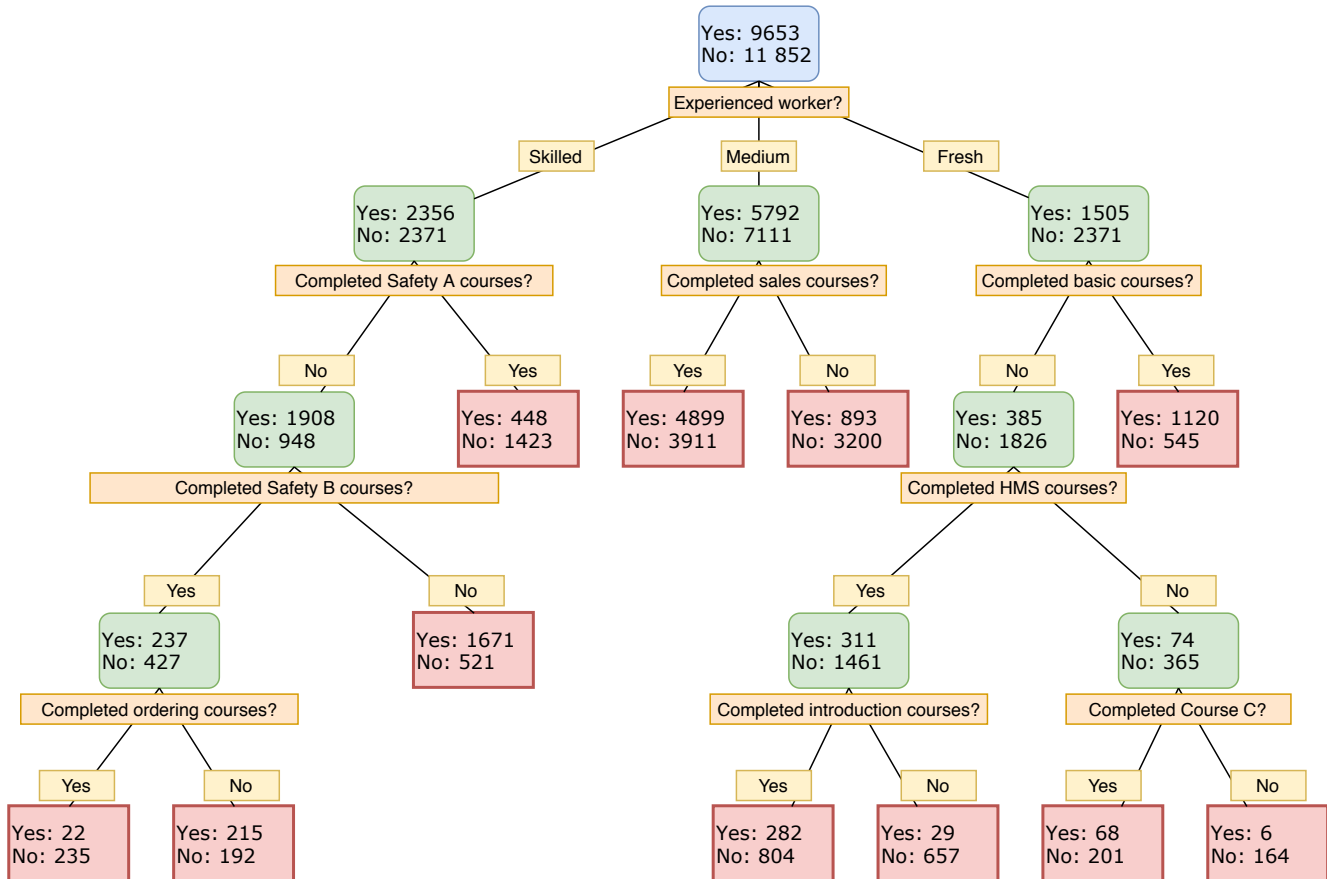


Figure 4: Decision tree for Company A employees showing mock data

To clarify this description, we have made an example; an employee from Company A is considering whether to take a training course or not. A simple case of yes/no output that the decision tree can try to predict the outcome of. There are 220 different courses to take (described in chapter 5.2), with varying numbers of completion degree. Figure 4 illustrated a decision tree, and if we study the root node it says “Yes: 9653”, meaning that 9653 employees have fulfilled all the courses that is expected of them to complete. It also says “No: 11 852” which means 11 852 employees that have not fulfilled all their courses or none.

If we traverse down the decision tree, illustrated in figure 4, we make decisions based on attributes

Terminal node:	Probability
Ordering courses: Yes	0.09
Ordering courses: No	0.53
Course a: No	0.76
Safety A: Yes	0.24
Sales courses: Yes	0.56
Sales courses: No	0.22
Introduction courses: Yes	0.26
Introduction courses: Yes	0.04
Course C: Yes	0.25
Course C: No	0.04
Basic courses: Yes	0.67
Average probability:	0.33

Table 1: Probability of each terminal node

from the input data from the employee. Let’s say he is a fresh employee, that has not completed the basic courses. He has completed the Course R courses as well as the introduction courses. From figure 4, we end up at the terminal node that has “Yes” with 282 occurrences of completion of their courses, and 804 with “No” that just have completed some of their courses. This gives a probability of 0.26 ($\frac{282}{804+282}$) that he has completed all of his required courses.

From this example, we end up with a fairly low probability that the employee has taken all the required courses, and it is, therefore, likely that he will take another course. If we study the probability of all the terminal nodes, we can see that it varies a lot, this is shown in table 1. The average probability of all of these nodes is 0.33, which is also fairly low. Of all the employees, 45% of them have completed all their requirements of courses that are both expected and mandatory. Considering the average of all terminal nodes being 0.33 and not corresponding with 45%, there are other factors that affect this number. For example, the sub-nodes are divided into categories and other attributes that make some of them overlap for several courses, as well as the data, is not too precise when presented in this format.

Considering the results this small example shows, we found that decision trees only solve some parts of the problem. Finding out whether an employee needs a course or not, is not our intended scope. However, we could have used findings when traversing the decision tree. When the results show that it is a high probability that the employee needs a course, we can see whether he has completed courses in for example the sales category, and find out other categories where he lacks skills or competence. The reason we did not choose this solution is that the decision tree requires that the course data is stored with attributes. These need to specify the skill level, what

categories it is in, if it is mandatory, if it has any courses that are required as pre knowledge and etc. Considering the need for such data quality, it is too time- and resource-consuming to be able to complete, as well as the decision tree becoming too complex.

Naive Bayes

The next algorithm we researched was the Naive Bayes algorithm. This algorithm is part of the Bayesian algorithms, meaning these algorithms apply Bayes' Theorem (shown in equation 1), solving classification and regression problems (Brownlee, 2020). The theorem calculates the probability of A happening, given that B has occurred ($P(A|B)$). A becomes the hypothesis that may occur, and B is the evidence. It also uses the probability of B happening given that A has occurred or happened ($P(B|A)$), and the independent probabilities of A ($P(A)$) and B ($P(B)$). Naive Bayes is very useful for large data sets (Ray, 2017).

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \quad (1)$$

Bayes Theorem

For example, let's evaluate whether an employee of Company A has completed all his required courses. Shown in table 2 (displaying mock data), we want to find out if he "has completed all his required courses", this becomes A. The other columns are whether an employee has completed the courses in that category or not and their experience. It is important to assume that the variable is independent and have an equal impact on the outcome (Gandhi, 2018). This means that if for example, an employee is "*skilled*" it does not imply that he has taken all his required courses, and if an employee is "*fresh*" it does not imply that he has completed all the basic courses.

ID	Experience	Basic course	Sales course	Ordering course	Required course
0	Skilled	Yes	Yes	No	No
1	Fresh	Yes	No	No	No
2	Fresh	Yes	Yes	No	No
3	Fresh	Yes	Yes	No	Yes
4	Medium	Yes	No	Yes	Yes
5	Medium	Yes	Yes	No	No
6	Skilled	Yes	Yes	Yes	Yes
7	Medium	Yes	Yes	No	Yes
8	Fresh	No	Yes	No	No
9	Medium	Yes	Yes	No	Yes
10	Medium	Yes	No	Yes	Yes

Table 2: Course completion on employees of Company A

We are first going to just use the condition that the employee is a “*Medium*” experience worker, this becomes the B. The probability that the employee has taken all the required courses becomes: $P(\text{Yes}|\text{Medium})$. Reading the table 2, there are in total 11 employees, so counting all the yes from the “Required courses” column, gives that $P(\text{Yes})$ is $\frac{6}{11}$. Counting the “Experience” column gives $P(\text{Medium})$ being $\frac{5}{11}$. The number of “yes” when the condition is that the employee has “*Medium*” skill is $\frac{4}{6}$; this becomes $P(\text{Medium}|\text{Yes})$. Following the equation 1, gives us the following calculation: $P(\text{Yes}|\text{Medium}) = \frac{P(\text{Medium}|\text{yes}) * P(\text{Yes})}{P(\text{Medium})}$. Substituting the probabilities we found gives: $P(\text{Yes}|\text{Medium}) = \frac{0.67 * 0.55}{0.45} = 0.82$. So if an employee has “*Medium*” experience, there is a probability of 0.82 that he has completed all his required courses.

$$P(A|B_1, \dots, B_n) = \frac{P(B_1|A)P(B_2|A)\dots P(B_N|A)P(A)}{P(B_1)P(B_2)\dots P(B_n)} \quad (2)$$

Bayes Theorem with n conditions

In order for us to use Naive Bayes algorithm, it requires that the data about the employees are stored and structured with labels or attributes that define whether they have completed some courses, what stages they are in, if they are experienced, etc. This is the same problem as we had with decision trees; the algorithm requires a certain data quality. We have to manually create these attributes, and it is too demanding and time-consuming. The courses also have to

be categorized with attributes, the same as the employees, and this is challenging as well. Based on these factors, we did not choose the Naive Bayes algorithm.

K-Nearest Neighbors

The last algorithm we researched was the algorithm called k-nearest neighbours (hereby referred to as KNN). KNN is an instance-based learning algorithm meaning that it compares new instances of data to the training data that is already stored in some database; it can also be called memory-based learning (Brownlee, 2020). Instance-based learning algorithms use this database to search for similar data to compare it to the new data, calculated by a similarity measure. This will find the best match, and it can make a prediction for a classification problem.

KNN can be used for regression problems but is most commonly used for classification problems, which is relevant for our research (Ray, 2017). All data and new data instances are assigned values that map it in a graph. To categorize a new data instance, it uses the labels of the k nearest neighbours. The distance used to find the closest neighbours are calculated using a distance metric such as Euclidean (Srivastava, 2018), which is described in chapter 6.2.1. Figure 5 illustrates k being 3, where 2 out of the 3 closest neighbours have the same labels. This gives a probability of the new label to be 0.67 ($\frac{2}{3}$).

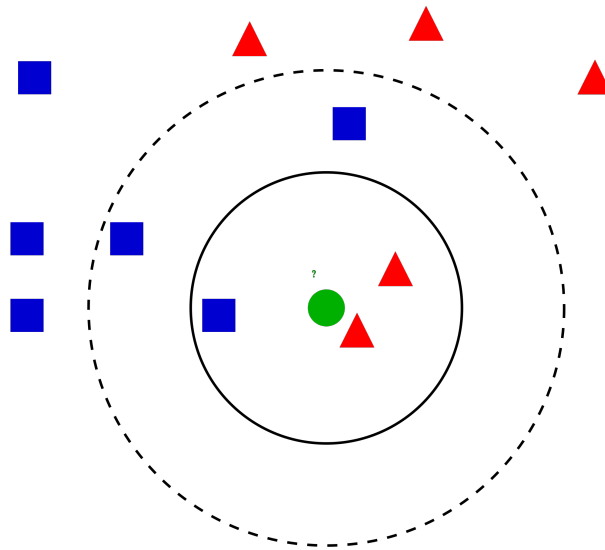


Figure 5: KNN where $k = 3$. Figure taken from Srivastava (2018).

KNN requires a database stored with labelled data to work. This is something we do not have

but can acquire if we manually label all the data fields of courses. However, this will take a lot of time to do, and will also require us to be in charge of choosing the different categories for each course, potentially making the data biased. KNN can to some degree solve our problem or parts of it, where it predicts the category the courses would be in. This can be an issue, but it is not the final solution we need and want to use. Because of this, we did not choose to use the KNN algorithm to label the courses with categories, finding the "best match" category.

However, KNN is a useful way of finding the most similar users (with the lowest distance score), to a given user. In chapter 6, we explain how the first prototype uses a tweak of this approach. In subchapter 6.2, we explain how this tweak uses different distance metrics for calculating distances between data points and users.

Logistic regression

Logistic regression is used when the variables used are categorical, meaning that the algorithm should determine between 0 or 1 (Swaminathan, 2018). For example, the algorithm predicts whether the user is sad (0) or happy (1), and these are the two categories.

Imagine a two-dimensional space with multiple data points. Logistic regression algorithm draws a graph between the data points. Data points above the graph can, for example, be true, and below false. This is illustrated in figure 6.

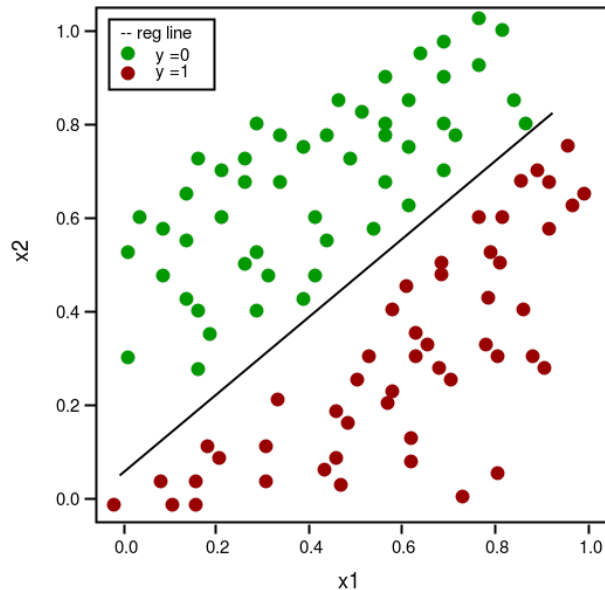


Figure 6: Logistic regression algorithm. Figure taken from Gupta (2018).

Logistic regression uses the probabilities from the data input, and uses the maximum probability on the two output categories to determine the *true* categorical output. The maximum probability can be adjusted with a *threshold* specifying how sure the model has to be before the category output can be determined correctly.

Multinomial logistic regression

Multinomial logistic regression is used to predict the probability of a categorical placement based on multiple independent variables (Starkweather and Kay Moske, 2011). Multinomial logistic regression is an extension of logistic regression that allows for more than two categories of output, hence multinomial (meaning several terms).

As figure 6 shows, logistic regression uses only one graph to determine the categorical output. Multinomial logistic regression uses several graphs, depending on how many categorical variable outputs there are. It uses the maximum probability to determine the categorical output, similar to what logistic regression does.

Gradient descent

Gradient descent is a machine learning technique for trying to find the most optimal set of parameters for a given problem. It is therefore called an optimisation algorithm. The idea behind gradient descent is that it returns a measurement of error, with a given set of parameters chosen at random, and then continue adjusting those parameters until the error minimises itself trying to find the local minimum (Pandey, 2019).

We use multinomial logistic regression with gradient descent in the second prototype described in chapter 7 (implementation chapter). In this chapter, we further describe how we use it and why we chose it.

3.3 Recommender systems

Recommender systems are a subclass of machine learning algorithms that is not an algorithm itself but uses machine learning algorithms (Seif, 2020). These algorithms are an essential part of a recommender system for it to work.

A recommender system is a program that filters and recommends products or content to users based on discovered patterns in their ratings or preferences they have given in the past. The usage of these engines are increasing each year, through advertisements or e-commerce articles,

and are unavoidable when browsing the web (Rocca, 2019). The main concept of recommender systems or engines are algorithms that calculate and suggest relevant items to the user. A large factor to the growth of recommender systems is the economic possibilities it offers business. An efficient recommender system can generate a huge amount of income or outcompete competitors in the same industry. In chapter 3.3.7, we further explore the economic motivations behind these algorithms.

Recommender systems are a concept of their own. They can use supervised and unsupervised learning approaches, but these learning approaches are merely tools for the recommendation system (Loshin, 2016). In that sense, a recommendation system can use, on the one hand, supervised learning to classify items into elements to be recommended or not recommended, and on the other hand, use unsupervised learning techniques such as matrix factorisation to try to predict a suiting item for the user.

3.3.1 How do recommender systems work?

Recommender systems tries to understand the people it recommends items for on an individual level; it can be (1) a customer, (2) visitor to a website and (3) a network of sites that share data between themselves. Either way, the recommender system starts with some data on every "user" and use it to figure out (or predict) each user's unique tastes and interests. This data can be categorised into two types of interest data: (1) implicit and (2) explicit. The two types of data are further explained in chapter 5.1.

Recommender systems can be used to predict items people want — before they know, they want it — based on historical patterns. Data-driven recommender systems find relationships between users and between items based on actions; more or less, without human curation. Recommender systems are not limited to recommending things/items but can also recommend content. The idea is similar but is instead looking at patterns in the content people consume instead of the items people buy.

3.3.2 Filtering methods

Recommender systems methods are typically divided into two major categories: *content-based* and *collaborative filtering methods*.

Collaborative filtering

Collaborative filtering methods use the interaction between items and a user. Recommendations are based on these past interactions (e.g. clicked, watched, purchased, rated, etc.). Figure 7 shows the principle of collaborative filtering. This is by finding an item that *user 1* has watched, and recommending it to *user 2*. These users are so-called “*similar users*”; this is based on their previous interactions with matching or similar items. It is therefore relevant to recommend the items *user 1* interacts with, to *user 2*.

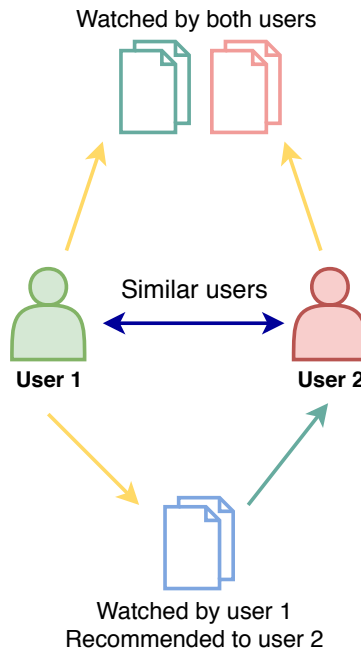


Figure 7: Collaborative filtering

The preference or interactions can be presented as a user-item matrix as figure 8 shows, where each column represent a user, and rows represent an item. Entries (e.g. p_{11}) to the user-item matrix can be numeric or explicit, with for example a rating range from 1 to 5, but are in many cases binary or implicit, e.g. clicked or purchased (Luo, 2018). The majority of entries are usually missing, meaning that the users have not interacted with the item, and the goal of recommender systems are then often to fill those missing entries. These entries are found based on similar items, or based on similar users.

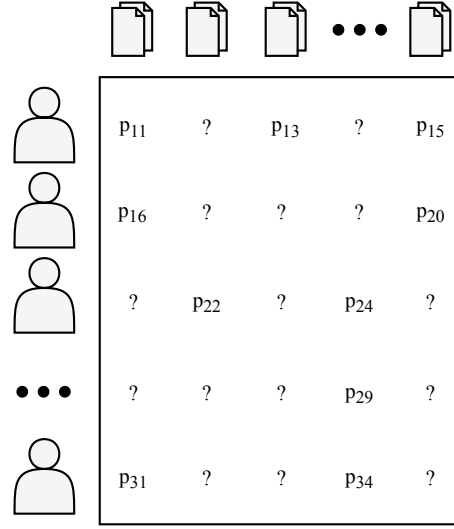


Figure 8: User-item matrix

There are two types of collaborative filtering: *model-based methods* and *memory-based methods*. Model-based methods use matrix-factorisation to handle sparse matrix, meaning they have a lot of 0 values, to predict a user’s rating of an item they have not interacted with (Mwiti, 2018). We are not going to use these types of methods and choose not to focus on it.

Memory-based methods use values of recorded interactions, meaning they do not predict ratings. They are based on the principle of nearest-neighbours (as described in chapter 3.2.3), where the method calculates what the most similar users or items are. Memory-based methods are often more simple to implement and to reason about the results (Mwiti, 2018). An issue that memory-based methods have is the “cold-start problem”. If a user has not interacted with an item, recommendations have nothing to be based on. This is an issue we have to address later.

Content-based filtering

A content-based recommender uses knowledge of each item or user to recommend similar items (similarity of item attributes). Content-based methods are computationally fast and interpretable and can easily be adapted to new items or new users (Deng, 2019). These types of methods use metadata in order to predict recommendations. If we take an example with movies, the method will then use data such as genre, producer, actor, age-restrictions, etc., to recommend new items on.

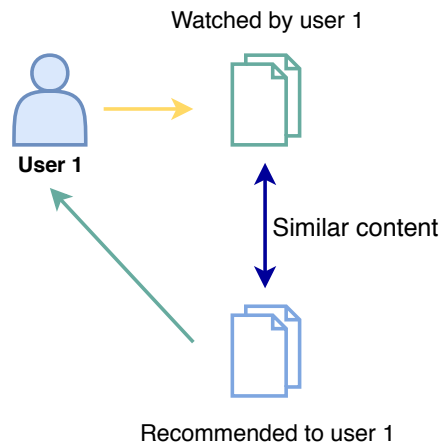


Figure 9: Content-based filtering

Figure 9 illustrates the concept of content-based filtering. For example, let's say a user likes the actor Robert Downey Jr., and has watched all Iron Man movies. Then based on the actor and the genre of the Iron Man movies, it is highly relevant to recommend the Avenger movies to the user. This is because they are very similar. Content-based filtering is based on the idea that if a user liked a certain item, it is likely that he likes something similar.

Content-based are a lot less affected by the *cold-start problem* than collaborative approaches (Rocca, 2019). The reason for this is that even though the user haven't watched any movies, the recommender system can still recommend items to the user based on information that the user has given, such as sex and age. It does not need a lot of information to calculate recommendations, and this is a factor that we need to evaluate when choosing a method.

3.3.3 Top-N recommenders

We can classify a top-N recommender system as a recommender system that produces a finite list of the top-ranked items to present to a given person. This list can be arbitrarily long, but should the recommender system, for example, return a list of three pages each with ten results; it is a top-N recommender where N is 30 (Deshpande and Karypis, 2004).

Research on recommender systems tends to focus on the problem of predicting a user's rating for everything a user has not already rated. Users want to see items they are likely to like and not the recommender systems ability to predict their rating for an item. There are many forms of recommender systems, widgets are for example, usually aggregate rating from other users and

not the rating a system thinks the user would rate the item. Thus, there is a difference in the type of recommender systems to study.

Anatomies of top-N recommenders

Top-N recommender systems can be configured in many ways. Most commonly, there is a data store representing the individual interest of each user. The interest data of each user is normalised using techniques such as *Z-scores* or *Mean Centering* to ensure that the data is comparable between users. Normalising the data effectively is not always possible because the dataset may be too sparse.

During the candidate generation phase, recommendation candidates — items that may be interesting to the users based on past behaviours — have to be generated. Each candidate is then compared to similar items in another data store based on aggregate behaviour. Candidates are then assigned a score based on how the user rated similar items and how strong the similarities are. Should the score not reach a certain threshold, the candidate is filtered out. There exist many different methods of candidate ranking such as "learning to rank" which employs machine learning to find the optimal ranking of candidates at each stage, or take average review score to help advance the highly-rated or popular items (Arya et al., 2017).

Before the top-N list is presented to the user it has to be filtered; filtering phase is used to eliminate recommendations the user has seen, may not want to see or are allowed to see. Stoplists are implemented to remove things that can be potentially offensive or below some minimum rating or quality threshold. Items are then handed to a display layer where a component of recommendations is presented to the user. Usually, candidate generation-, ranking- and filtering live inside a distributed web service that the front-end queries to receive the data required to render a page for a specific user.

Other architectures that may be deployed can be to build up a database ahead of time with the predicted rating of each item by all users. Eliminating most of the work done in the candidate generation phase and thus ranking the items is a matter of sorting them. This approach is not recommended unless the dataset contains a small catalogue of items to recommend because it requires to compare every single item in the catalogue for every single user.

3.3.4 Designing and evaluating recommender systems

It can be challenging to measure the quality of a recommender system. There are different ways to measure quality, and these measurements can conflict with each other. Essentially a recommender system is a machine learning system — it is trained by using prior user behaviour and then predicts which items a user might like. Evaluation of a recommender system can be done through similar methods as any other machine learning system.

Train/Test

The simplest way is to measure the recommender system’s ability to predict how people rated items in the past. As discussed earlier, to keep the results honest, the dataset is split into a training- and test-set. Ratings are assigned randomly into one of the sets, but the training-set should total at least 80% of the data. The recommender system is trained by only using training data — learning the relationships between items or between users. Once the set is trained — it can be used to make predictions about how a new user might rate an item the user has never seen before. Next, the data reserved for testing — ratings that the recommender system has never seen before — to ask the recommender system what it thinks a user would rate an item without telling the answer. Doing this process on a large data set will return a meaningful number to evaluate how good the recommender system is at recommending things people have already seen or rated.

K-Fold Cross-Validation

K-fold cross-validation is the same idea as train/test but uses multiple randomly assigned training sets instead of a single set. Each individual training set or "fold" is used to train the recommender system independently. A measure of accuracy is then achieved by testing the resulting systems against the training set. Each fold then receives a score of how each fold managed to predict the user ratings, and then the scores are averaged together (Sanjay, 2018).

K-fold cross-validation provides insurance against optimising ratings specified in the training set instead of the test set by ensuring that the recommender system works for any set of ratings — not only a few in the training set that was chosen.

3.3.5 Accuracy measures

Finding the best way of measuring how accurate a recommender system is, is very important. In the following sections, we will describe some techniques used to calculate recommender systems

accuracy.

We will use some of these accuracy measures when evaluating the implementation of the prototype we made. We will further explain this in the implementation chapter (chapter 7).

Mean Absolute Error

Mean absolute error (MAE) is the mean or average absolute values of each error in the rating predictions. An error is undesired, and low mean absolute error score is good. Mean absolute error can be calculated by taking each rating, in a test set of n ratings, predicting the value of rating y and comparing the rating y with the actual user rating x . The absolute value of the difference between the predicted rating and the actual rating is then summed up with other errors across the entire test set and then divided by n to get the average or mean.

Root Mean Square Error

Root mean square error (RMSE) is utilised more as RMSE gives penalties when the rating is not close and penalises less when the prediction was reasonably close. Instead of summing all the absolute values of each rating prediction error — the squares of the rating predictions errors are instead summarised. Both absolute values and taking the square of value ensures positive numbers but squaring the value inflates the penalty for more substantial errors.

Essentially root is the square root of the entire set, mean refers to an average, and square error is the square of each individual rating prediction error.

Hit Rate

A "hit" is a recommendation in the top- N recommendations that a user already has rated. The hit is a prediction that the user already found interesting on their own and can be considered a success. To get the Hit Rate, add all "hits" in the top- N recommendations for every user in the test set, divide by the number of users, and that is the hit rate.

An issue by following a train/test or cross-validation approach utilised for measuring accuracy is that these approaches focus on individual ratings, not the accuracy of top- N lists for individual users. Measuring the hit rate directly on top- N recommendations created by a recommender system trained on all of the data would lead to a hit rate of 100%. Evaluating a system by using the data it has trained with is undesirable.

Leave-One-Out Cross-Validation

Leave-one-out cross-validation is a method of computing the top-N recommendations for each user in the training data and then remove an item from the user’s training data. The recommender system’s ability to recommend an item in the top-N list for each user that was left out from the training data.

Issues with using leave-one-out suggest that the dataset has to be large since getting a specific item right during testing is harder than to get a random item of the top-N recommendations. Leave-one-out has the advantage of being more user-focused as users tend to focus on the beginning of lists. Similarly to hit rate but also accounts for where the hits in the top-N list appear. Items in the top slot give a higher reciprocal rank than an item in the bottom slot. A hit in the first slot of the top-N list would yield weight of 1.0, but a recommendation in slot 4 yields a weight of 0.25.

Leave-one-out metric depends a lot on the use case and how the top-N list is displayed. Good recommendations hidden by scrolling or pagination should be penalised by the user’s effort to find the item.

Otherwise, a cumulative hit rank can be a good solution as it throws away hits if the predicted rating is below a threshold instead of crediting the system for the recommendation.

Another take on hit rate is to use the predicted rating score to get an idea on the distribution of how good the algorithm thinks recommended items are, that get a hit.

Cross-entropy

Cross-entropy is commonly used in machine learning as a function for calculating loss. It gives a measure of how different two probability distributions for a set of variables are (Brownlee, 2019). This indicates how *wrong* or what loss the ML model has suffered. It can be used to optimise classification models such as logistic regression.

It is used by a classification model whose output is a probability value between 0 and 1 (Loss Functions, 2017). Classifications that are incorrect are penalised more than classifications that are close. Cross entropy metric is shown in formula 3.

$$-\left(\frac{1}{n}\right) \sum_{i=0}^n Actual \cdot \log(Guess) + (1 - Actual) \cdot \log(1 - Guess) \tag{3}$$

Cross entropy loss metric

High entropy is related to high loss, and low entropy is related to low loss and that probability predictions are more correct.

3.3.6 Evaluating recommender systems

There exist a high diversity of different ways and approaches to evaluating a recommender system. The following sections cover some of the evaluation approaches that we will use in chapter 9 (evaluation chapter).

Coverage

Coverage is the percentage of possible recommendations that the system can provide and can give a sense of how quickly new items will appear in the recommendations. There is a tension between enforcing a higher quality threshold to improve accuracy at the expense of coverage, requiring to find a proper balance.

Diversity

Diversity is a measure of how wide a variety of items the recommender system is displaying. An example of low diversity would be to recommend a course that is next in a succession of a series. In the context of recommender systems is not always desired. High diversity can be achieved by recommending completely random items.

Diversity can be measured by computing the opposite, a similarity between items. Average the similarity scores S of every pair in a list of top- N recommendations and then average the results to get a measure on the similarity. Use the result and subtract 1 to get a number associated with diversity.

Diversity has to be used in conjunction with metrics of quality to avoid bad recommendations as high diversity scores more often than not are just bad recommendations.

Novelty

Novelty is a measure on the popularity of the items recommended. As discussed earlier, recommender systems exist to surface items in the long tail. Most engaged with or sales often come from a small number of items. Items in the long tail — items that cater to unique, niche interests — adds a lot to the overall value.

Users might engage less with the recommended items if the recommendations appear to the user as random. To build the concept of user trust, a few of the items need to be familiar. The user

needs a few *"This was a good recommendation for me"*. Popular items are enjoyable by a large segment of the user base, and these items would be expected to be good recommendations for the user base who has not seen them yet.

Again, there must be a balance between a familiar, popular and serendipitous discovery of the items the user has never heard of before. New items recommended allows the user to discover items they might love, and familiar items establish trust with the user. Recommender systems should be used to help people discover items in the long tail that cater to their unique, niche interests. Other than being an excellent moneymaker for the company, it can help people explore their passion and interests.

Churn

Churn is how often the recommendations for a user change. How often does the user see the same recommendation? High churn in itself is undesirable, and by randomising the top-N recommendations can expose the user to more items they would not have seen otherwise. All of the metrics discussed have to be looked at together to determine the best tradeoffs between them.

Responsiveness

Responsiveness determines how quickly new user behaviour influences the recommendations. Instantaneous responsiveness in recommender systems is complex, challenging to maintain, and expensive to build. Thus, a balance between responsiveness and simplicity has to be addressed.

A/B Tests

None of the metrics discussed matters more than how users respond to the recommendations produced. A surrogate problem is an effect where accuracy metrics in an algorithm are good, but the results fail in a test with real users (Covington, Adams and Sargin, 2016). Accuracy can not be used as a surrogate for good recommendations.

Recommendation systems should be designed with the user in mind. Using A/B tests is useful to measure how users react to recommendations presented to them. Ideally, users are exposed to multiple different algorithms, and then track the users' interest in recommendations presented to them. Continuously testing in controlled online experiments to see whether the recommendations cause people to discover or purchase more new things than they otherwise would, matters and yields the most value for both the users and the business.

Previously mentioned metrics such as accuracy, diversity and novelty can indicate interest from a user but will not replace feedback from users. From a practical standpoint, accurate rating predictions are worthless should the user not find new items to engage with or buy. Replacing an algorithm with a newer, more complex algorithm, should be discarded if users interacting with the recommendations does not lead to measurable improvement. A/B tests with users should be used to avoid introducing complexity that adds no value — complex systems are hard to maintain. Thus, the results of A/B tests with the users are the only metric of success for a recommender system.

Perceived quality

Measuring "perceived quality" can be done by asking the users for explicit feedback recommendations in the same fashion as asking the users to rate items. How to define a "good" rating is not easy, and explicit feedback on recommendation can be advantageous. Though, there are multiple disadvantages with this approach: (1) users can confuse whether they rate the recommendation or rate the item, (2) requires extra work from customers with no clear contribution of value for them, and (3) getting enough ratings on the recommendations to be useful can be hard. A/B testing with interest and engagement is the most precise measurement of quality.

3.3.7 Economical considerations

Economic theory often portrays the goals of business organizations as being related to profit maximization. Hence, business problems and opportunities often relate to increasing revenue or decreasing cost through the design of effective business processes. In the last years, recommender systems have been used for these purposes, to improve the cross-selling effect and to strengthen customer loyalty (Long-Sheng et al., 2008).

Considering the viewpoint of the customer, recommender systems try to suggest helpful or suiting items; however, from the vendors perspective, it is effectively used as targeted advertisement, to increase profit (Das, Mathieu and Ricketts, 2009). The best case for a vendor is that the customer follows their habits and preferences when purchasing or browsing their systems. If the customer actively chooses not to follow the vendor's recommendations, because the "trust" to these recommendations are fading, resulting in the transfer to another business is the worst case. It is therefore vital to be careful when incorporating item profitability or other factors into the recommender system that affect the trust too much.

$$Dice(\vec{r}) = \frac{2\vec{c} \cdot \vec{r}}{\|\vec{c}\|^2 + \|\vec{r}\|^2} \quad (4)$$

Dice coefficient

A good principle is that as long as the vendor uses an algorithm where the recommendations are relatively similar to the customers own ratings, the level of trust will not be too compromised. This can be calculated using, for example, *Dice coefficient* as a similarity measure, shown in equation 4, that measures similarities between two vectors. The vector \vec{c} is the consumer's true ratings for some items, and the vector \vec{r} is the recommendation vector for the same items. A function $T(\vec{r})$ (from equation 4 being $Dice(\vec{r})$) gives a value that indicates the similarity between \vec{r} and \vec{c} , where higher value indicates similarity. These measures are based on the assumption that the trust from a customer is somewhat equivalent to how similar the vendor's recommendations are to the actual ratings from the customer, as well as the assumption that the trust remains on a high level as long as $T(\vec{r})$ remains high.

Another issue with recommender systems is that they tend to recommend already popular products (Fleder and Hosanagar, 2007). If an item has a lot of clicks by a lot of different users, these items will be more frequently recommended to other users. In figure 10, a diagram shows this, where an item with a lot of ratings will get more frequently recommended, and items with few ratings will include the opposite. Jannach and Adomavicius names this problem as "*rich-get-richer*", where these products only increase in number of ratings, based on their frequent recommendations (Jannach and Adomavicius, 2017). This is a problem that necessarily don't need any solution but instead paid attention to in case it affects the business profitability or a reduction in the user base.

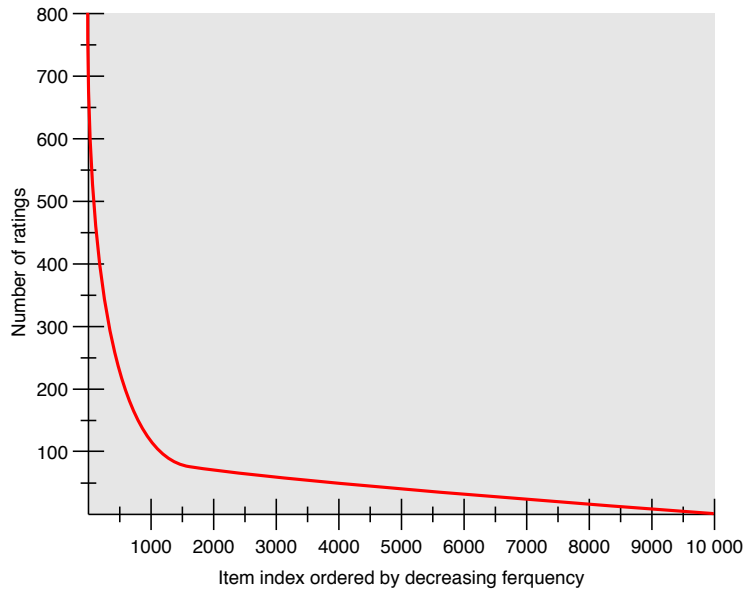


Figure 10: Rating frequency distribution. Figure taken from Liao (2018).

As we mentioned earlier, the “*cold-start problem*” is an issue for recommender systems. This can also affect business profitability. *Cold-start problem* of recommendation methods is that collaborative filtering cannot recommend unrated or unpurchased items (Fleder and Hosanagar, 2007). Jannach and Adomavicius suggest that sometimes it might be better to recommend popular items to avoid this (Jannach and Adomavicius, 2017). Another solution can be hybrid approaches to deal with sparsity (Long-Sheng et al., 2008). These can use information from both user-item interactions and users/items’ characteristics. A hybrid method can become more accurate as more data are available by depending on content-based recommendations when there is no or little activity on a user or an item, where collaborative methods fail to recommend.

3.3.8 Motivation

Regardless of method or approach, when designing a recommender system with a product line approach, the conditional probability will always have to be considered. As a consequence, any change in factors that were not initially designed as a feature in the machine learning component will change the behaviour and performance of the machine learning model. Making rapid experiments and iterations in the product will cause problems with keeping the ML model robust. Hence, should business goals or the original problem and the ML model formulation change, the original ML models, will no longer be valid to support the current business goals. Zinkevich

states that the product's first iteration should be minimally viable since machine learning is one of the most complicated features added (Zinkevich, 2019). Validating and experimenting with the market and building a solid product is more important in the early stages of product development, than having an ML model that can be rendered obsolete by the product evolving or any changes in the business goals.

When it comes to different recommender methods, several aspects have to be evaluated. Whether we are making a prediction algorithm that predicts a users rating for a user-item combination, or a ranking algorithm that presents a list of “*best match*” items to a user. The algorithm needs to avoid “*rich-get-richer*” problem, as figure 10 shows with the *Long Tail Effect*, where niche products do not get recommended to a user (Pandey, 2019). We also have to evaluate whether *relevance* of the recommended items are most important if *novelty*, in a sense, that the users have not seen the item before can be important, or *diversity* among the items in the result list should be something that we emphasize.

Another factor that affects the decision of recommender approaches and methods are the data set we can acquire from Company A and Company B or other forms of mock data. We will come back to this in chapter 5, but the data quality has a lot of impact on what method we can choose and how we can apply them to solving the problem. It will also impact the architecture of the prototype and how components and modules are structured, but we will evaluate this later (in chapter 9).

4 Research process

In this chapter, we describe the research process we have conducted, consisting of evaluation and data analysis methods in our thesis.

First, we present the research process we followed throughout the entire process of conducting the thesis. Then we present how we utilized a framework from design science research on how to conduct multiple design steps. Then we describe how we used a set of guidelines for the research process, also from design science research. And finally, we present the methods we have chosen for evaluation and how we analyse and retrieve data from these.

4.1 Conducting research

Research, in general, is an activity that contributes to the understanding of a phenomenon. In our thesis, those phenomena are SPL and ML. We want to improve our knowledge base on these topics.

There are multiple ways of conducting research, and numerous processes to follow to achieve results and new theories. *Design science research* (hereby referred to as DSR) offers a framework and model for doing so. In DSR, all or part of the phenomenon may be created as opposed to naturally occurring. The phenomenon is typically a set of behaviours of some entity that is found interesting by the researcher or by a group (a research community). Research must lead to the contribution of knowledge, usually in the form of theory, that is valid and new. Valid (or true) knowledge may allow for prediction of the behaviour of some aspect of the phenomenon.

4.1.1 Foundation for the right research process

Our objective in this chapter is to use the performance of DSR in Information Systems as a conceptual framework and guidelines for understanding, executing and evaluating our research.

At the moment, two paradigms characterise much of the research in Information Systems discipline: *behavioural science* and *design science*. Both paradigms are foundational to the Information Systems discipline, positioned at the confluence of people, organisations and technology. The *behavioural science* paradigm seeks to develop and verify theories that explain or predict human or organisational behaviour. The *design science* paradigm aims to extend the boundaries of human and organisational capabilities by creating new and innovative artefacts.

In the design science paradigm, knowledge and understanding of a problem domain and its solution are achieved in the application and building of a designed artefact, to analyse its behaviours.

Focusing on the paradigm of design science is most relevant concerning our thesis. Through the development of a design artefact we want to, not only develop, but research and analyse the possibilities of this artefact, possibly within an organisation. This is the reason why the design science framework is the right choice as a foundation for our research process. Because researching with a DSR framework focuses on the development and evaluating an artefact.

Hevner et al. present a process model that DSR offers, consisting of 5 steps, that can be followed as a general research process (Hevner et al., 2004). The first step is (1) *awareness of problem*. In this step information from multiple sources are gathered, to understand the opportunity for research. Then there is (2) *suggestion to problem*. In this step, the artefact is planned with functionalities and descriptions. The third step is (3) *development*. In this step, the artefact itself is developed according to the previous step. After this step, there is (4) *evaluation*, where the artefact is evaluated based on suiting evaluation methods. The final step is (5) *conclusion*, which is made based on the results from the evaluation step.

4.2 Our process

The overarching process we have been following through the process from start to finish of our thesis, is illustrated in figure 11. Each circle (with a distinct colour) represents a significant step in the process. The process model proposed by the design science framework is a model we have followed and conducted during our thesis, and our research process is inspired by this process model.

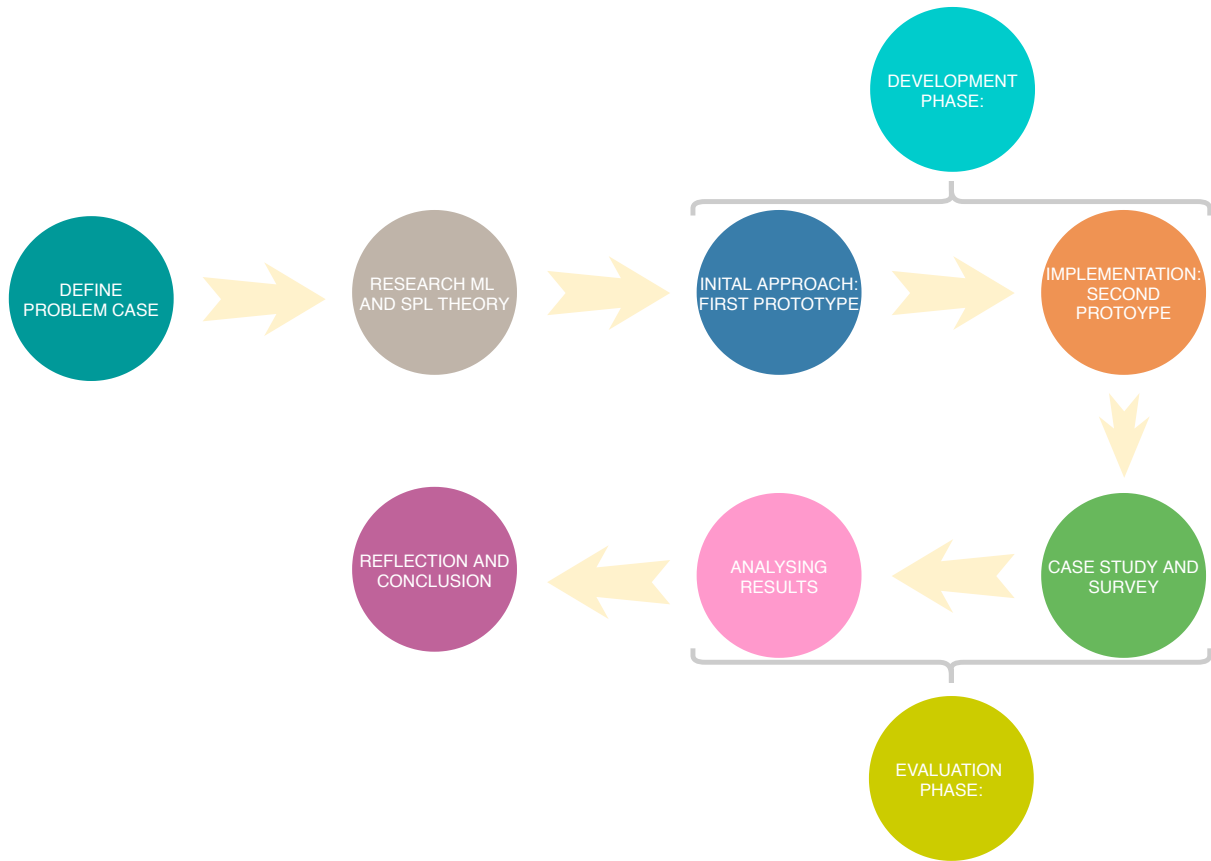


Figure 11: Our research process throughout the thesis

4.2.1 Define problem case

The first step we did was *defining the problem case* (described in chapter 2). This was an essential step of the thesis to understand what areas we wanted to conduct our thesis on, and setting the scope to narrow it down.

4.2.2 Research theory

Because of our problem case regards two major topics of software development and engineering, it was natural to start by creating a knowledge base on these fields. After we had chosen what we wanted to write about in our thesis, we did a substantial *literature study* on both machine learning and software product line theory.

We read articles, journal and websites to gather and structure all the knowledge we found

relevant, and we thought we could use in our thesis. A lot of what we found has been described in the background chapter (chapter 3).

These two steps (define problem case and research theory) are the *awareness of problem* step from the DSR process model. The source of the problem case is Snapper, when they reached out and handed us a case they wanted us to solve. However, they are most interested in the prototype we have made, and not the research and findings we do along the way of making it.

4.2.3 Initial approach

After defining the problem case, we received the product requirements from two different companies that Snapper Net Solutions works with. When we had received this and had conducted a literature study, we sketched and described what the solution to the problem could look like (the artefact), with requirements and components description.

This is the *suggestion to problem* step from the DSR process model. We conducted a creative phase, making suggestions to the problem. This was done before the first development step (initial approach). We redid the suggestion phase after the first development step, and before the second. In retrospect, it is smart to repeat this step as often as possible, conducting it multiple times, to get the functionalities as clear and well-defined as possible.

We started the first - of two, *development phases* by implementing the first prototype. This resulted in an *initial approach* of solving the problem case.

4.2.4 Implementation of second prototype

We saw some issues and flaws in the first prototype that we were not completely satisfied with. After revising some of the product requirements and functional requirements, we started the *implementation phase* of the second prototype.

We did a split of the *development* phase, from the DSR process model, into two phases. This was something we did after conducting the first development phase. We realised that there where a need for refinements to the prototype, so we redid the *suggestion to problem* phase and a new *implementation* phase.

Making a lot of assumptions and concerns about human cognition is not too relevant in our thesis, because we develop a proof of concept prototype showing just the possibilities of it. It is

not meant to be tested directly on actual users in a relevant setting.

4.2.5 Evaluation

When we had a second prototype that we were satisfied with, we started the *evaluation phase*. We conducted a *case study* of the prototype we had developed, with three different experts. We also made a *survey* that we published, asking questions about the use of and trust people have in recommender systems.

Completing the case studies and receiving answers on the survey we had published gave us a lot of results. We, therefore, *analysed the test results* to see if we had what we needed and if they were valid or not. We structured and compared the results.

The case study and survey, and analysing the results is the *evaluation* step from the DSR process model. It states that once the prototype has been constructed, it should be evaluated according to hypothetical predictions or criteria made in the *awareness of problem* phase. Sometimes, it is relevant to repeat the *suggestion* and *development* phase to change how the prototype behaves. Observations of changes to the test results or hypothesis are normal, and should also be considered in the evaluation Hevner et al., 2004).

We did a minor *evaluation* phase after the first *development* phase (initial approach). It was this phase that resulted in a new implementation phase to refine the prototype. We saw it lacked some functionalities that we wanted to be implemented. The first prototype did not have all the elements and proper architecture we wanted, and it, therefore, did not behave the way we wanted it to do. After the second *implementation* phase, we conducted the main *evaluation* phase consisting of the case study and the survey.

4.2.6 Reflection and conclusion

The final step of our research process was a *reflection and conclusion* step. After we had conducted the evaluation phase, we had enough results to conclude the research process.

This is the *conclusion* step in the DSR process model. We evaluated the results against our hypothetical predictions on the behaviour of the prototype and other assumptions we had made during our research, looking for deviations and other implications. We finished this step by concluding on different outcomes on the research questions (presented in chapter 2.3). We also presented the knowledge and lessons learned on the different steps in the research process.

4.3 Design science research

Design science is an outcome-based information technology research methodology, which offers specific guidelines for evaluation and iteration within research projects. Design science research (DSR) focuses on the development and performance of (designed) artefacts with the explicit intention of improving the functional performance of the artefact. DSR is typically applied to categories of artefacts including algorithms, human/computer interfaces, design methodologies (including process models) and languages. Its application is most notable in the Engineering and Computer Science disciplines, though it is not restricted to these and can be found in many disciplines and fields.

We will describe these guidelines further in section 4.5 that we have been using. Science of the artificial (design science) is a body of knowledge about the design of artificial (human-made) objects and phenomena (artefacts) designed to meet certain desired goals. Whereas the traditional natural sciences focus on a class of things — objects or phenomenon — in the world (nature or society) that describes or explains how they behave and interact with each other.

In other words, design science is knowledge in the form of constructs, techniques and methods, models, and theory for achieving this connection. The know-how (procedural knowledge) for creating artefacts that satisfy given sets of functional requirements. Design science research is research that creates this type of missing knowledge using design, analysis, reflection, and abstraction.

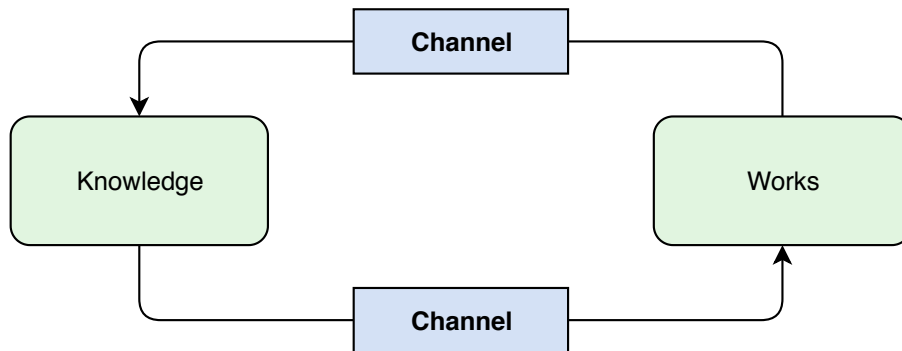


Figure 12: Model:knowledge is generated and accumulated though action

We can draw the following model, in figure 12, as a basis to understand how knowledge is generated and accumulated through action. Figure 12 illustrates how creating something and

then evaluating the results helps to build knowledge; this is the process that we do when we develop the artefact and testing it afterwards. Furthermore, abstraction and reflection are also mattering in the knowledge building process. Building knowledge through construction is sometimes considered lacking rigour, but the process is not unstructured. The channels in figure 12 of the general model are systems of conventions and rules under which the discipline operates. They typify the measures and values that have empirically developed as 'ways of knowing' as the discipline has matured.

Our objective in the thesis is to use design science research primarily as a research method when conducting our research. Knowledge and understanding of the problem domain and its solution are achieved in the application and building of the designed artefact. Combining the capabilities of the Information Systems and characteristics of the organisation, its work systems, its people, and its development and implementation methodologies determine the extent to which that purpose is achieved.

Down the line, we will have to be aware of the risk in not distinguishing our work from a design effort; simply creating state-of-practice design, and not research. Design science research (within its community of interest) is distinguished from a routine design by the production of interesting new and true knowledge. Though routine design can lead to DSR. To find out the missing knowledge in a new area of design, it is quite useful to attempt carrying out the design using existing knowledge — giving a better feel for the number of unknowns in the proposed design (missing knowledge).

4.4 DSR framework

Hevner et al. present a framework that structures different components, combining the concepts of behavioural science and design science (Hevner et al. 2004, p. 80). It maps out the relations and connections between the components and illustrates how they interact with each other. We have structured all of this in figure 13, to present our chosen methods and components.

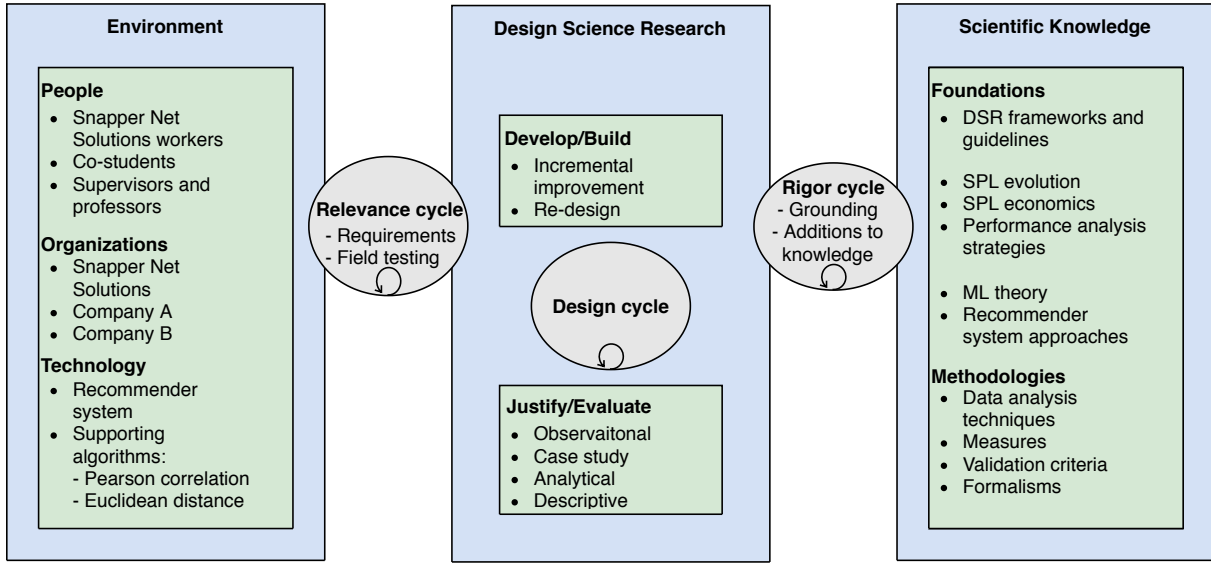


Figure 13: Design Science Framework

DSR proposes a set of steps or activities and through these are the artefact developed, based on organizational structures and requirements on a common knowledge base. Figure 13 illustrates some of these cycles. Each cycle is typically iterated a number of time before the artefact is designed and complete. The *Relevance cycle* is to assure relevance from the environment, where the business requirements and other demands are specified. *Rigor cycle* is to ensure that the appropriate foundations and methodologies are used, and other additions to the knowledge base might be added here.

Environment section

The environment section describes what actors (*People* and *Organizations*) that have relied on and cooperated with. For example, we used supervisors and professors for guidance on the thesis, and the workers of Snapper Net solutions was good support for evaluation and feedback on the artefact. Company A is a working partner with Snapper Net solutions, and since we cooperate with Snapper, we relied on data providence from Company A. The *Environment* section describes the organizational strategies and structure that the artefact needs to be evaluated on.

Design Science Research section

As we described in chapter 4.3, the design is both a process and a product (the artefact). In figure 13, in the *Design Science Research* section we have a *Develop/Build* box that describes different

design models we have used. We *incrementally* improved our prototypes, and *re-designed* the first prototype, into a second version. In the *Justify/Evaluate* box, the evaluation methods we have conducted are listed. The *Design cycle* between the boxes illustrates the process of going back and forth between evaluation and development. First, we develop the product, then we assess the evaluation of the artefact, to then refine it and start re-developing again.

Scientific Knowledge section

Scientific Knowledge is a section that describes our fundamental knowledge that we based our thesis and our research on. This includes all the models and current theories on topics that already exist out there. It also includes the methodologies that we chose to use as tools when conducting and documenting our research. For example, was it necessary for us to choose different data analysis techniques to use after we had conducted our evaluation, in order to produce results.

4.5 DSR Guidelines

In the article “*Design Science in Information Systems Research*”, Hevner et al. have described seven guidelines for the design science research process (Hevner et al. 2004). Design science concerns the development of knowledge and understanding of a design problem. It is through building an artefact and finding solutions to this problem that knowledge is acquired, the seven guidelines are derived from this aspect. These create a template that we used as a foundation for our research process. In the following chapter, we describe these guidelines and how we have applied them to our research.

4.5.1 Guideline 1: Design as an artefact

An artefact is an innovation that defines the ideas, practices, technical capabilities and products, that can be analysed, designed and implemented, and be used or accomplished in information systems effectively and efficiently (source).

“Design-science research must produce a viable artefact in the form of a construct, a model, a method, or an instantiation.” (Hevner et al. 2004). The artefact must address or try to solve an important organizational problem. For the artefact to be of value, it needs to be applied or implemented in an appropriate domain. By studying the definition from Hevner et al., it does not need to be a standalone product or system, the artefact can be a simple model or an instance of a larger system that solves a specific problem of varying size. Any models, methods

or constructs that were used in the development phase should also be part of the artefact.

In chapter 2 we described the research problem that we want to conduct our research on. Part of the research questions is to see if it is possible or beneficial to solve this problem. In order to do so, we wanted to develop a prototype or proof of concept. Part of this prototype is a generic recommender system that is a component. Exploration and development was done by mainly focusing on this component, but the artefact itself is all the components, modules and constructs that we have developed during the process. We hereby refer to the artefact as the prototype.

4.5.2 Guideline 2: Problem relevance

Having a common understanding of the relevance of the chosen topic for our thesis is important. Hevner et al. state that *“the objective of design-science research is to develop technology-based solutions to important and relevant business problems.”* (Hevner et al. 2004). When conducting work for a company (Snapper Net Solutions), they have already stated their need for a solution, which imply relevance for the research where doing. However, it is not relevant for them all the other time we spend researching similar topics through articles. This is what motivates us because we see the relevance of exploring a topic that has not been heavily researched.

A problem can be defined as the differences between a goal state and the current state of a system (Hevner et al., 2004). We will further describe the different states of the prototype in chapter 6 (initial approach) and chapter 7 (implementation) that we reached when starting from scratch. Solving a problem is defined as a search process using actions to reduce or eliminate the differences; this is further explained in guideline 6 (in chapter 4.5.6). Artefacts are designed, constructed and used by people and are therefore shaped by values and interests from communities or investors (Orlikowski and Iacono, 2001). There is always an economical approach and motivation behind a business need or requirement; this increases the problem relevance.

4.5.3 Guideline 3: Design evaluation

Evaluating and testing the prototype is necessary when conducting research in order to produce some results or indications. *“The utility, quality, and efficacy of a prototype must be rigorously demonstrated via well-executed evaluation methods.”* (Hevner et al., 2004). It is important that the evaluation is based on some metrics or criteria and in a strict environment. The “Publication

Scheme for a Design Science Research Study” (Gregor and Hevner, 2013, p. 350) also points out validity as an important criterion to prove for the prototype. This, along with the other criteria, gives evidence whether the prototype is useful.

Evaluation should be based on the following:

- Business environment establishes the requirements upon which the evaluation of the prototype is based
- This environment includes technical infrastructure which itself is incrementally built by the implementation of new IT prototypes
- Evaluation includes the integration of the prototype within the technical infrastructure of the business environment

Figure 14 is listing all the *Design Evaluation Methods* DSR offers. The highlighted ones are the ones we have chosen to conduct.

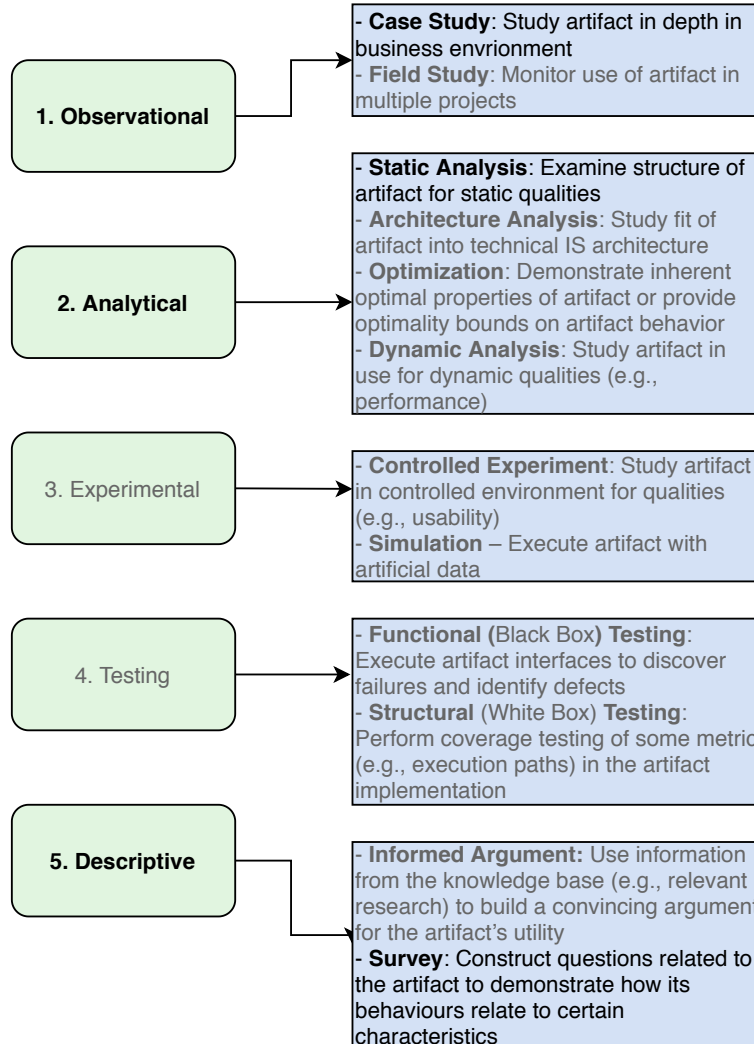


Figure 14: Design Evaluation Methods. Figure taken from Hevner et al. (2004), p.86

We find *observational* evaluation with a case study to be suited for our prototype. A case study can be a realistic way of analysing the prototype in an appropriate environment, where functionality, usability, accuracy, etc can be measured.

We also think that an *analytical analysis* of the codebase can be appropriate and possible to do. By doing so, one need experienced supervisors on machine learning or SPL to evaluate the software architecture and code base, to give feedback while we take notes.

A *descriptive* evaluation of the prototype based on the knowledge base and creating some sce-

narios may also be relevant to conduct. This can give indications on whether it fulfils some theories or to demonstrate the utility of the prototype.

In chapter 4.6, we will further present what methods for testing we have chosen, and how we retrieve data from it. In chapter 8 (evaluation chapter), we will present the test results we received after conducting the tests.

4.5.4 Guideline 4: Research contributions

“Effective design-science research must provide clear and verifiable contributions in the areas of the design artefact, design foundations, and/or design methodologies.” (Hevner et al., 2004).

The research must have clear and well-defined contributions. It is therefore important to ask: “What are the new and interesting contributions?”. There are three types of research contributions based on novelty, generality and significance of the prototype. At least one of these contributions must be discovered or accounted for. The contributions are:

1. **The prototypes** - Often it is the artefact itself, designed to extend knowledge base or apply existing knowledge in new ways
2. **Foundations** - all the constructs, models, methods or instantiations that extend and improve the existing foundations in the design-science knowledge base
3. **Methodologies** - use of evaluation methods, (examples: experimental, analytical, observational, testing, descriptive). This will produce data and results.

Our contribution is mainly the prototype and the knowledge gained and lessons learned when developing and evaluating it. We have researched what it is possible to utilize it for and for what purpose. This has contributed to a knowledge base on the research area as well as the prototype. The data and the results we produced during the research process is also part of the contribution.

4.5.5 Guideline 5: Research rigor

Hevner et al. state that the *“design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artefact.”* (Hevner et al., 2004).

This implies that to produce well-tested research, the process that leads to the results has to

be strictly conducted based on accuracy. It is also important that the data validity and results adhere to the analysis techniques. The research results need to be thoroughly analysed and checked concerning the knowledge base and existing research; this is the main point of rigour research (Peppers et al., 2008).

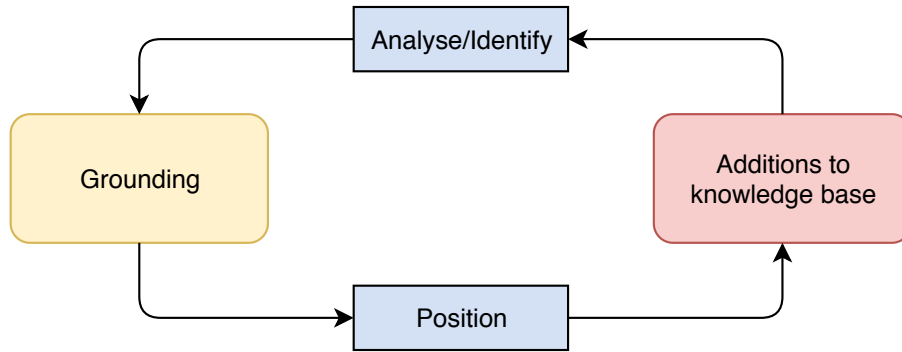


Figure 15: Rigor cycle

Figure 15 illustrate how we have performed our research rigor cycle. When producing results, we have continuously checked and analysed these against the knowledge base to see if it was reasonable and possible results, this is called “*grounding*”. If the results cohere with existing theories, it is positioned in the knowledge base, if not it will also give additions to the knowledge base but for example, as falsifications. We used the rigour cycle, mainly in the test and evaluation phases.

4.5.6 Guideline 6: Design as a search

Problem-solving can be viewed as utilizing available means to reach desired ends while satisfying laws existing in the environment (Hevner et al., 2004). This is the essence of design as a search. Through designing and developing a prototype, the research process is a search for doing this most effectively and in the best way.

- **Means** are the set of actions and resources available to construct a solution
- **Ends** represent goals and constraints the solution should satisfy
- **Laws** are uncontrollable forces in the environment or restrictions that prohibit or limit the prototype in some ways

Effective design requires knowledge of both the application domain (e.g. requirements and constraints) and the solution domain (e.g. technical and organizational). Means, ends and laws are iteratively refined to be made more realistic. As a result of this, the prototype becomes more relevant and valuable. These iterations of *"best optimal design"*, can be done through a generate/test cycle (Presthus et al., 2016).

In our thesis, the means is for example the data sets, helping components and algorithms that we have used when developing and constructing the prototype. The ends are all the functionalities the final prototype have. This defines the goal state of the prototype and how it turned out. The laws are the restrictions of the prototype we met prior to and while developing it. Often, there can be limitations in the data sets, other information that is sensitive and requires a non-disclosure agreement for one to use, and etc. Mapping out the means, ends and laws helped us to structure the research process, and the environment we developed the prototype in.

4.5.7 Guideline 7: Communication of research

When completing a research process and producing valuable results, a vital factor of the process is publishing it to other researchers. A thorough and complete article of the research is essential, but it is also important how to communicate it and to what audience. *"Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences."* (Hevner et al., 2004). Rarely, it is just one specific audience involved or having an interest in a research field; this is important to take into consideration.

We have stakeholders both at the technology-oriented and management-oriented audience that have an interest in our work. People at Snapper and at UiO require different approaches to the case solution and other findings. In the following chapters, we will be presenting at different strategy levels and views to cover a wider area of readers.

4.6 Methods for collecting data

We have chosen two different methods for collecting data. The reason for this is to get feedback from a larger target group, with a broader perspective on the topics we research, to get more insights. Through collecting empirical data, we have had an inductive approach to the research methods we have chosen (Sander, 2019).

As described in 4.5.3 we have chosen an observational evaluation with a *case study*. This can

give a good way of analysing the prototype with fitting people, and we can analyse measures such as reusability and functionality. Part of this observational evaluation will be an *analytical analysis* with an *interview* where we study the code base and architecture. This is where the people are asked questions so we can get feedback and results on the prototype.

The second method we chose was also an observational evaluation. We made a *survey* that we published to receive a response from a broader target group. We wanted to get more insight into peoples attitudes and experience towards recommender systems.

We did not find a *descriptive* evaluation (as mentioned in chapter 4.5.3) of the prototype as fitting. Creating scenarios for the prototype was considered, but we thought it would not provide any valuable results for our research. This is because it requires the prototype to be placed in different environments or settings, and this is not necessary as it is only a proof of concept.

In the following sub-chapters (4.6.1 to 4.6.3) we explain how we conducted these evaluation methods, how we learnt from them and how we analysed the results we got from conducting them.

4.6.1 Case study

To evaluate the prototype, we used the BAPO model, illustrated in figure 16, to show what scope or perspective we focused on when conducting the observational evaluation. BAPO model represents the four concerns that need to be addressed with SPL engineering (BAPO Model, 2011). We do not focus on the organizational view, as this is out of the scope (not connected to the research questions). It is also not relevant for the context of Snapper, because it is a relatively small company.

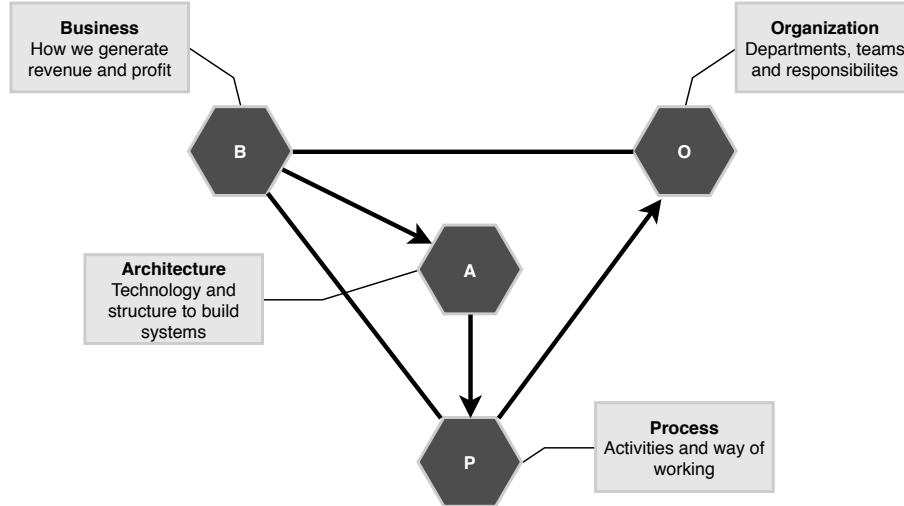


Figure 16: BAPO model. Figure taken from Bosch (2017).

The case study was conducted with three test subjects. Two of them are specialised software developers, and the last one is CTO at Snapper. The CTO could give feedback from a business and process point of view because he has experience with leading larger projects. The software developers gave technical feedback from an architectural view, on the codebase.

How the study was conducted

One of the interviews were conducted over Whereby because of the COVID-19 virus. Two of the three interviews were conducted in person, where we met the test subjects. Because of the situation, it was not possible to do the last of the interviews in person, as we wanted to. However, it was still very possible to conduct the entire process of the evaluation method, and we got the full extent of the results we wanted out of the case study.

When we met the test subjects, we started by letting them analyse and understand the code base of the prototype. This first phase, we named the *observational phase*. While the test subjects analysed the prototype, we had a passive role where we, for the most part, took notes, and observed. They could always also ask us questions if they wanted to.

The next phase was the *interview phase* of the case study. Here we had prepared a set of questions based on what viewpoint we wanted to study (from BAPO model in figure 16). We could also ask unprepared to follow up questions that we felt were necessary to get more insights into their thoughts and opinions on the prototype. We took notes during this phase as well and

filled in missing parts we did not have time to do during the interview after we were done.

How we chose test subjects

It can be tough to have access to relevant and suiting interviewees and test subjects (Cragg and Cook, 2007). This is because they should be able to observe analyse and answer questions about our prototype, that is both relevant and useful to us. We got help from Snapper Net Solutions on finding people that could do this. Two of the people we interviewed are employees from Snapper, and the last one is a software developer at a consultant company called Bouvet. We contacted the different people directly, informing them about the thesis and research we were conducting, before presenting them the case. After this, we received their acceptance, and we then agreed on a meeting time and place.

The reason that it was challenging to find test subjects was that the people would preferably have a lot of experience in both SPL and ML. We could not find this, but the software developers we used had a lot of experience in developing software and some experience on SPL. Both had a lot of interest in ML, and some experience there as well. The CTO had more experience with SPL, and this was very important because we needed an analysis from a business and process point of view. The CTO was familiar with some of the topics related to ML as well.

The way we have found and recruited the test subjects is called *theoretical sampling* (Cragg and Cook, 2007). This is because the people we use for evaluation are chosen because of specific characteristics they have and the quality of information they, therefore, can offer to the evaluation methods. *Theoretical sampling* is valuing quality rather than quantity and diversity in the test subjects. This was done because it was the most convenient way of evaluating the prototypes as well. Having several extra interviews, with multiple experts would not be beneficial because it is too time-consuming.

Interviews

Conducting interviews has been the primary evaluation method for collecting data to our research from the chosen test subjects. The importance of interviews in research is because it provides insight into how the test subjects interpret and understand some events or objects (the prototype) they are exposed to (Walsham, 2006). This gives valuable information we can retrieve and use when trying to answer the research questions. Interviews were suiting for our research because we could go in-depth with the experts we interviewed, to understand their aspects and opinions on the prototype they analysed.

Interviews can be conducted structured, semi-structured or unstructured, depending on the purpose and the wanted outcomes of the interviews (Crang and Cook, 2007). This depends on how predetermined questions you want to ask and how much control the interviewer wants to have in the conversation (Pollock, 2019). We chose to conduct semi-structured interviews, with a lot of predetermined questions, and some follow up questions as well. This is because we had some questions we wanted the answer to, and some that felt natural to ask during the conversation, some aspects and topics that we wanted to elaborate and go deeper into.

As mentioned earlier, we conducted two types of interviews, focusing on getting different information on different viewpoints illustrated in figure 16. The first type of interview is focusing on the architectural viewpoints of our prototype, and the second type is focusing on the business and process viewpoint. This is to get a wider perspective on the aspects of SPL engineering with ML. Our research questions (from chapter 2.3) focus on a broader perspective than just the technical part of merging the two concepts, it also focuses on the evolution and cost-beneficial aspects of it.

The templates for the different interviews we conducted can be found in appendix B. The interviews with the architectural viewpoint we conducted with software developers can be found in appendix B.1. The interviews with business and process point of view conducted with the CTO of Snapper can be found in appendix B.2.

Observation

To avoid bias when collecting data, it is essential to conduct multiple forms of evaluation methods (Walsham, 2006). During the case study of the prototype, we conducted not only an interview but an observational study before that as well. Observational studies are often conducted to understand *how* the test subjects react to some tests or objects (Crang and Cook, 2007). Observations can either be conducted in a natural or controlled environment and can be direct or indirect meaning whether the observer is present or not (Anguera et al., 2018). Indirect observations can be conducted through audio recordings or logs that are read.

Conducting observation of this prototype in a natural environment was not relevant because it is only a proof of concept, and not a final product ready to be deployed into production. Conducting a direct observation was challenging (further explained in the next section). We had a passive role while directly observing the test subjects while they analysed and tested the prototype. We did not collect too much data out of this process, and it was merely to see how

quickly they understood the code base and architecture. Observations were to get a picture of how understandable and easy it was for the test subjects to comprehend how the prototype was structured and the different components worked together. That is why we had a passive role and did not want to interject in the process of their *first experience* with the prototype. The degree of passiveness could vary slightly from the different test subjects. This was depending on how many questions they had and if we were unsure if they had understood something.

We took notes during the observation phase that we later came back to in the interview. Together, we discussed with the test subjects about their experience and understandings of the prototype.

4.6.2 Survey

We published a survey through Google Forms, consisting of 11 questions. The survey included questions related to recommender systems. Users are asked to answer if they understand what recommender systems are, how often they are exposed to them, and other questions related to trust and items the systems recommend, and the area of usage. The survey is related to research question R5 to see how a software product line affects the quality of a recommender system.

A survey is a form of an observational evaluation method, and it is indirect because the conductors of the study are not directly involved but study the results through recordings (Anguera et al., 2018). It can also be called a retrospective observation because the results are based or recorded on past events (Mitra, 2020).

When creating surveys, biases often occur and are a big issue. The reason that they occur is that unfairness undergoes the different options that the users can choose. The parameters can often be statistically undermentioned. The actions we took to avoid bias in our survey was to make it anonymous, not creating leading questions that were neutral, have similar scaling parameters on all questions and allowing the users to withdraw from the survey at any time. The survey was also not directed at a specific target group but was open for everyone to answer.

The process of doing the survey was conducted with an expert in statistics. He helped us formulating the questions with clear and distinct options and a language that was easy to understand. We created several drafts and versions of the survey that we sent to him and received feedback on, before publishing the final version.

The survey questions we published through Google Forms can be found in appendix C.

4.6.3 Methods for data analysis

When collecting data through evaluation methods, it is imperative to analyse the results. An analysis is a creative process that should be structurally and thoroughly conducted (Cragg and Cook, 2007). Several different methods for data analysis exist for both qualitative and quantitative data. In the following sections, we explain what methods we used for the (1) case study (qualitative data) and the (2) survey (quantitative data).

Content analysis

Retrieving absolute meaning from qualitative data is difficult, because it is made up of observations and words; therefore, it is mostly used for exploratory research (Bhatia, 2018). After conducting the case study, we were left with notes and answers from the observation and interview, resulting in three different texts. Bhatia suggests four steps before and while analysing the text, in the following order (Bhatia, 2018):

1. **Get familiar with the data:** we read the results and notes so we were completely familiar with the content in the texts.
2. **Revisit research objectives:** we read over the research questions, so we had a clear and common understanding of the research objectives, and what questions that needed to be answered.
3. **Develop a framework:** first we read the texts to mark behaviours, opinions and certain phrases or answers. Then we assigned codes to them (color codes). This is to structure and label the data in the text. Each color code is related to one research question (shown in figure 17).
4. **Identify patterns and connections:** then we read the text looking for patterns and connections between answers. We also looked for an overall theme, which substantiate and lead the answers in one direction.






-  RQ1: How possible is it to create machine learning components that work for multiple products in a software product line?
-  RQ2: How reusable are machine learning components in a software product line?
-  RQ3: How feasible is it to create and consume reusable machine learning models in a software product line?
-  RQ4: How can you support a software product line evolution containing machine learning components?
-  RQ5: How does a software product line affect the quality of its recommender systems?

Figure 17: Color codes mapped to research questions

We did these four steps for each text and conducted what is called a *content analysis* (Bhatia, 2018). In such an analysis format, you analyse the content of texts or notes based on some research questions. Figure 18 shows an example of how we analyse the interviews with each color code representing a different research question (from figure 17). After we had done a content analysis of each text, we compared the structured and labelled data to conclude.

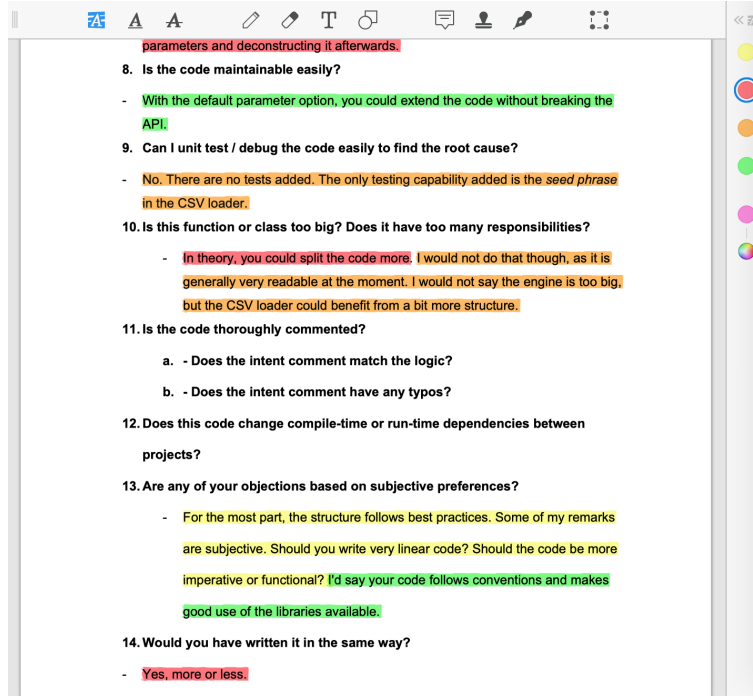


Figure 18: Example of how we analyse interviews

Descriptive statistics

The results from the survey can be represented or illustrated in many ways. We have chosen to use *descriptive statistics* to illustrate the results from the survey. The most commonly used descriptive statistics are: (1) *mean*, (2), *median*, (3) *mode*, (4) *percentage*, (5) *frequency* and (6) *range* (Bhatia, 2018).

We will be using (1) *mean* (average answer), (3) *mode* (most common answer), (4) *percentage* (answer distribution) and (5) *frequency* (how many times an answer is found). Percentage and frequency are the most important descriptive statistics we will be using, as these shows where the majority of the feedback can be found, giving the best indication of what the respondents' opinions are. This is illustrated in figure 19, were we show a pie chart with a distribution of different answers. Each slice in the pie chart represents an answer.

How often do you think you are exposed to a recommender system?

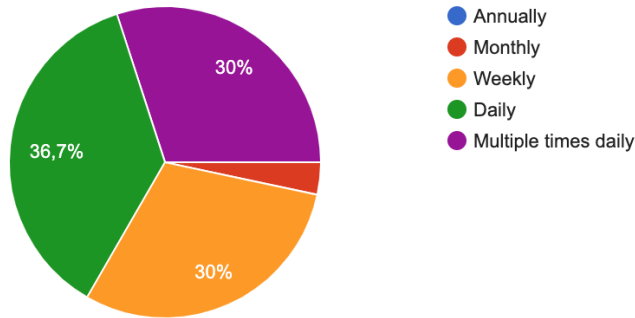


Figure 19: Example of pie chart from survey

In chapter 8 we present the results that is based on the evaluation methods and the background for data analysis methods presented in this chapter.

5 Product foundation

In this chapter, we present the different data sets we worked with during the implementation of the different prototypes we made. Then we present a foundation for structuring and creating an architecture for the different prototypes. And finally, we present requirements elicitation in SPL, how you research and discover requirements for different products and how they are structured within the same SPL.

5.1 Data access and gathering

Before developing the first prototypes, described in chapter 6 (initial approach), we had received no data. At this stage, we had no employee data from neither Company A nor Company B. In order to test the prototype, we generated two sets of mock data for simple user testing. The two sets consisted of explicit and implicit data we generated about 10 fictional users. We could have generated more data, as it is easy to randomize ratings and courses they have taken, but we did not find that necessary to provide the results we needed.

5.1.1 Explicit data

Explicit data is collected when users are asked to give feedback on an element. It is called explicit data because users are explicitly asked for feedback on an item. Acquiring or obtaining this type of data are usually completed by asking users to rate a type of item often in terms of "stars" in a *scale of one to five*, give the content a *thumbs up* or *thumbs down*, or by *rating content* they see with reactions (Pandey, 2019). In other words, asking the users "*Do you like what you look at?*", and use that data to build a profile on that user's interests. There is, therefore, no need to make any assumptions about the user's preferences.

Issues with explicit data are that (1) data often tend to be too sparse, leading to low-quality recommendations, because some users do not leave feedback and others do. (2) Those subjective standards on feedback and rating differ a lot based on personality, country and age. The issue is that the ratings and the feedback mean different from user to user. For example, can a 5-star rating mean that the user "*loved*" the item, but to another user, it meant that he was "*just satisfied*" with it.

Figure 20 shows the mock data we created about the 10 employees. After each course is the rating represented. For example, did Sebastian rate "*Safety A*" to be a 4.5 out of 5-star course.

It is easier to calculate similarities between users when we use the explicit data because we have more to base our calculations on.

```
"Sebastian": {
  "Safety A": 4.5,
  "Safety B": 5.0,
  "Course A": 4.5,
  "Course B": 4.5,
  "Course C": 4.0,
  "Course D": 3.5,
  "Course E": 4.0,
  "Course F": 3.0,
},
"Jorgen": {
  "Course G": 5.0,
  "Course H": 5.0,
  "Course I": 4.5,
  "Course J": 4.0,
  "Course K": 3.5,
  "Course O": 3.5,
},
"Edvard": {
  "Course L": 4.5,
  "Course M": 4.0,
  "Course N": 4.0,
  "Course C": 5.0,
  "Course O": 5.0,
  "Course P": 4.0,
  "Course J": 5.0,
}
"Mats": {
  "Course R": 4.5,
  "Safety B": 4.5,
  "Safety A": 5.0,
  "Course B": 4.0,
  "Course K": 4.5,
},
"Lars": {
  "Course F": 5.0,
  "Course Q": 4.0,
  "Course R": 5.0,
  "Course B": 4.5,
  "Course K": 4.0,
},
"Roar": {
  "Safety B": 3.5,
  "Course C": 2.5,
  "Course P": 3.0,
  "Course A": 4.0,
  "Course Q": 3.0,
},
"Hans": {
  "Course F": 2.5,
  "Course K": 2.0,
  "Course R": 5.0,
  "Course L": 4.5,
  "Course C": 2.0,
  "Course N": 4.0,
}
```

Figure 20: Explicit mock data

5.1.2 Implicit data

Implicit data is collected from a user's interaction with elements or sites, it is then interpreted as indications of interest or disinterest (Pandey, 2019). An example is clicking on an item in a web store, this is considered as a positive interest from the user, and it is, therefore, relevant to use this for future recommendations. It is easy to gather a lot of implicit data, because of all the roaming and clicking users do on websites these days.

In figure 21 shows the implicit data we generated about the 10 employees. This data only shows what courses each employee has taken stored in the “courses” list, containing just the name of the course (e.g. “Safety A”). Based on this list, we can calculate similarities between users, using binary values. Value 1 represents if they have taken a course, and value 0 represents that they have not.

```

"Sebastian": {
  "courses": [
    "Safety A",
    "Safety B",
    "Course A",
    "Course B",
    "Course C",
    "Course D",
    "Course E",
    "Course F",
  ]
},
"Jorgen": {
  "courses": [
    "Course G",
    "Course H",
    "Course I",
    "Course J",
    "Course K",
    "Course O",
  ]
},
"Edvard": {
  "courses": [
    "Course L",
    "Course M",
    "Course N",
    "Course C",
    "Course O",
    "Course P",
    "Course J",
  ]
},
"Lars": {
  "courses": [
    "Course F",
    "Course Q",
    "Course R",
    "Course B",
    "Course K",
  ]
},
"Roar": {
  "courses": [
    "Safety B",
    "Course C",
    "Course P",
    "Course A",
    "Course Q",
  ]
},
"Hans": {
  "courses": [
    "Course F",
    "Course K",
    "Course R",
    "Course L",
    "Course C",
    "Course N",
  ]
}

```

Figure 21: Implicit mock data

Implicit data can compensate some of the issues with explicit data as individual tastes can be

understood through implicit behaviour. Click data is a common metric to track behaviour and build a user profile of tastes and interests. Though it is not a reliable indication of interest because misclicks happen, or the user can have been lured or forced to click on something. Furthermore, bots doing nefarious acts can pollute the data. Purchase history or consumption data is a better indication of interest as it is more resistant to fraud. Consumption data requires the consumption of time, thus making it a reliable indicator of interest compared to click data.

In many cases can explicit data be useful; if you can convince your users to give them to you. Otherwise, implicit data gives you a lot more to use — in the case of purchase data — it might be better than explicit data.

Choosing the right sources of data, that is available, is an essential early step when designing a recommender system. Unless the recommender system has large amounts of useful data to use, it can not produce good results. We used the implicit and explicit data illustrated in figure 21 and 20 to test the first prototype described in chapter 6 (initial approach).

5.2 Company A data access

After completing the first prototype (described in the initial approach, chapter 6), we received access to Company A's intranet. We could, therefore, see how they structure, categorise and label each course that their employees could conduct. Access was granted before we started the second implementation (described in implementation chapter 7).

Company A's intranet is mainly used by the administration and boutique managers to see numbers on which employees have conducted what courses. It also shows every course, each employee has conducted, and what store they belong to or work in. Each store displays which employees have taken their mandatory courses, and therefore if they have fulfilled their requirements or not. Reports can be generated from each store or on each employee.

The intranet display information about 21 505 employees located in 1666 different boutiques and stores. It shows an overall amount of all the employees' completion degree of all the courses, and this can be filtered on stores or employees.

Company A's intranet is also a course catalogue of all the courses their employees can take. As well as a list of all the courses, each course has an information page specific to its own, that is clickable to enter. This page explains what the course is about, and the purpose of taking the

course.

Courses can be filtered on different categories, these are:

1. **E-kurs** - is all the digital courses Company A offer. There are 208 courses in this category
2. **Course category** - is what courses are divided into or labeled as. There are 25 different categories
3. **Ekvivalenter** - are special courses that have sub courses that needs to be conducted well. There are 12 courses in this category

5.3 Generating mock data

Based on the information we got from roaming and exploring the intranet that Company A has, we could generate mock data that the implementation prototype could be tested on. This mock data is shown in figure 22.

1	userID	roleID	courseID	rating	relevant	learned
2	1	1	12	4.5	8	1
3	1	1	14	3.0	7	1
4	1	1	15	2.0	3	1
5	1	1	18	1.5	4	0
6	2	2	2	2.5	6	1
7	2	2	11	3.5	7	1
8	2	2	12	4.0	9	1
9	3	1	9	4.5	9	1
10	3	1	27	3.0	5	1
11	3	1	28	2.5	2	0
12	3	1	35	1.5	1	0
13	4	3	2	4.0	7	1
14	4	3	5	3.5	6	1
15	4	3	27	2.0	4	0
16	4	3	28	2.5	3	1
17	4	3	32	4.0	8	1

Figure 22: Excel view of the mock data generated in a *csv* file

Figure 22 shows a *csv* file that we randomly generated. The first line (top column) in the *csv* file shows the data labels. These are: *userID*, *roleID*, *courseID*, *rating*, *relevant* and *learned*.

These data labels are explained in table 3. The *roleID* data label can be mapped to a role name, for example is *roleID 2* a *butikkmedarbeider*. Label *courseID* is the same as *roleID*, it can be mapped to a course name. For example is *courseID 15* *Course A*.

Data label	Description
<i>userID</i>	An id unique for each employee
<i>roleID</i>	An id unique for each role the employees have
<i>courseID</i>	An id unique for each course an employee can conduct
<i>rating</i>	A number between 1 and 5 representing how much an employee liked a course
<i>relevant</i>	An integer between 1 and 10 representing how relevant an employee think a course is
<i>learned</i>	An integer of 0 or 1 representing whether the employee learned something (1) or not (0)

Table 3: Explanations of data labels from mock data in csv file

Why we use a *csv* file to test this, and how this works is further described in chapter 7, where we used the generated mock data illustrated in figure 22.

5.4 Foundation

Before we could start developing an SPL with machine learning components, we had to develop a product that were satisfying only the requirements of one customer. Since we had good documentation of these from Company A it was natural to start with them. The list of functional requirements (user stories) can be found in appendix A.1.

When developing a product, it is important with a structured architecture that is easy to understand and well planned. It should also be easily modified and with reusable components. The following subchapter is a brief introduction to some concepts of architecture in an SPL, and some approaches we evaluate and analyse.

5.4.1 Architecture

A common architecture is important for a set of products that are part of an SPL, hence sharing large components or parts of the architecture. It is essential that the architecture captures the

commonalities that specify requirements for the shared components; the differences need to be dealt with in an effective manner. The shared architecture in a product line is called *reference architecture* (van der Linden, Schmid and Rommes, 2007). The reference architecture is a generalised architecture that is composed of elements supporting the range of products in the SPL, and it also needs to have support for the variabilities each product have (as described in chapter 3.1.4). It is a common asset that is meant to be extended and gives easier access to shared assets for the product line.

Architecturally significant requirements (hereby referred to as ASR) are requirements that have a profound and essential impact on the architecture (Chen, Ali Babar and Nuseibeh, 2013). ASRs will define the reference architecture in many ways, so it is important to map out these early. It might be that some product-specific requirements are in conflict with each other, the reference architecture should, therefore, support variabilities.

We are mainly focusing on functional requirements in this thesis. In chapter 5.5, we go more in-depth on how we handle the functional requirements we are working with. Quality requirements such as performance, security and safety are not relevant to consider, because they are not relevant to the research questions, nor for the SPL Snapper is developing.

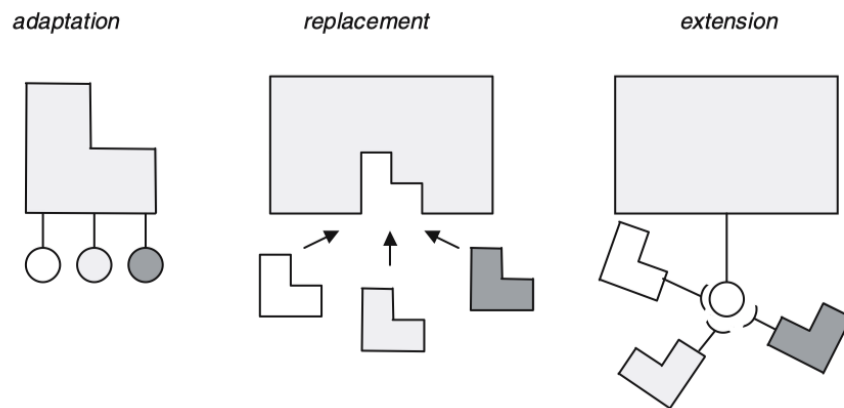


Figure 23: Three basic techniques for realising variability in an architecture. Figure taken from van der Linden, Schmid and Rommes (2007), pp. 41.

Variability is one of the most important architectural drivers in a software product line architecture (Tan, Lin and Ye, 2012). There are three different techniques to realise variation in an architectural perspective: (1) *adaptation*, (2) *replacement* and (3) *extension*. Figure 23 illustrates

the difference between these three techniques.

1. **Adaption** is when there is only one implementation available for a specific component. This component offers an interface that allows it to change or adjust its behaviour. An interface can offer these adjustments through a configuration file, parameters or variables, or source code patches. Other options do also exist.
2. **Replacement** technique allows for several different implementations of a component. Each implementation must follow specific guidelines from the architecture. A product-specific component must be chosen or developed each time for each product.
3. **Extension** technique is reliant on the architecture to provide interfaces that allow adding new components to it. These can be product-specific but does not have to be. The difference from replacement is that now components can be added through generic interfaces, allowing a various amount of components. In contrast, replacement technique uses interfaces that specify only certain components to be added.

Considering these techniques for the first prototype (initial approach), the adaptation technique seems most suiting for our prototype as it allows for more reuse than the other techniques (relevant to our research questions). Through this, we can create a parent-child relationship between the generic recommender engine and the product-specific recommender engines. The child components can, through an interface, change the behaviour of the generic component by passing variables. There are several implementation mechanisms to realise the adoption technique. One is *inheritance* which states that a subclass (the child component) can change some default behaviours of a class as needed. This is what we want to do, maintain a high degree of reusability with the machine learning components.

5.5 Requirements elicitation in SPL

Requirements elicitation is the process of researching and discovering the requirements of a system or a product (Masters, 2010). This is done from the user, customer and stakeholders perspective. The process is also known as requirements gathering or management. We did requirements elicitation after we had received the product requirements form the different customers of Snapper. This was done as part of the *suggestion to problem* step in the DSR process model (as described in chapter 4).

In appendix A.1 we have stored the requirements for the product for Company A, and in appendix A.2 is the planned product requirements for Company B found. These have some similarities (*commonalities*) and some variabilities that we have illustrated in figure 24. Figure 24 illustrates Company A's requirements as Req_1 . The part of the requirements from Company B that are unique for their product are drawn out to a single recommender engine (R_1). Req_2 illustrates the requirements for Company B's product. The unique ones are also stored in recommender engine (R_2).

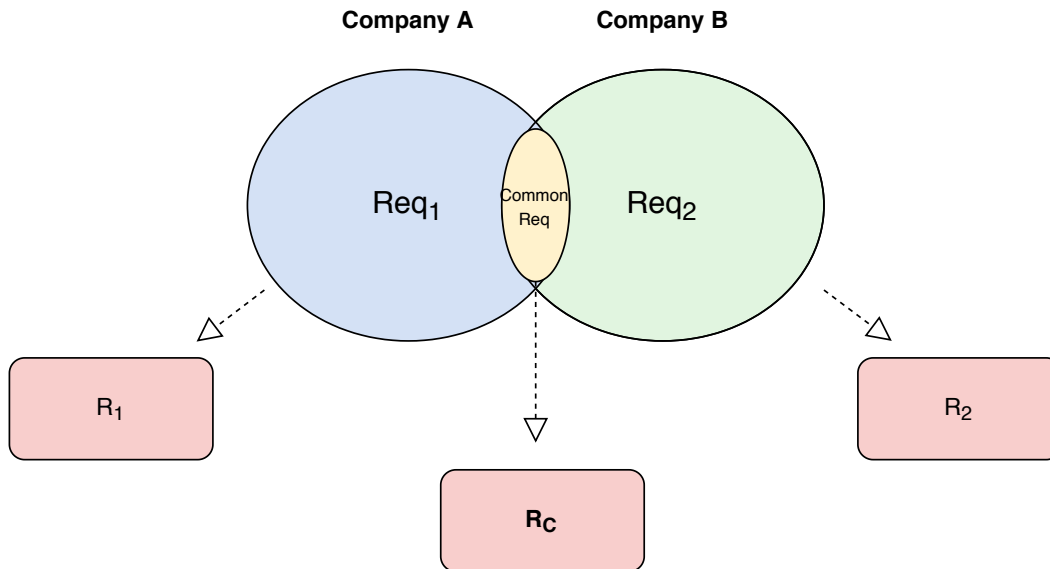


Figure 24: Product requirements for Company A and Company B

In between Req_1 and Req_2 can the *Common Req* be found. These specify the commonalities in both Company A and Company B's requirements. The commonalities are drawn out to a common recommender engine that is called R_C . This is the idea of creating a SPL for both the customers, to support several product requirements at the same time. R_1 , R_2 and R_C represents the machine learning components we are developing.

The *Common Req* section is illustrated relatively small, but this is just an illustration to visualise what it can look like. Commonalities in requirements specifications from several products can vary to a high degree and can have a lot of overlap. Considering only two products, commonalities are probably more usual to see, but when we start analysing the requirements of a third product, it can decrease a substantial amount. This is an aspect that is interesting to

observe, and we consider it throughout the research process.

6 Initial approach

In this chapter, we describe the initial approach we had to solve the case. We start by presenting the architecture of the first prototype we made. Then we present different distance metrics for calculating user similarities. We explain how this prototype recommends items and what machine learning algorithm this prototype uses. Then we present the next version of the prototype, that allows for the implementation of SPL. And finally we present data we measured while developing the first prototypes, the process of going from a simple architecture to an architecture that supports SPL.

6.1 First prototype architecture

Before we could make a prototype with a reference architecture, we started making a prototype satisfying just the requirements of one customer, for Company A. The architecture of the first prototype is illustrated in figure 25. It contains a few amounts of components, but that was the goal throughout the development process, to make it simple. The prototype is written in the JavaScript library React.js (on version 16.10.2).

The prototype uses a recommender system approach called collaborative filtering (explained in chapter 3.3). This is based on finding similar users, to the given user, and then recommend their preferences or like items. The machine learning algorithm it uses is based on k-nearest neighbours, but this is further explained in chapter 6.1.1.

At this stage, we do not consider implementing any architectural patterns because the prototype doesn't have to satisfy the product requirements. It would also make the architectural layout more complex, and the entire development would take more time and be more cumbersome.

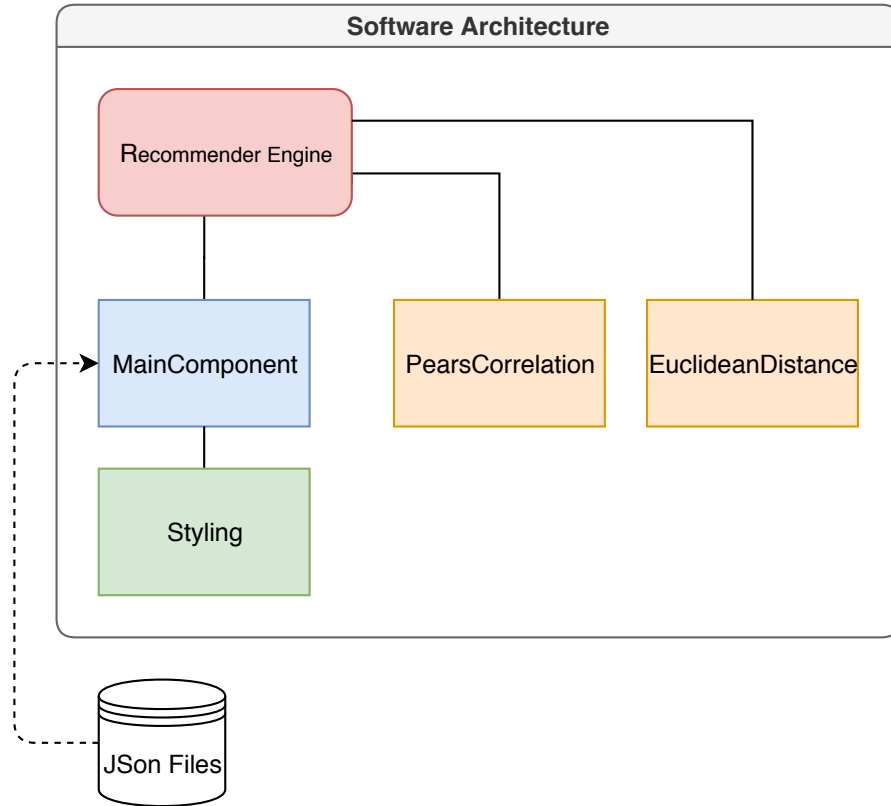


Figure 25: Architecture of first prototype

The first prototype has a software model with a *MainComponent* that retrieves data from *JSoN files*. It also has a separate *Styling* component. The *MainComponent* loads and reads the JSON files and then stores them in dictionaries. After this step, the *MainComponent* passes the dictionaries to a *Recommender Engine*. When the dictionaries are passed to the recommender engine two variables are also passed: *user* and *type*. The *user* variable specifies which user that the items should be recommended to; hence the engine will calculate the similarity of this user, to every other user. The *type* variable specifies which calculation formula that should be used. There are three types of parameters that the engine allows, these are illustrated in table 4.

Variable: <i>type</i>	Calculation formula
<i>euc</i>	Euclidean distance
<i>pears</i>	Pearson correlation
<i>avg</i>	Average between <i>euc</i> & <i>pears</i>

Table 4: Three parameters for *type* variable

Our recommender engine uses two different sub-programs. These are two different types of formulas to calculate how similar two objects are, and returning a score on this. The first formula is called Euclidean distance, and the second is called Pearson correlation. We explain more in-depth on how these calculations are made and how the rankings are done based on these score in chapter 6.2. We also explain why the *type* variable allows for *avg* input, and why this can be smart.

When these variables have been sent to the recommender engine, the two subprograms (*PearsCorrelation* and *EuclideanDistance*) calculates all the similarity scores to the other users. After all, users are ranked based on these scores, and a list is created with the user rankings. Courses are recommended based on this list to the given user that was passed as input. The ranking of users and recommendation of items are also further explained in chapter 6.2.

The next step is to create a generic recommender engine that can do ranking and recommendations on any types of input data or objects, specified by each instance of the recommender engine. We also need a reference architecture that the generic recommender engine can be a part of, so that each product instance can adapt to with an inheritance relation to the recommender engine.

6.1.1 A tweak of K-Nearest Neighbours

As described in the background chapter (in chapter 3.2.3) about k-nearest neighbours (KNN), is that it is an instance-based algorithm (also called memory-based learning). This means that it compares new instances of data, to already calculated data stored in a database.

We do not have a database with stored training data that we can compare new data instances or data points to. We, therefore, can't use it as a memory-based approach. Instead, we calculate every user score, relative to the given user, and each time recommendations should happen. This

makes it a tweak of the KNN, because as figure 27 shows, we use the recommendations from the closest k neighbours of a user. We place all users in an n-dimension space, and calculates, using distance metrics, to find who is the closes.

Our recommender system tweak of KNN works like this:

1. Load the data
2. Initialize threshold to your chosen limit of user similarity score (selecting k closest users)
3. For each user in the data:
 - (a) Calculate the similarity score (distance) between the given user and the current user in the data
 - (b) Add the similarity score (distance) and the user ID from data to a list
4. Sort the list of similarity scores (distances) from largest to smallest (in descending order, the higher similarity score the lower distance)
5. Return ranked list of users

Based on the ranked list of users, we can make recommendations. From this list that is returned, we can get the K first entries of users, and get their items. How recommendations are made based on this list is further explained in chapter 6.2.3.

When we have real data from Company A or Company B that the prototype can use, the optimal solution would be to use some of it as training data. This is pre-calculated user scores stored in a database. Each time new data points from new users need recommendations, we calculate a user score for each new instance and then compare it to the database.

6.2 Distance metrics

We have been using two different formulas for calculating the similarities between users: *Euclidean distance* and *Pearson correlation*. These two formulas are distance metrics meaning that they define and calculate a distance between a pair of elements placed in a two-dimensional graph (Sharma, 2019). In the following sub-chapters (chapter 6.2.1 and 6.2.2) we will explain both these formulas.

6.2.1 Euclidean distance

Euclidean distance is a calculation formula to measure the distance between several points in Euclidean space (Robinson, 2017). Euclidean space is a two- or three-dimensional space, and it can even be a finite number of dimensions, where points are placed by coordinates (one coordinate for each dimension) to calculate the distance between the points by a distance formula (Hosch, 2011).

To calculate the Euclidean distance between two points in one dimension is just the absolute value of the difference between the coordinates. The calculation then becomes $|p_1 - q_1|$, where p_1 is the first coordinate to the first point and q_1 is the first coordinate to the second point. Because distance is almost always considered to have a non-negative value, the absolute value is used.

If we have two points: p with coordinates (p_1, p_2) and q with coordinates (q_1, q_2) in a two dimensional space. Constructing a segment between these two points will form the hypotenuse in a right triangle (Robinson, 2017). The length of the legs of the triangle are given by $|p_1 - q_1|$, and $|p_2 - q_2|$. Using the Pythagorean theorem, the length of the hypotenuse is, therefore, the squared sum of the two legs. The formula is illustrated in equation 5. The distance between two points, therefore, becomes the length of the hypotenuse.

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2} \quad (5)$$

Euclidean distance between two points in a two dimensional space

When writing JavaScript to calculate the Euclidean distance between two points: p with coordinates $(1, 4.5)$ and q with coordinates $(4, 3)$, the following code snippet (figure 26) shows how the code syntax will look. The variable **euclid** is the hypotenuse length between the points, and is, therefore, the Euclidean distance between the points.

```
1 var euclid = Math.sqrt(Math.pow(1 - 4, 2) + Math.pow(4.5 - 3, 2));
```

Figure 26: Euclidean distance (JavaScript) example code

Figure 27 shows a diagram of 6 employees and their different course ratings on the courses *Course A* and *Course B*. This is how we can structure user ratings in a two-dimensional space

to calculate how similar the users are. When placing the points in a diagram, it gives a good indication of which users that are quite similar, because their points will be fairly close to each other. For example we can easily see that $User_1$ is more similar to $User_2$ than to $User_6$ because of their closeness in placing.

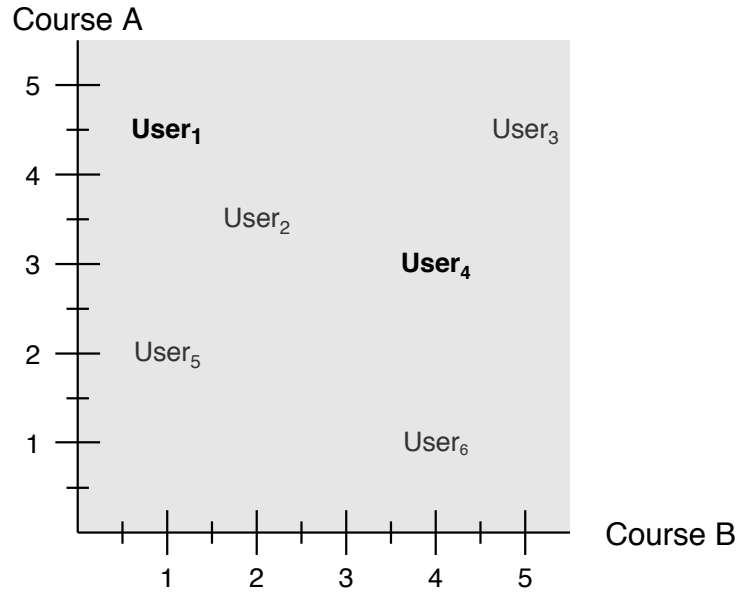


Figure 27: User-feedback diagram

When you are calculating the Euclidean distance in several dimensions between two points, the list of coordinates for one point can be very long. For example for the point p the coordinates will be (p_1, p_2, \dots, p_n) where p_n is the n th dimension. When we are calculating the difference between two employees, each dimension represents a course, and because there are a lot of courses, we will operate with a lot of dimensions. The formula for calculating Euclidean distance in n dimension is shown in equation 6. This is the formula we are using to calculate Euclidean distance between two employees.

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^n (p_i - q_i)^2} \quad (6)$$

Euclidean distance between two points in a n dimensional space

Effect of using implicit or explicit data

The difference between using implicit or explicit data in the Euclidean distance formula is crucial. When using explicit data, it is simple to calculate how similar two users are, resulting in an understandable and logic score, that cohere with the two data points.

However, when using implicit data, the Euclidean distance formula does not work. That is because the formula only compares the same coordinates, and ignore coordinates that only one of the data points have. This is similar to one employee that has taken a course the second employee has not. When the formula receives implicit data, it uses the value 1 if an employee has taken a course, and 0 if he or she has not. When the formula ignores the 0 values, the formula will always give the output score of 1 if they have some similar courses, and 0 if they have none.

This is one reason why we take the average between the two similarity scores, calculated by Euclidean distance and Pearson correlation, but we will go further into this in chapter 6.2.3.

6.2.2 Pearson correlation coefficient

Pearson correlation coefficients is a formula to measure the linear correlation between two variables X and Y. There are several different types of the correlation coefficient formula, Pearson correlation is one of the most commonly used, especially in statistical analysis (Correlation Coefficient: Simple Definition, Formula, Easy Calculation Steps, 2020).

The coefficient is the *covariance* of the two variables X and Y, divided by the product of their *standard deviation* (Pearson correlation coefficient, 2020). Covariance in probability and statistics is a measure of the joint variability of two variables. If the covariance is positive, the variables vary together in the same direction, but if the covariance is negative, they vary inversely (Weisstein, 2020). Standard deviation in statistics is a measure of the amount of variation in a set of values. The Pearson correlation formula is shown in equation 7.

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{(\sum X^2 - \frac{(\sum X)^2}{N})(\sum Y^2 - \frac{(\sum Y)^2}{N})}} \quad (7)$$

Pearson correlation coefficient formula

The following code (figure 28) is a code snippet from the *Pearson correlation* function. It is written in JavaScript and can be found in the *PearsonCorrelation* component in our prototype.

```

19 var num_existence = Object.keys(existp1p2).length;
20
21 for (item in existp1p2) {
22     X += dict[p1][item];
23     Y += dict[p2][item];
24     X2 += Math.pow(dict[p1][item], 2);
25     Y2 += Math.pow(dict[p2][item], 2);
26     XY += dict[p1][item] * dict[p2][item];
27 }
28
29 var numerator = XY - (X * Y) / num_existence;
30
31 var st1 = X2 - Math.pow(X, 2) / num_existence;
32 var st2 = Y2 - Math.pow(Y, 2) / num_existence;
33
34 var denominator = Math.sqrt(st1 * st2);
35
36 var score = numerator / denominator;

```

Figure 28: Pearson correlation (JavaScript) code

The explanations of the variables from the code snippet (shown in figure 28) are respectively:

- **existp1p2** is the merged dictionary with all the similar items user 1 (**p1**) and 2 (**p2**) has
- **X** is the sum of all the item scores from user 1 (**p1**)
- **Y** is the sum of all the item scores from user 2 (**p2**)
- **X2** is all the item scores by the power of 2 summed up from user 1 (**p1**)
- **Y2** is all the course item by the power of 2 summed up from user 2 (**p2**)
- **XY** is the product between the item scores by user 1 and 2 on the same course
- **dict[.][.]** variable is the dictionary with all the data fields on users and items
- **numerator** variable is all the calculations over the fraction bar, hence it is called numerator

- **denominator** variable is all the calculations below the fraction bar, hence it is called denominator
- **num_existence** is the number of items in the dictionary (**dict**) (**N** from the formula 7)
- **score** is the final Pearson correlation score the function return as output (**r** from the formula 7)

Pearson correlation between data fields in a data set is a measure of how well they are related, meaning how well do the two data fields fit into a straight line, being linear (Correlation Coefficient: Simple Definition, Formula, Easy Calculation Steps, 2020). The Pearson correlation formula returns a value between 1 and -1, where:

- **1** indicates a strong positive relationship between the variables. This means that the two users have the same rating for all the same items.
- **0** indicates that there is some relation in the variables. The two users have some similarities in their item rating.
- **-1** indicates a strong negative relationship between the variables. This means that the two users are not similar.

From the code snippet (in figure 28) the **score** variable is the return value from the function. This indicates how similar two users are, with the value ranging from 1 to -1. The Pearson correlation function calculates similarity score based on input with two users specifications, and a dictionary of their items (*dict*[.][.] variable), and gives an output, that is the score.

Effect of using implicit or explicit data

There is a slight difference when Pearson correlation formula calculates similarity on explicit or implicit data. When using explicit data, it is much easier to calculate (more exact) how similar two users are, and how their different ratings on the same items make the coefficient, for example, deviate less, resulting in a higher similarity score.

When using implicit data, the score will vary less from user to user because their rating of the item can not be used. What will affect the score in any direction (towards 1 or -1) is whether one user has a lot more items than the other user, that he or she has not. Therefore it is preferable to use explicit data when calculating similarity score, same as Euclidean distance metric.

6.2.3 Recommending items

In table 5 we have used some of the explicit mock data from figure 20 to show how items are recommended to a user (e.g. Hans) based on user similarity scores. In this table, the similarity scores are only calculated based on the Euclidean distance formula, but this is just an example of how it is done. Using Pearson correlation or an average between both formulas does not change the way items are recommended.

Employee	Similarity	Course A	S.xCourse A	Course R	S.xCourse R	Course C	S.xCourse C
Sebastian	0.99	3.0	2.97	2.5	2.48	3.0	2.97
Jorgen	0.38	3.0	1.14	3.0	1.14	1.5	0.57
Edvard	0.89	4.5	4.02			3.0	2.68
Mats	0.92	3.0	2.77	3.0	2.77	2.0	1.85
Lars	0.66	3.0	1.99	3.0	1.99		
Total			12.89		8.38		8.07
Sim.Sum			3.84		2.95		3.18
Total/Sim.Sum			3.35		2.83		2.53

Table 5: Similarity ranking among users on mock data.

The explanations of the different terms in table 5 are:

- **Similarity** is the score Euclidean distance formula calculates. This is the result from calculating similarities between other users and Hans
- **S.x...** is the similarity score (one similarity score on a user) times the score rating the same user has given on a course (e.g. Course A). **S.x...** can be found in the top column, for example **S.xCourse A**
- **Total** is all the **S.x** scores summed up
- **Sim.Sum** is the sum of all the **Similarity** scores from other users that have reviewed the given course
- **Total/Sim.Sum** is **Total** divided by **Sim.Sum**

Similarity scores are either calculated by the Euclidean distance formula, Pearson correlation formula or an average between them. After these are calculated, we can see that Sebastian is

the most similar user to Hans. However, recommending items to Hans is not only based on the fact that Sebastian is a very similar user. It is therefore crucial with the $S.x$ variable that weights the different scores on the courses against the similarity score. All the weighted scores (e.g. $S.xCourse A$) are summed up and divided by the sum of similarity scores ($Sim.Sum$) from only the users that have reviewed this course.

We end up with $Total/Sim.Sum$ and this is the ranking of the items (in this case courses). We can see that *Course A* has the highest-ranking score, followed by *Course R*, and *Course C* has the lowest score. It is, therefore, most likely that Course A is the most relevant course for Hans. This is based on that a lot of the most similar employees to Hans, has given a high review on this course. The recommender engine will, therefore, recommend courses in the following order: (1) Course A, (2) Course R, (3) Course C.

Average between Pearson correlation and Euclidean distance

The reason that we use the average between the Euclidean distance score and the Pearson correlation score is that they calculate similarities between users in different ways. Pearson correlation is a measure of how well two sets of data fit into a straight line. This formula is a bit more complicated than Euclidean distance, but it gives a better result when the data is not normalised (Aggarwal et al. 2001). The Euclidean distance metric is a measure of how close two data points are in a dimension space.

Euclidean distance uses only similar coordinates from the two data points. This means that when calculating similarities between two users, the Euclidean distance metric ignores the courses that just one of the users have taken, and uses only courses that both have completed. This is a problem when we want to recommend courses to a user that have completed a few amounts of courses, and there are not a lot of other users to compare to and recommend courses from.

Pearson correlation uses the sum of all the ratings of courses from two users and compares how much these sums deviate from each other. In the numerator in Pearson correlation, a variable ensures that if two users have taken a lot of the same courses they similarity score is higher (XY variable form code snippet from figure 28). This variable does not have to be there if they have not taken any of the same courses; the similarity is just decreased. Therefore Pearson correlation can calculate similarities between all users if they have taken the same courses or not.

Because Euclidean distance can not find similarities between two users with no similar courses, and Pearson correlation can, we, therefore, use both formulas when calculating similarity score, and use the average between these. We came up with this idea after testing a lot and seeing that the average score provides a good indication of how similar two users are based on two different formulas. The recommendations both formulas provide does not deviate a lot, and when using the average, the recommender engine could provide recommendations in a lot more cases.

Calculating the average between the two different scores is conducted in the following way:

Pearson correlation score: -0.8

Euclidean distance score: 0.75

$$(-0.8 + 1)/2 = \underline{0.1}$$

$$(0.1 + 0.75)/2 = \underline{\underline{0.425}}$$

This is just an example, and it is improbable that the two formulas will provide scores with such a high deviation. The first calculation with result 0.1 is to standardise Pearson correlation according to Euclidean metric because Pearson correlation results range from -1 to 1, and Euclidean distance range from 0 to 1. The average between the two scores is 0.425.

6.2.4 Evaluating other distance metrics

A lot of machine learning algorithms use different types of distance metrics, both supervised and unsupervised algorithms. ML algorithms use these distance metrics to improve their performance of classification, clustering and informational retrieval process (Kundu, 2019). A distance metric can also help the algorithm to recognize similarities between data or the content. It is important to use the right distance metric because a distance metric plays an important role in the ML model. This is because the models rely on a distance calculation between two data points.

Minkowski distance is a distance metric to calculate the distance between two data points, as calculating the length of a vector going from one data point to the other (Gohrani, 2019). Minkowski distance is a generalized distance metric that other distance metrics is based on. Distance metrics that are based on Minkowski distance are for example *Manhattan distance*, *Chebyshev distance* and *Euclidean distance*. These calculate distances between data points in the following way:

- **Manhattan distance** is used to calculate the distance between two data points that are following a grid path. The grid path contains straight lines and corners that are orthogonal. To calculate the distance between two points is, therefore, to neglect all corners, and calculate the length as it only has one corner (Gohrani, 2019). The calculation, therefore, becomes the length of two lines and the sum of these.

Calculating distances in a fixed but high amount of dimensions in space is optimal to use Manhattan distance for (Aggarwal et al., 2001). This is because the path is following a grid, and the corners are of a limited number. However, we do not know the number of dimensions, and it might be changing from product to product in the SPL. Using Manhattan distance is, therefore, not optimal.

- **Chebyshev distance** (also called Maximum value distance) calculates the distance between a set of objects in any direction in a dimension space (Teknomo, 2015). The dimension space can have several dimensions, other than three space dimensions. The path between two points can be in any direction unlike Manhattan distance, so the distance calculation is, therefore, the maximum length of a line between two data points.

To calculate distances between two points, all coordinates are needed, and only the same coordinates are being compared in the formula. Say, for example, two users have only taken a single course, but the courses are different ones, then Chebyshev distance formula is not possible to use between the two data points. This is an issue that makes Chebyshev less applicable to solve our case and the reason we did not choose to use it.

- **Other distance metrics** such as Cosine similarity and Hamming distance does not apply so good to our case the way Euclidean distance does. *Cosine similarity* is used to calculate the similarity between two vectors using the angle between them. The vectors represent data fields with a given value. *Hamming distance* is used to calculate similarities between two binary data strings. The data input needs to be categorical to calculate distance. We did not evaluate these distance metrics a lot because they are not that applicable to solving our case.

6.3 Prototype with SPL implementation

When developing a new prototype with support and structure for an SPL, that is based on the old prototype (from chapter 6.1), there where a lot of requirements that needed to be considered. As described in chapter 5.5, structuring out and handling the different product requirements

were necessary to understand what functionality that needed to stay generic and open to all products, and what needed to be drawn out to specific products.

After we had done this we started organising and cleaning up code, as well as commenting out all functions that were used, and analysing where all variables were passed. When we got a clear understanding of the data flow between components and specific functions, it was easier to divide components into smaller ones.

We started developing an instance of the first product in the SPL. This instance (R_I) is based on the product requirements from Company A (found in appendix A.1). We developed the generic recommender engine simultaneously with R_1 because the first prototype was based on Company A's product requirements and all the functionality for R_1 was located in the *Recommender Engine* component (from figure 25).

This (improved) prototype is also written in the JavaScript library called React.js. It also uses collaborative filtering for recommending items, and uses the same tweak of k-nearest neighbours as a machine learning algorithm.

6.3.1 Architecture

The architectural layout of the new prototype with SPL implementation has only a few changes from the first prototype. The biggest changes are a modification of code within some of the components, and that some code has been moved to other components.

The new prototype has a generic recommender engine component (R_C) that can be specified or modified into unique instances of recommender engines for a specific task. These instances form the product line, and are extended from the generic model. The generic model acts as a parent component to each child component (R_1, R_2, \dots, R_n).

The R_C component is new from the first prototype, as well as a *SimilarUsers* component that has been drawn out of the old recommender engine. This component communicates with R_C and the *PearsCorrelation* and *EuclideanDistance* components. These two components still do the similarity calculations, and have not had the biggest changes (further explained in chapter 6.3.2). The *SimilarUsers* component now does the ranking of users and passes them back to R_C as a list of scores. This is the component that finds the most similar users, using the KNN tweak algorithm (described in chapter 6.1.1).

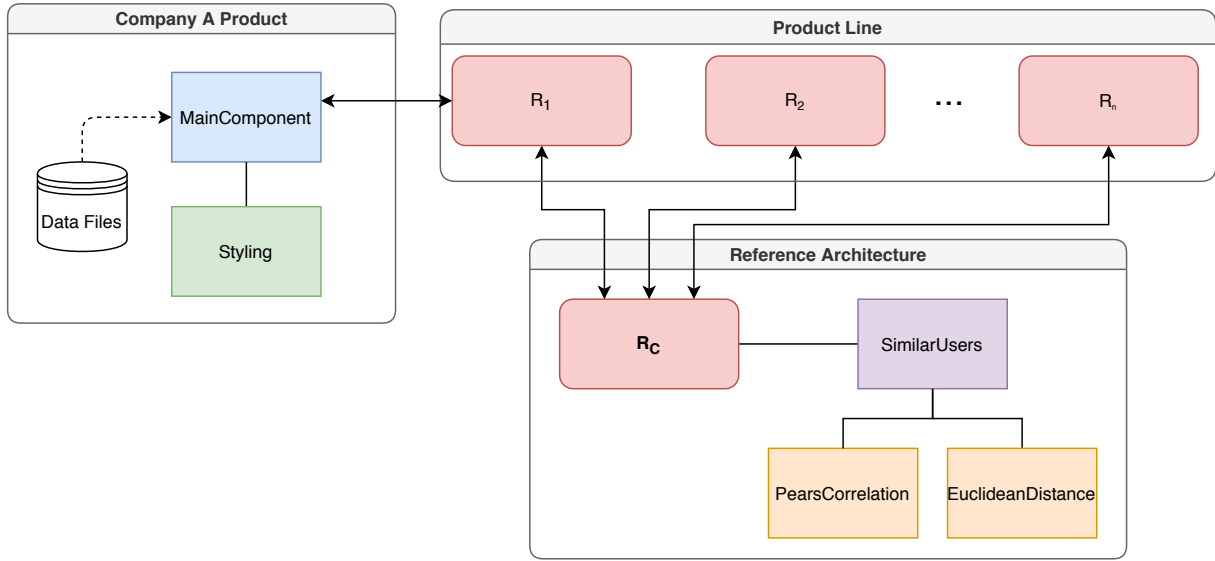


Figure 29: Architecture of SPL components

The R_C component receives back the list of ranked users from SimilarUsers and passes them to the child component recommender, in our case R_1 component. R_C component controls the data flow with SimilarUsers, and sends only the list of ranked users back to each subcomponent.

All the components in the product line (R_1, R_2, \dots, R_n) follows the adaption technique illustrated in figure 23, and there is only one implementation of R_C that all these components use. Each instance of R_C inherit the functions from this component, and has as parent-child relation (implementation technique from section 5.4.1).

Through a set of parameters, the child components can change how some functions work in R_C , that further sends data to SimilarUsers component for user ranking (illustrated in figure 32). Each product instance (e.g. R_1) specifies the following parameters:

- **dataFormat** - whether the data is explicit or implicit when calculating similarity scores
- **type** - meaning whether EuclideanDistance or PearsonCorrelation component should do the similarity calculations, or an average between these two values should be used
- **person** - specifies which user they distance metrics should calculate other user similarities against

- **threshold** - a variable that specifies where the threshold of how similar two users should be. For example can this be 0.4, meaning that all the users with lower similarity score than 0.4 should be discarded
- **dict** - is the dictionary of users and items passed from the sub component

The R_C components use these parameters when communicating with the SimilarUsers component. It passes the *dataFormat*, *Type* and the specific user that was sent from the sub-component, as well as a second user so that through iterations, all other users are compared to the specific one. These parameters are passed, as well as an extract of the dictionary of users. When the R_C component receives back the ranked list of users, it uses the *threshold* variable to trim it, before it passes the ranked list of users back to the subcomponent.

The sub-component calculates the recommendations for the given user based on the rank list on user similarity scores from R_C , and creates a ranked list of items with a relevance score for each, in the order, they should be recommended.

6.3.2 Generic code

To make a generic recommender engine, it is important that the code is as generic as possible. By generic, we mean that it can be used with multiple different types of data sets and therefore by different products in the SPL. In order to make the code as generic as possible, the code must be written in terms of *to-be-specified-later* (Generic programming, 2020). This means that when instantiated, each product defines a set of parameters to fit their instance, so it can, for example, load their data set with a specific amount of data fields. This can vary from product to product.

```

1  export default function euclidean(p1, p2, dict) {
2      var existp1p2 = {};
3
4      for (var key in dict[p1]) {
5          if (key in dict[p2]) {
6              existp1p2[key] = 1;
7          }
8      }

```

Figure 30: *EuclideanDistance* components module constructor

In order to make the *EuclideanDistance* and *PearsonCorrelation* components work for any data set, the two calculation functions (distance metric) within these components should only use a minimum of specified parameters when doing calculations.

Figure 30 illustrates the constructor of the *euclidean* function in the *EuclideanDistance* component. The *dict* variable is a data set needed to calculate similarity, and *p1* and *p2* are two users specified by just an ID or a unique name. Both these exist in the dictionary. The *for*-loop iterates over the dictionary and finds the similar items between the two, and stores them in a new dictionary: *existp1p2*. This dictionary is used for the similarity calculations.

```
1 export default function pearsonCorrelation(p1, p2, dict) {
2   var existp1p2 = {};
3
4   for (var item in dict[p1]) {
5     if (item in dict[p2]) {
6       existp1p2[item] = 1;
7     }
8   }
}
```

Figure 31: *PearsonCorrelation* components module constructor

Figure 31 shows the constructor of *pearsonCorrelation* function within the *PearsonCorrelation* component. It does exactly the same as the *euclidean* constructor. Because *PearsonCorrelation* and *EuclideanDistance* components use so few parameters, they can work on any type of data set. All that is required is a data set (*dict*) and two user names (*p1* and *p2*) to calculate how similar they are.

```
1 import similarUsers from "./similarUsers";
2
3 export default function genericRecommender(dataFormat, type, person, threshold, dict) {
```

Figure 32: Generic recommender (R_C component) modules constructor input

Figure 32 shows the *genericRecommender* (R_C) component, and its constructor function. It only takes in the minimal amount of parameters, and only imports the components that it uses and sends data to. In this case it is only the *similarUsers* component. When doing so, we keep the code clean and ensures that data only flows to the components we specify it to do.

6.3.3 Layout of components and views

The layout of the different views of the prototype is illustrated in figure 33 and 34. As mentioned earlier, this prototype is made in the JavaScript library React.js, and it is therefore hosted and tested locally (on `http://localhost:3000/`). Figure 33 shows, under the *Recommender System* header a *Dataset* view. This is to show what data is in the JSON file. In this view, you can also specify what JSON file that should be used (both explicit or implicit data), and there is a *Reset* button to reset the prototype so that the cache and dictionary is cleared.

Under the *Dataset* view, there is a user similarity view. From two drop-down menus, you can specify two different users based on which data set you have chosen (changed dynamically). A *Euclidean Dist* score is shown, illustrating the Euclidean distance score, and a *Pearson Correlation* score is shown. This shows the similarity scores and how the two formulas can differ when calculating similarity. Two bars (blue for Euclidean distance, and yellow for Pearson correlation) shows a percentage of how similar the two users are. Figure 33 shows an example where similarity scores between *Jorgen* and *Hans* are calculated.

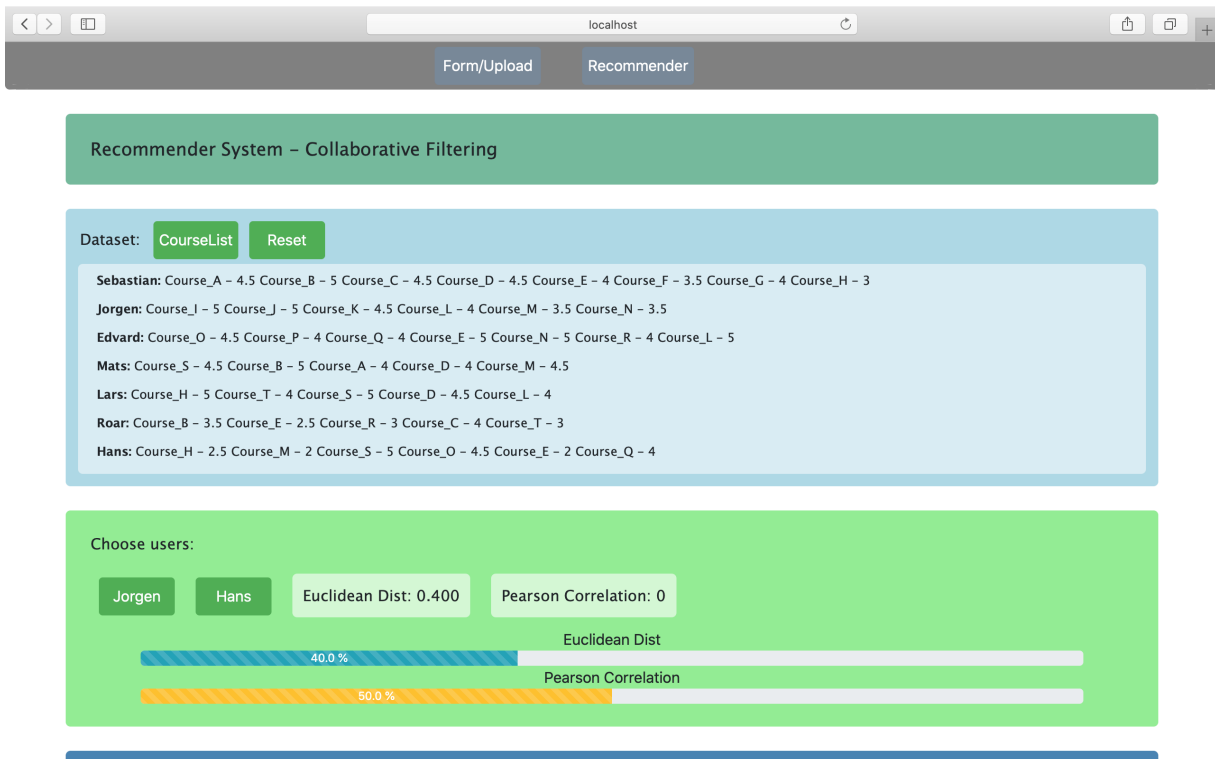


Figure 33: First part of SPL prototype

Underneath the similarity view, there is a recommendation view, shown in figure 34. This view also has a dynamic drop-down menu based on the data set, where you can choose a user. Based on your choice, three different columns show a *Recommendation list* based on three different calculations. As mentioned earlier, these are Pearson correlation, Euclidean distance and an average between them. Because a data set with explicit data are chosen, a predicted rating will appear next to the recommended item, in figure 34. This is based on the user *Jorgen's* conducted and rated courses. The recommendation list shows the top 10 recommended courses for him. Under the recommendation list, there is a *Similar users* list, showing the other users ranked based on similarity.



Figure 34: Second part of SPL prototype, recommending courses

Figure 33 and 34 shows the implementation of a generic recommender engine, and a product instance (R_1) that satisfies the product requirements from Company A. It only shows explicit mock data, and not real Company A data.

6.4 Process from first prototype to SPL prototype

Throughout the first two phases of implementation, resulting in one prototype, we have collected some data about the process we went through. The data we collected was: *number of hours* spent, *number of components* made and finally *number of lines of code* written on each development phase. On the number of lines of code, all styling files are not considered, this includes, for example, all files ending with *.css*.

Table 6 shows the collected data after the first prototype was made. This is the prototype that only satisfies the product requirements from Company A (described in chapter 6.1). This prototype has no implementation or architecture for an SPL, that can share architecture or components.

First prototype	
Time spent (hours):	136
Number of modules/components:	8
Number of lines of code in <i>src</i> folder	
src/components	489
src/utility	154
src/data	534
src/	163
Total:	1340

Table 6: Collected data from implementation of first prototype

Table 7 shows collected data after further developing the prototype through a second development phase. This prototype supports SPL with reference architecture and generic components (described in chapter 6.3).

SPL prototype	
Time spent (hours):	114
Number of modules/components:	12
Number of lines of code in <i>src</i> folder	
src/components	524
src/utility	255
src/data	534
src/	163
Total:	1476

Table 7: Collected data from implementation of SPL prototype

Comparing table 6 and 7, we spent 136 hours on the first development phase, and 114 hours on the second development phase. That is 114 hours spent to make 4 extra components, and just 136 lines of code extra.

During these 114 hours, we made the R_C component and other supporting components (part of a reference architecture). We also made an instance of the generic recommender engine (R_1). The second development phase took 22 hours less than the first, but it resulted in the same prototype with the same views, satisfying the same product requirements. The only difference after the second development phase was that it had a reference architecture that was supporting an SPL.

6.5 Estimates of implementing R2

After the first prototype was made into a new prototype with a reference architecture, the *Recommender engine* component was drawn out to a R_C , a generic component and R_I , a specific recommender system component for Company A. The next step is to make a new instance of the R_C into a R_2 that satisfies the product requirements from Company B (found in appendix A.1).

Making a new instance of the product is not something that is requested from Snapper, and we, therefore, will only make predictions about what a new instance of the *Recommender engine* would look like based on the product requirements.

R₂ component for Company B	
Time spent (hours):	80
Number of modules/components:	14
Number of lines of code in <i>src</i> folder	
src/components	x
src/utility	x
src/data	x
src/	x
Total:	1700

Table 8: Predicted data from implementation of R₂ recommender system for Company B

Table 8 shows the predicted data when implementing R₂ for Company B. It shows only two new components (14, 12 earlier), because this includes an R₂ instance and a product-specific data loader. We predict that this will result in a total of 1700 lines of code (approx 230 lines extra), which is a fairly low amount of code for each new component. This is based on the similarities with Company A product requirements, and that the data sets are predicted to be somewhat similar. Because of this, we predict that the time it takes to implement will be reduced significantly from the previous implementation (114 hours) to approximately 80 hours.

6.6 Considerations for the future

The exciting aspect here is evaluating the hours spent to make a reference architecture that supports an SPL. Is it worth it to create this rather than stand-alone products, considering the time spent? If you compare the extra amount of components and lines of code, the answer may be no, because it looks like a very little productive time spent. However, during the second phase, we had to learn a lot about how you create optimal generic code and cleaning up and structuring the code.

The probability that product requirements from future customers are similar to Company A and Company B's are relatively small. We predict that creating a product instance for Company B (R₂) would not take that much time, because there is a lot of overlap in Company A's and Company B's product requirements and data sets on their employees. But when you add a third product (R₃), the requirements may change a lot, that also requires changes to the

generic recommender engine, or maybe even how similarity or ranking are calculated. Then the implementation phase would be time-consuming, and a lot of unnecessary work may appear, that did not if you developed a stand-alone product.

In the long-term, it might be cost-beneficial. Because for each new instance of the recommender system, more experience is gained, so it might require less effort and time to implement. It is normal to break even after 3 products (as figure 1 shows, from chapter 3.1.1). It is therefore relevant to evaluate how much you save for each new product instance in the long-term, and where you break even.

7 Implementation

In this chapter, we present the implementation of the final prototype we made. We start by presenting the overall architecture of the system Snapper has developed, which the prototype is constructed for, then we present the architecture of the prototype we have developed, and finally, we present how the prototype is implemented in an SPL.

Machine learning applications are becoming popular in the industry; however, the process for developing, deploying and continuously improving them is more complex compared to more traditional software, such as a web service or a mobile application. They are subject to change in three axes: the data, the model and the code itself. Their behaviour is often complex and hard to predict, test, improve and explain.

"*Hidden Technical Debt in Machine Learning Systems*" by Sculley et al. (2015), highlight that only a small fraction of real-world machine learning systems is comprised of actual machine learning code. There is a vast array of encompassing infrastructure and processes to support their evolution. Furthermore, Sculley et al. discuss the many sources of technical debt that can accumulate in such systems, some of which is related to data dependencies, reproducibility, model complexity, testing and changes in the external world.

Many of the concerns are also present in traditional software systems, and Software Product Line Engineering has been the approach to develop software applications (software-intensive systems) using *software platforms* and *mass customisation*.

Additionally, to the code, changes to ML models and the data used to train them are another type of change that needs to be managed and adapted into the software product line, as shown in figure 35.

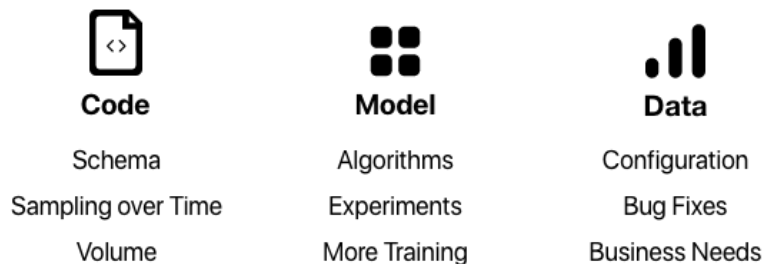


Figure 35: The three axes of change in an ML application: data, model and code, and reasons for them to change

With this in mind, we shall look into how we have implemented a recommender system into a software product line.

7.1 Overall architecture

The architecture of the core *Product A* product has two types of clients (1) a mobile application, and (2) a web application. The mobile client tier is written in React Native, and has shared single codebase across both iOS and Android. Moreover, the client tier for the web application is written in ReactJS for benefits of having to learn a similar API for both mobile and web development. The client tier is the what the user will interact with to access the features of the application.

Both types of display layers share a common API served by business logic tier. The business logic tier is represented by the application server, in which acts as a bridge of communication for the client tier and the database tier.

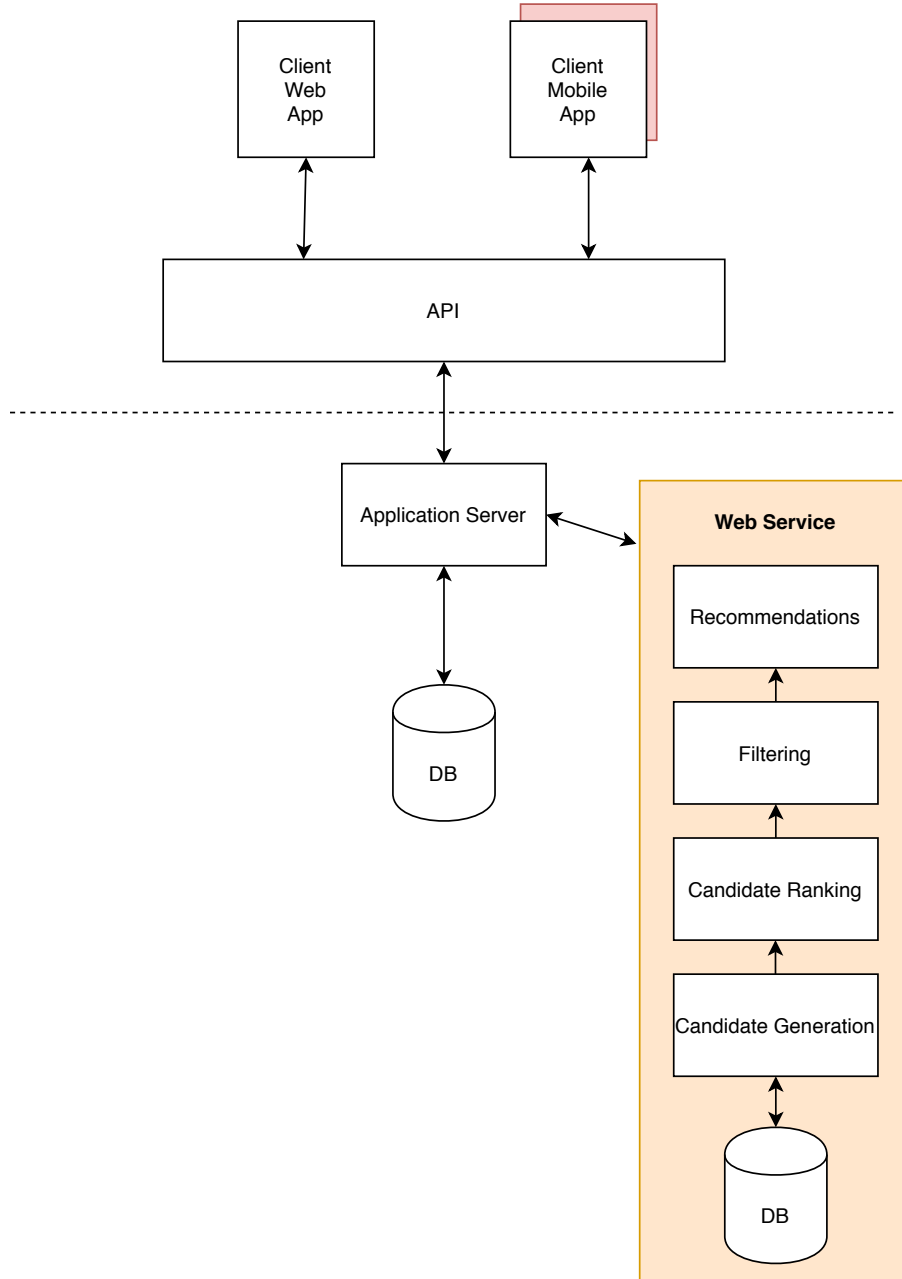


Figure 36: Product A architecture

We propose that the recommender system with its components will live inside a distributed recommendation web service as figure 36 shows.

7.1.1 Architecture of the recommender system

Our recommender system follows a conventional structure of a top-N recommender and do have the following components (as shown in figure 37):

1. Candidate Generation
2. Candidate Ranking
3. Filtering
4. Database

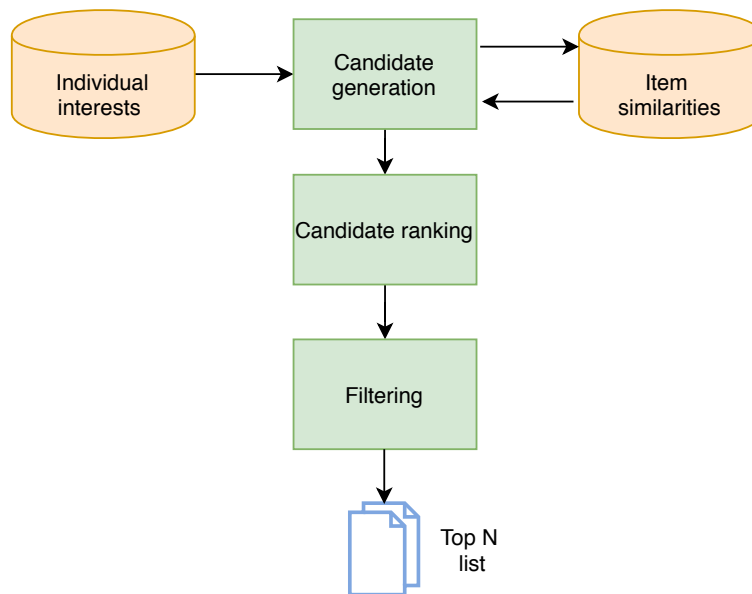


Figure 37: Process architecture

Candidate Generation

During the candidate generation phase, events (or items) from the user has indicated interest in from past activity is used as input, and is returned a subset of items from the database that are similar to those items based on aggregate behaviour.

Candidate ranking

In the process of building up those recommendations we assign scores to each candidate based on the selected properties and can be filtered out if not given a high enough decision boundary.

Filtering

In the third stage, the system takes into account additional constraints before presenting the final list to the user. The filtering component these constraints (i) items the user already have rated, potentially offensive items or items below a minimum threshold.

The filtering component acts as a *stoplist*, should certain terms and keywords be present in the titles, descriptions or categories of the items in the course catalogue. The stoplist can easily be updated and applied quickly should the need arise.

7.1.2 Architectural flow with recommender system

Figure 38 gives an overview of the overall process from requesting and retrieving the course recommendations.

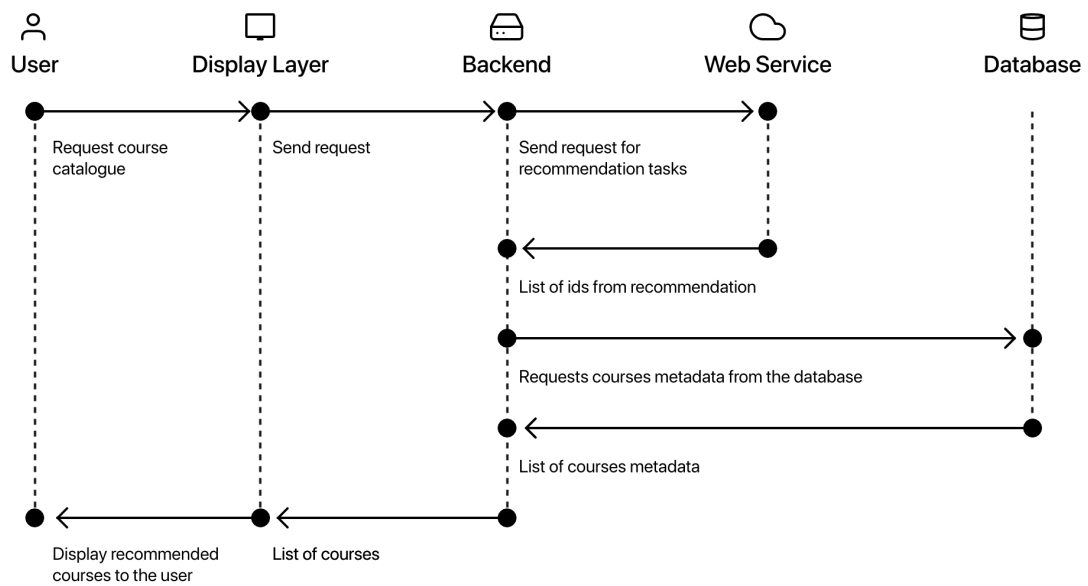


Figure 38: Sequence diagram of how the recommender system work

When a user enters the course catalogue, the client sends a HTTP request to the RS Web Service to fetch a top-N list of course IDs that the RS Engine recommends. After the top- N list is fetched, the application server makes a request to the database to retrieve a list of the course metadata. Whenever this task is completed, the application server sends the list of courses to the client, which presents the recommended courses to the user.

7.2 Implementation of a recommender system in a SPL

During our research we found that there are aspects to consider when using a software engineering approach in which a cross-functional team produces machine learning components based on data, models and code in a software product line:

- (i) the approach should enable the teams to deliver high-quality software efficiently
- (ii) all prototypes of the ML software production process require different tools and workflows that must be versioned and managed accordingly
- (iii) the process of releasing ML software into production is reliable and reproducible even though the model outputs can be non-deterministic and hard to reproduce
- (iv) the release of software prototypes should be divided into small increments, which allows visibility and control around the levels of the variance of its outcomes, thus adding safety into the process
- (v) and preferably keep adaptation cycles short, to create a feedback loop that allows adaptation to the models by learning from its behaviour in production.

Figure 39 shows the recommender system pipeline for our prototype and consist of five phases.

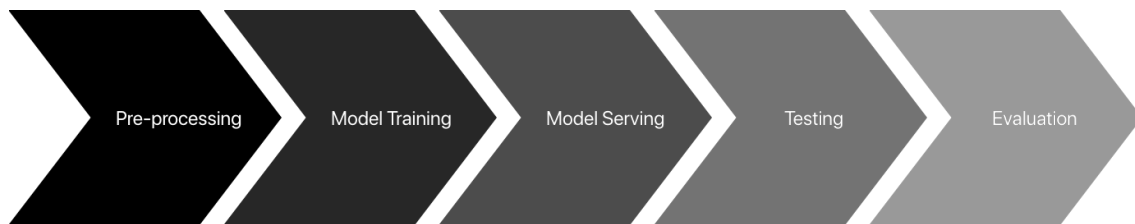


Figure 39: Recommender system pipeline

The first step is to understand the dataset. In our case, it was a collection of CSV files containing information about:

- the *users*, like their roles, their mandatory courses assigned to them, and their completed courses
- the *courses*, metadata about the courses

- the *course reviews*, feedback related to the completion and quality of the course

At this stage, with more massive data sets, it is common to perform some Exploratory Data Analysis (EDA) to understand the shape of the data and identify outliers and broad patterns.

In many organisations, including Snapper, the data required to train useful ML models, will likely not be structured precisely the way a Data Scientist might want; therefore it highlights the first technical component: discoverable and accessible data.

7.2.1 Pre-processing

Within our research and task scope the core transactional systems will deliver our source of data. However, in the real-world, there might be value in brining in other sources of data from outside of the company. Regardless of which flavour of architecture, the data must be easily discoverable and accessible. Building useful models will take longer if the it's hard to find the data needed. In the process, it should be considered that the Data Scientists would want to engineer new features on top of the input data that might help improve the model's interface.

After doing Exploratory Data Analysis, we want to de-normalise the multiple data files into a single CSV file and clean up the data points that were not relevant or could introduce unwanted noise to the model. By using the CSV file to represent a snapshot of the input data, we can devise a simple approach to version our dataset based on the folder structure and file naming conventions. The data versioning has change in two axes: actual sampling of data over time and structural changes to its schema.

Using the CSV file to represent a snapshot of the input training data, we can devise a simple approach to version our dataset based on the folder structure and file naming conventions. Data versioning has a change in two axes: structural changes to its schema, and actual sampling of data over time.

It is worth noting that, in feature implementations, we will likely introduce a complex data pipeline to move data from multiple sources into a cloud storage system like Amazon S3, Azure Storage Account or Google Cloud Storage.

SPL approach to load and segregate data with requirements for model training

The comma-separated values (CSV) format is the most common import and export format for spreadsheets and databases. A CSV file is a delimited text file that use a comma to separate values (Comma-separated values, 2020). Each line of the file is a data record and consists of one or more fields, separated by commas. The CSV file format is not fully standardised, which means that subtle differences often exist in the that produced and consumed by different applications.

This lack of standardisation can cause issues when designing a general-purpose component to handle loading and pre-processing of CSV files. There exist various libraries to handle the loading of CSV files. All of these libraries have a few challenges and limitations. They often lack the requirements to handle common machine learning issues like extracting columns, splitting- and shuffling of datasets, or the ability to convert values to the desired format.

After the initial approach of building a recommender system in a software product line — we recognised the lack of consistency in the datasets and the struggle of pre-processing our data. Instead of creating product-specific variation each time — the process could be eased by creating a reusable component, to be a part of the *core assets*.

In the process of gathering requirements and designing our `loadcsv` prototype to be a part of the core assets of the product line; we established that `loadcsv` should be singular in purpose. `loadcsv` should be responsible for loading up a CSV file and then do a little bit of pre-processing.

This new `loadcsv` component would not have to conform to requirements outside the scope of the software product line. Where most CSV loaders were written in JavaScript, by default output the format as JSON, we could instead output it as a `tf.Tensor`. A `tf.Tensor` is a tensor — a generalisation of vectors and matrices to potentially higher dimensions — internally in TensorFlow, represented as an n-dimensional array of base datatypes (TensorFlow tensors, 2020). As our recommender system will be implemented with TensorFlow, it is favoured to output the CSV in the correct format. Overall, consistency will improve as the developer can have everything in a single component with an easy interface to manage.

During the software engineering process, we expect that the `loadcsv` component would continuously evolve with new features while keeping it in a releasable state. Consistency between relationships has to be maintained throughout software evolution as prototypes ordinarily not are isolated but interrelated. Further improvements will be proposed, and design decisions would continuously have to be made. Thus, when evolving software, it is important to reflect and build

on former decisions. Otherwise, the influx of inconsistent decisions is likely to contribute to an erosion of software architecture or introduce other quality problems. Reflection on former decisions is particularly crucial for long-living software systems where many decisions build on another.

Documenting design decisions is vital since various developers are involved at different times and cannot communicate directly. In this initial implementation, we've defined a few requirement specifications for our CSV loader to be able to support our recommender system prototype in a software product line.

Requirements

Table 9 shows the requirements for the CSV loader to handle ML tasks.

Nr.	Requirements
1	Encoding should be the default, UTF-8
2	Loading a CSV file should yield no extraneous characters
3	Ability to pass in custom options when handling the data
4	Values should have the option to be converted to more appropriate measures, e.g. a Boolean true/false to binary number 0/1
5	Ability to select only the required columns of data, not the entire set
6	Ability to shuffle the data
7	Use custom seed phrase for debugging
8	Ability to split the data
9	Ability to handle large files

Table 9: Requirements for CSV loader

`loadcsv` has to be able to load large CSV files. To achieve this `loadcsv` uses `createReadStream` from the `fs` node module. `createReadStream` is a stream, a data-handling method used to read or write input into output sequentially. Streams help to handle reading/writing files, network communications, or any end-to-end information exchange in an efficient way.

Making `loadcsv` scalable and applicable to several use-cases it was essential to avoid reading a file into memory all at once, but instead, read chunks of data piece by piece, processing its

content without keeping it all in memory.

This makes streams powerful when working with large amounts of data, for example, the file size can be larger than your free memory space, making it impossible to read the whole file into the memory to process it.

Streams provide two major advantages compared to other data handling methods:

- *Memory efficiency*: you don't need to load large amounts of data in memory before you can process it.
- *Time efficiency*: it takes significantly less time to start processing data as soon as you have it, rather than having to wait with processing until the entire payload has been transmitted.

Streams also give the power of "composability" in our code. Designing with compatibility in mind means that several components can be combined in a certain way to produce desired results. We have leveraged the pipe functionality in Node.js to transform data with the conversion option into the desired format.

Trim

Spreadsheet programs can export incorrectly and might accidentally add a couple of empty columns to the CSV file. Values are usually comma-separated, and an empty column would be recognized as trailing commas at the end of the row or the file, similar to ',,,,'. This is a prevalent error that can easily be prevented by trimming input values — ensuring no extraneous characters.

Conversion

Data does not always have the correct form of value and therefore need to be converted before use. Our ideal solution is to pass in a function that can be used to parse just the selected values. Depending on the data set that is being used for a product can then use the module to transform the data as they see fit.

Selection

Not all data fields are relevant when training a model. The developer should have the opportunity to select a few features and labels and then disregard some columns entirely.

Shuffle

Shuffling the training data is generally a good practice during initial preprocessing steps. The purpose of shuffling data is to reduce variance and to make sure that models remain general and overfit less.

As described earlier, it is common to have an 80% / 20% split of training and test data. The split may ignore class order in the original data set. Class labels that might resemble a data set would include target values that resemble the following:

For example: [0, 0, 0, 1, 1, 2, 2, 2, 3, 3]

In the example above, splitting data without shuffling might lead to poor performance in test set evaluation. Only classes 0, 1, and 2 would be captured in the training data, and only class 3 would be represented in test data. The training/test/validation sets should be representative of the overall distribution of the data.

The ability to shuffle the entire dataset is crucial for algorithms such as K-Nearest Neighbor, Stochastic Gradient Descent, or when performing batch type testing.

Gradient Descent algorithms are susceptible to becoming "stuck" in numerous local minima while a better (deeper/lower) solution may lie nearby. This is liable to occur if X is unchanged over each training iteration because the surface is fixed for a given X ; all its features are static, including its minima.

The idea behind batch gradient descent is that by calculating the gradient on a single batch will return a reasonably good estimate of the "true" gradient. This way, computation time is saved by not having to calculate the "true" gradient over the entire dataset every time.

Data should be shuffled after each epoch to not run into the risk of creating batches that are not representative of the overall dataset, and thus, the estimate of the gradient will be off.

In Stochastic Gradient Descent, each batch has size 1. Data should be shuffled after each epoch to keep learning general and to create independent change on the model. Otherwise, each gradient will be biased with what updates the previous data point (Gowda, 2017).

Furthermore, in our implementation, we need to address how to make the indices match up to the same labels. We could either (1) shuffle data before it is parsed or (2) use the ShuffleSeed

library. ShuffleSeed is a library to shuffle an array of records. ShuffleSeed has the option to use a seed phrase to determine how the array of records are shuffled. Given two identical phrases, the labels will be shuffled in the same order.

The side-effect that subsequent shuffles — e.g. rerunning the analysis — data will always be shuffled in the same order, each time. This is a beneficial standpoint when debugging because it means that developers can get equal and derandomised results when debugging. Split data In essence, the whole point of a machine learning system is to be able to work with unseen data. The data is split into a training and a test set. The training set is used to train the model, and the test set is used to test the accuracy of the model. Typically the distribution between the sets is 80% / 20%.

In this implementation, we use the simplest method of splitting data, which is to split it serially. The first 80% of rows are put into the training set. Remaining 20% of rows is put into the test set. In order to prevent overfitting, the distribution of training samples for each classification, or range if the label is a real value has to be fairly equal. If the training data is overly unbalanced, the model will predict a non-meaningful result. We can achieve this by shuffling the dataset before it is split.

In the current implementation, 80/20 partition is the default split value unless you specifically decide to specify a number.

A screen shot of the code in `loadcsv.js` can be found in appendix D.2.

7.2.2 Model training

Once the data is available, we move toward the iterative workflow of model building. This usually entails splitting the data into a training set and a validation set, which we already have achieved with our CSV loader and then trying different combinations of algorithms, and tuning their parameters and hyper-parameters that produces a model that can be evaluated against the validation set, to assess the quality of its predictions. The step-by-step of this model training process becomes the machine learning pipeline.

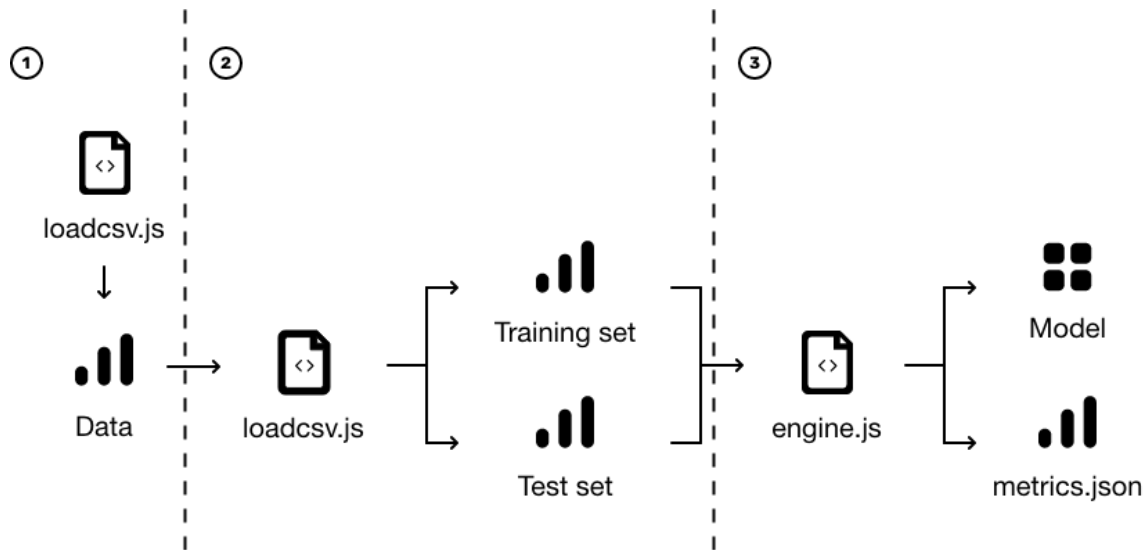


Figure 40: ML pipeline for the recommender system

Figure 40 shows how we structured the ML pipeline for our recommender system, highlighting the different source code, data, and model components. The input data, the intermediate training and validation data sets and the output model can potentially be large files, which we don't want to store in the source control repository. Also, the stages of the pipeline are generally in constant change, which makes it hard to reproduce them outside of the local environment but something we have tried to address with *seed randomisation* in `loadcsv`.

In order to formalise the model training process in code, we used TensorFlow, which provides tools to solve ML-specific problems. We have a background in web development and no knowledge of machine learning prior to the thesis. We chose to use the framework TensorFlow for our second approach of building a software product line for the recommender system since we have experience with Node.js and JavaScript beforehand. We also think it will benefit from using the same language across the entire stack. TensorFlow is an open-source library for numerical computation and large-scale machine learning. TensorFlow distributes processing across various CPU cores on the computer, or across multiple machines on a network and enables solving computing problems in a distributed manner. Having the ability to run on various machines allows the opportunity to push processing down to the end user's device. Lost internet connection in an autonomous device can have fatal consequences, e.g. a car. Instead, the neural network could be running from an embedded computer in a vehicle.

At a low level, TensorFlow distributes mathematical operations on groups of numbers or tensors, which is an array or a matrix of values.

Implementation of our recommender prototype has been written in JavaScript with the `*tfjs*` library. TensorFlow.js is an open-source hardware-accelerated JavaScript library for training and deploying machine learning models [<https://github.com/tensorflow/tfjs>]. TensorFlow provides native TensorFlow execution under the Node.js runtime with the same TensorFlow API.

Design constraints when building a recommender system for Snapper

We ran into the issue of most of the courses in Snapper's course catalogue is mandatory. Making recommendations based on previous behaviour with a small course catalogue (approx. 100 courses) can be hard. When designing the recommender system, we decided to base our recommendations on explicit data, e.g. feedback from the user.

We decided in this implementation to look at the problem as a classification problem with multiple output values. We would like to make a prediction on how a user would rate a course based on past ratings by other users.

Engine implementation

We decided to solve the task with *multinomial logistic regression* to classify which star-rating a user would give a course based on explicit and implicit data. Then use the model to predict the user's rating for multiple courses and present a top-N list. Multi-nominal logistic regression is a regression model that generalises logistic regression to classification problems where the output can take more than two possible values (as described in chapter 3.2.3). Logistic regression makes use of one or more predictor variables that can be either continuous or categorical and predicts the target variable classes. Logistic regression model output is helpful in identifying important factors that will impact the target variable and the nature of relationships between each of these factors and dependent variables. While we use the model in a limited form for a recommender system the models purpose is to be as reusable so it can be used for other purposes in the software product line. The algorithm is useful in identifying the relationships of various attributes, characteristics and other variables to a particular outcome. Based on the attributes of a user or employee e.g., gender, income, age, competences, etc., analysis with the model can check the level of likely satisfaction with a product or with the job.

We decided to solve the task with *multinomial logistic regression* to classify, which star-rating a

user would give a course based on explicit and implicit data. Then use the model to predict the user's rating for multiple courses and present a top-N list. Multi-nominal logistic regression is a regression model that generalises logistic regression to classification problems where the output can take more than two possible values. Logistic regression makes use of one or more predictor variables that can be either continuous or categorical and predicts the target variable classes. The model output helps identify important factors (X_i) that will impact the target variable (Y) and the nature of relationships between each of these factors and dependent variables. In other words, useful in identifying the relationships of various attributes, characteristics and other variables to a particular outcome.

While we use the model in a limited form for a recommender system, the purpose of the model is to be reusable as enabling other use-cases in the software product line. Based on the attributes of a user or employee, e.g., gender, income, age, competences, and so forth, analysis with the model can check the level of likely satisfaction with, e.g., with the job/position, a course or a product.

Implementation details

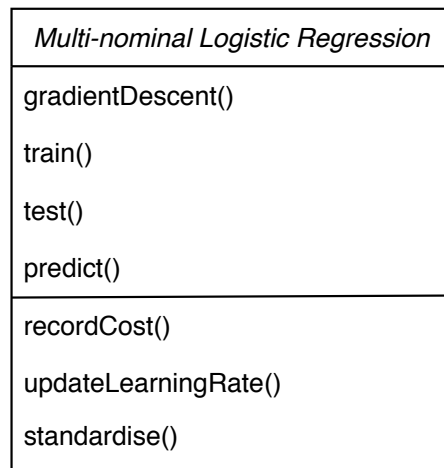


Figure 41: LogisticRegression class diagram

Our `engine.js` file consists of a multi-nominal logistic regression class. This is shown in figure 41. The following sections explain each function in the class.

A screen shot of the code in `engine.js` can be found in appendix D.1.

gradientDescent()

The `gradientDescent()` is given a set of parameters chosen at random, and then uses the learning rate to adjust those parameters until the error minimises itself trying to find the local minima. The learning rate has to be continuously updated and is handled by the `updateLearningRate()` helper method.

The measurement of error in the learning system is cost, e.g., how close the predicted values are to the observed values.

`gradientDescent()` is implemented with the sigmoid function. Given that the sigmoid function is convex for values less than 0, and concave for values more than 0 — the slope can possess multiple optima. Either with a single concave or convex function means that depending on our position towards a global minimum of that line or equation, we can increase or decrease B. The formula for gradient descent is shown in equation 8.

$$\theta_j = \theta_j - \alpha \frac{\alpha}{\alpha \theta_j} J(\theta) \quad (8)$$

Gradient descent formula

recordCost()

In `recordCost()` a cross-entropy loss function, is used with the classification model as it returns a probability value between 0 and 1. We utilise cross-entropy instead of Mean Square Error because it penalises incorrect classifications more than the close ones. The cross-entropy formula is shown in equation 3.

train()

Both Batch Gradient Descent and Stochastic Gradient Descent can be utilised in our code to get convergence on values of M and B than attempting to iterate through the entire dataset before updating. Using either approach will help find the optimal values of M and B with fewer iterations.

Essentially the process of Batch Gradient Descent is (1) guessing the starting value of B and M, (2) calculating the slope of cross-entropy using a number of observations in the feature set with current M and B values, (3) multiplying the slope with the learning rate and (4) then updating B and M.

This implementation works for both Batch Gradient Descent and Stochastic Gradient Descent. With Stochastic Gradient Descent, the process would be the same, but we would use one observation at a time to update B and M instead. Choosing either Batch Gradient Descent or Stochastic Gradient Descent is done when configuring the initialisation of the regression class.

predict()

`predict()` returns the prediction of an observation. The observation is matrix multiplied and applied the sigmoid equation. Sigmoid takes a real value as input and outputs another value between 0 and 1. Equation 9 show this calculation where *Prob* is the calculated output.

$$Prob = \frac{1}{1 + e^{-(m \cdot x + b)}} \quad (9)$$

Sigmoid equation

test()

The test function uses the `testFeatures` and `testLabels` tensors with the predict function and then utilises Marginal Probability Distribution to consider possible output cases in isolation. We can use the prediction accuracy as a criterion for assessing model performance (formula 10). The prediction accuracy can be used for a threshold, e.g., a prediction accuracy above 70% is useful.

$$\frac{predictions - incorrectPredictions}{predictions} \quad (10)$$

Accuracy formula

standardise() and processFeatures()

The class has two other helper methods (i) `standardise()`, and (ii) `processFeatures()`.

(i) `standardise()` rescales attributes, so they have a mean value of 0 and a standard deviation of 1. Standardisation is used because it accounts for more extreme values than normalisation does.

(ii) `processFeatures()` takes in a feature set and utilises the standardise helper method to return a standardised feature set.

Hyperparameter optimisation

Table 10 shows the hyperparameters we used in `engine.js` for the multi-nominal logistic regression class.

Parameter	Descripton
<i>iterations</i>	Number of times the batch size is passed through the algorithm
<i>batchSize</i>	Number of entries in training set utilized in one iteration
<i>learningRate</i>	The amount that the weights are updated each iteration, varying from 0 to 1

Table 10: Description of the hyper parameters we use

In our dataset of approximately 1000 entries — we split the training and test set into an 80/20% split. We found that a high number of iterations (above 100) and a small batch size (less than 32) increases the probability output for the model, i.e., it is more accurate. The learning rate didn't change the output much unless it was above 0.5, which in turn made the model less precise.

The final set of parameters with a test set of approximately 200 entries is shown in table 11.

Parameter	Value
<i>iterations</i>	100
<i>batchSize</i>	10
<i>learningRate</i>	0.5

Table 11: Values for the hyper parameters we used

We only conducted a high-level analysis of hyper-parameters to see how the probability output of the model changed. Our reason for simplifying this process is; because this is an area that otherwise would have required multiple datasets with higher quality and an in-depth deep dive into the topic, to be useful for our research on machine learning components in a software product line.

Once we are satisfied with the model, the model has to be treated as an prototype that needs to be versioned and deployed to production.

7.2.3 Model serving

Once a suitable training model is found, it has to be decided how it will be served and used in production. In our initial approach, as mentioned earlier, we utilised an embedded model. Which is a more straightforward approach, where the model prototype is a dependency built and packaged within the consuming application. The application prototype and version is a combination of code and the chosen model. This was a lot simpler since both the application code and the chosen model was written in JavaScript.

In this second approach, the model shall be deployed independently of the consuming applications. This allows updates to the model to be released independently, but it also introduces latency at inference time, as there will be a remote invocation required for each prediction.

Many of the cloud providers have SDKs and tools to wrap the model for deployment into their MLaaS (Machine Learning as a Service) platforms, such as Azure Machine Learning, AWS Sagemaker or our chosen Google AI Platform. We have decided to go with the Google AI Platform as there are more dedicated resources for deployment with TensorFlow. When thinking about which provider to chose, it is crucial to evaluate the right option for the product needs. There is no clear standard (open or proprietary) that can be considered optimal since it is a current area of development, and various tools and vendors are working to simplify this task.

Regardless of using a pattern with an embedded model or model as a service, it is worth noting that, there is always an implicit contract between the model and its consumers. In our case, the model expects input data in the shape of a `tf.Tensor` and changes to the contract, to require new input or add new features can cause integration issues and break the applications using it. Therefore the application and the model must be documented and tested thoroughly in the Software Product Line.

7.2.4 Testing, experimentation and evaluation

Several types of tests can be included in the ML workflow. Overall, testing and quality for ML systems are complex and should be the subject of another in-depth paper. Notwithstanding, we will propose a few tests that can add value and improve the overall quality of the ML system:

Validate data

A validation test can be utilised to validate input data against the expected schema (e.g. not

null or fall within expected ranges). For engineered features, unit tests can ensure that the features are calculated correctly — e.g. numeric features are scaled or normalised or missing values are replaced duly.

Validate component integration

For validating component integration, a similar approach to the testing of integration between services can be utilised. Contract Tests can be used to confirm compatibility between the expected model interface and the consuming application. The exported model should still produce the same results when the model is productionised in a different composition. Relevant tests can be performed by running the original and productionised models against the same validation dataset to ensure that the results are equal.

Validate model quality

We are interested in the accuracy of the model. Metrics such as error rates, accuracy, recall, etc. can be used to evaluate a model's performance (also useful during hyperparameter optimisation). Threshold Tests of the metrics can be included in our pipeline as quality gates to warrant against that new models degrade upon a known performance baseline.

Validate model bias and fairness

There might be an inherent bias in the training data with more data points for a given value of a feature (e.g. gender) compared to the actual distribution in the real world. Thus, there should be included tests to check how the model performs against baselines for specific data slices.

Managing and curating test data is essential, but trying to assess a model's quality holistically is hard. Should we compute the same metrics against the same dataset, over time, we can overfit. Also, should we have other models already live, it is crucial to ensure that the new model version will not degrade against unseen data. When models are distributed or exported to be used by a different application, issues with engineered features being calculated differently between training and serving time can occur. An approach equivalent to Integration Tests in traditional software development to help catch these types of problems could be to distribute a holdout dataset along with the model artefact. The consuming application team can use the holdout dataset to reassess the model's performance after it is integrated.

We will not delve any deeper into the topic of testing during this thesis, but it is undoubtedly essential when developing software product lines with machine learning components. Given a

chance, we will research this in the future.

8 Results from evaluation

We have conducted a case study where the prototype has been analysed and observed by three test subjects or experts. Then we have interviewed with each expert to get answers on questions related to the prototype and the research questions. Two of the experts are software developers; in their interviews, we were focusing on the architectural point of view. In the last interview with the CTO from Snapper, we focused on a business and process view (as figure 16 shows, from chapter 4.6).

We also published a survey through Google Forms with 11 questions on recommender systems and peoples trusts towards them. After publishing the survey, we received 157 replies.

We have read through the three interviews separately and analysed the results according to the methods presented in chapter 4.6. We have been marking the text with a colour code (representing each research question) to structure the interviews, and then compared them. For the survey we have presented the data we got using descriptive statistics to analyse the results (also described in chapter 4.6). In the following subsections (8.1 to 8.4) we present each research question and a summary of the results and findings we got from the interviews relevant to each question. In the last of these subsections (8.5) we present the results from the survey.

8.1 RQ1: How possible is it to create machine learning components that work for multiple products in a software product line?

Architecture

In the code base of the prototype, there are no repetitive lines of code that require any useful abstractions. If the code base grows in the future because more features are added and the complexity increases, it might be relevant, for now, it may only create overhead. The code follows “best practices”, as it does what it is intended to do, and is written in a clean and simple way. Both experts would have written the syntax with minor changes (e.g. one expert prefers arrow functions).

The codebase fulfils the functional requirements and is structured as the predefined architecture was defined. It has a good implementation of TensorFlow and Lodash (JavaScript framework) and has few hard coded variables, the ones that are used are necessary to have. The CSV loader can handle large amounts of data but could improve more on this. To create a new product to

the SPL, one expert estimates a workload of 2 to 3 days, and the other suggests 1 to 2 days and half a day to modify the CSV loader.

Business

The impact an SPL with ML has on the business performance is that the system (that Snapper is developing for Company A) may become more relevant by offering courses to the employees. If it is something the current system lacks, it has to be displaying relevant information, and the current version has a lot of static information. With an SPL, Snapper can offer a product that the end-user will experience as a more flexible and relevant system. This relieves focus from the content distributor.

SPL engineering can be introduced to the company (Snapper) by having better structuring on the work routine, and more focus on gaining the right competence. It is important to ask “What is it that the employees need to learn in order to have the necessary tools and experience?”.

The SPL prototype would have a remarkable impact on the user’s interaction with the system and increase its value and its future evolution. It may, therefore, be very marketable for the company. The expert claims that *“it is easier to develop ML and AI now, more than before. Customers (e.g. Company A or Company B) will have more faith and belief that the product they are buying is relevant to the time”*. The expert believes that the process of adopting new products to the model or creating new instances to the SPL would be cost-effective in the long term.

The market is very mature but is maturing day by day. Customers rarely change their systems because of the content and experience required to do so, and roles and requirements affecting it at the same time. They are rarely in a position where they can change system providers. The normal time for a project with all the data is a process that is spanning over a year. Trends (courses, content, structure and design of systems) are changing, but the system itself is rarely swapped out.

Process

The process of adopting new instances to the SPL may require a cross-functional team, but that depends on what the tasks are. To add a new service in the form of black-box or to integrate an API that communicates with a web service (that is the ML model), it may not. But to develop and maintain a full lifecycle of an SPL with ML components, a cross-functional team is required

to have internal in the company. In the cross-functional team, you need someone to validate the data and someone to develop the system or the service. This team does not have to be divided into domain- and application-specific teams.

SPL may give better quality to the end-user in the form of usability, but integration with other systems can increase risk and the need for testing. The experts point out that "*there are risks when integrating with third-party services regarding versions and change*". In terms of reliability and safety, it might not have too big of an impact in the short term, but over time it can have valuable benefits. There will always be some start-up expenses that are needed to be accounted for as well.

Summary

Creating ML components for an SPL is possible, and it is easier if components are well structured and written cleanly and simpler. Abstractions may only lead to too complex integrations. It can be very beneficial for companies to implement SPL with ML components, but it requires the right structure and competence of the company. The process of doing so does not need a cross-functional team.

8.2 RQ2: How reusable are machine learning components in a software product line?

Architecture

The codebase follows the standard guidelines for the structure, no specific design pattern has been used, but the experts evaluated it as not necessary. Default configurations are implemented by passing them through parameters and deconstructing them afterwards. The functions and classes are well-sized and with the right amount of responsibility. The main class can be de-structured a bit if it is preferable, but not needed to improve the reusability. One of the experts also said that "*the CSV loader could have a bit more structure for making it easier to reuse*".

The code is reusable in terms of following the DRY (Do not Repeat Yourself) principle. The components, services and functions are reusable for an SPL. In terms of generic functions and classes, the code is also reusable. The code follows the OOAD (Object-Oriented Analysis & Design) principle and the single responsibility principle partially. Both experts point out this, and one expert further stated that "*this is not necessary unless the plan is to integrate for more formats*".

Meets non-functional requirements, but it depends on which ones that are relevant to measure. The important ones that the code base follows are readability, supportability, performance and extensibility. The code snippets in themselves are not that reusable because they are quite specific. The experts said that *"you cannot reuse them in other parts of the application except the helper methods. That would not be the goal anyway"*. On a high level, the CSV loader component and the engine components are entirely reusable and can be reused to compose other variations of the ML model.

Business

You start by creating a general approach (R_C), this would be based on the first product instance (Snapper Bits, which is already made). Then you start adding new products to the product line, for example, Company A and Company B, being product R_1 and R_2 . NIF is also a typical customer for Snapper Bits.

Process

"The SPL would contain as many products as possible that it can support", but the experts also state that it is hard to give a precise estimate to this. Everyone (customers) needs such products, but there may be a lot of product-specific changes to the products in the start. When maturing the SPL, the products might be more general in the future, to get more out of them. *"How many of the products that need product-specific configurations to depend on how long it takes to implement the product-specific configurations"* states the expert.

Summary

ML components can be reused but should be well-sized and structured in order to do so, and with a fitting amount of responsibilities. The CSV loader and the engine is very reusable if properly written. Create a specific product instance and then derive it into generic components that can serve the SPL with as many products as needed.

8.3 RQ3: How feasible is it to create and consume reusable machine learning models in a software product line?

Architecture

Both experts pointed out that *"the code is easily maintainable"*. And that *"through an API the SPL can consume and reuse ML models"*. With the default parameter option, the code can be extended without affecting the API.

The code follows the right conventions and makes good use of available libraries was the expert's opinions. A reusable CSV loader allows for more SPL products to consume and use the same ML model. One expert pointed out that in the CSV loader, it might be useful to add the option to use delimiter values if the data sets vary.

Business

The products from the same SPL can support each other over time on the market because some customers ask for the same functionality in their product. For example, Company C wants similar functionality as Union and M2 wants. The expert stated that *"it may be possible for customers to merge an order, or to have the same order on a product that is composed of new components, and share the expenses"*.

In a growing market, it can be relevant to branch out to smaller customers with less strict requirements, that have less information flow. The expert claimed that *"it is easier to compete in a less mature market with fewer "edges"*.

Process

The process of adopting new products to the ML model would not be time-consuming or demanding. Snapper is doing similar work today, so the expert said that this is possible. He also estimated it to be cost-effective in the long term. However, it is still a risk to utilise ML in applications for Snapper because it is a relatively small company.

The SPL prototype with a simple recommender system is a *"basic"* requirement for the customer. The expert said that *"when you get a model that serves as a personal trainer that monitors the evolution over time of an end-user, then you are in the "delighters"-category"*. In the start, a lot of the products are product-specific but becomes more general after some time. When the products become more general, the expert claimed that *"it is easier to use the same ML model for multiple products in the SPL"*.

Summary

Multiple products in an SPL can consume the same ML model through an API. A shared and reusable CSV loader can also allow for this, and good use of libraries. If multiple products consume the same ML model, they can support each other over time, and be beneficial for each other. This is easier when the SPL contains more product.

8.4 RQ4: How can you support a Software Product Line Evolution containing Machine Learning Components?

Architecture

Both experts had the opinion that the codebase and the interface are relatively simple and easy to understand, but it requires some initial explanations. Useful abstractions may have an expense of readability and maintainability. The experts said that *"as of now, the codebase seems easy to maintain, debug and test"*.

The seed phrase in the CSV loader makes it easy to test. The function `splitTest` should be renamed. It is useful to know that it is index-based. A feature should be included to show that it is formatted to display the results. One expert's opinion on the CSV loader was that it was *"good implementation that it converts the results straight from the input, by placing an object in and receiving an object out"*. One experts stated that it was no need to split the code in the CSV loader more. The engine is well structured. Without the stream implementation in the CSV loader, the system would have been struggling with large files. One expert stated that *"the prototype could batch the file input, but it looks like the right solution"*. The code, therefore, can use huge amounts of data.

The code is readable, supportable and extensible and with excellent performance (hence easy to support the code for further evolution to the SPL). These are some important non-functional requirements. There are only a few hard coding and some configuration values. Both experts pointed out that framework features are used extensively.

Business

As it is today, customers would regularly deliver more requirements about functionality to the system Snapper delivers. The expert claimed that *"the SPL will evolve on how relevant Snapper wants the system or application should be towards the end users"*. He pointed out that the plan is to use the same component library, but changes will mainly be focusing on content and data handling rather than the codebase itself.

The plan is that Product A (currently existing app for Company A) becomes a product in the SPL (R_n). Some of its functionalities are drawn out to a general system. Existing products will become a part of the SPL, then some products from the same SPL can supplant each other over time. But this will happen more of natural evolution rather than out-competing each other.

To mature the SPL is a step by step process, and the expert estimates it to go from prototype to production code and then a product that is ready to be deployed. This will take between 3 to 6 months. The prototype is a simple *“proof of concept”*. A product has several shapes, and a first version may also have a lifetime of 3 to 6 months before it is being replaced by a newer version that offers more value. This is a process that has to be conducted iteratively until the product is mature enough.

Process

Dividing the team into a domain- and application-specific teams can be interesting and relevant if the development team is growing to a large group. Then, it can be relevant to have someone just working on core functionality, and to have dedicated software developers to specific tasks or problems. The expert claimed that *“when you have developers that work on core functionality and application implementation simultaneously, there can be scenarios where they can take an easy way out and use “quick fixes” to solve a problem faster”*. He further explained that a dedicated core developer would have more interest in developing for stability and arrange for expansion and evolution. In contrast, an application developer often works to deliver a product on a short deadline.

It is good practice to use or learn from the experience of big suppliers that have experience with delivering services and systems like this when creating the architecture. They can offer good availability of the system. Using them when developing services like authentication, file management, and preferably ML components, will make the development easier and ensure more security, stability and quality to the product.

The expert pointed out that *“the process of supporting product-specific configurations usually is work for consultants, rather than software developers”*.

Summary

Making readable, structured and extensible code makes it easier to maintain and support while being part of an SPL. Abstractions and hard coding can make it harder to support ML components. Maturing an SPL is an iterative process, and it takes time. If a large development team is working on an SPL, it might be relevant to split up the team with different areas of responsibilities.

8.5 RQ5: How does a software product line affect the quality of its recommender systems?

Before the respondents began answering the survey, they read a short description of the background of what recommender systems are. The complete survey with description and questions can be found in appendix C. When we describe different options to choose from they range from 1 to 7. Here, 1 is not likely or not important, and 7 is likely and important.

The respondents differed in all ages between 15 to 80. Of all the 157 replies 77.4 % understood what a recommender system is, and only 3.2 % did not understand it. 19.4 % understood the gist of it. 36.7 % of the respondents said they are exposed to recommender systems daily, and 30 % said they are exposed multiple times daily. That is a total of 66.7 % (or $\frac{2}{3}$) of the respondents that say they interact with recommender systems on a daily basis.

Figure 42 shows the respondents answers to if they think recommender systems will help them make decisions more efficiently or not. 22.3 % percent (35 people) believe that it is very likely (answering option 7) that the recommender system will do so. 80 % percent (120 people) chose option 5 or better, which means that they say it is likely or very likely.

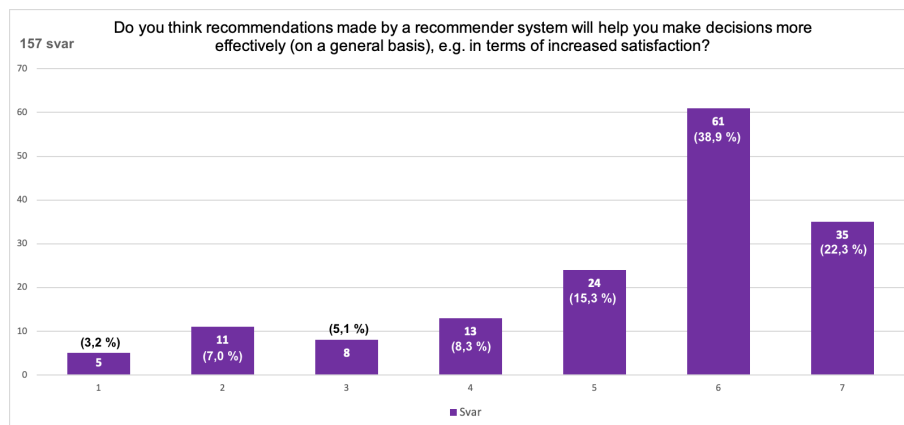


Figure 42: Diagram showing the degree people think recommender systems make them do decisions more effectively

Figure 43 shows the respondents answers to if they lose trust in recommender systems if bad recommendations are responded by it. 54.6 % percent believe that it is very likely (answering option 6 or 7), and only 22 % percent think that it is not likely or less likely (answering option 1, 2 or 3).

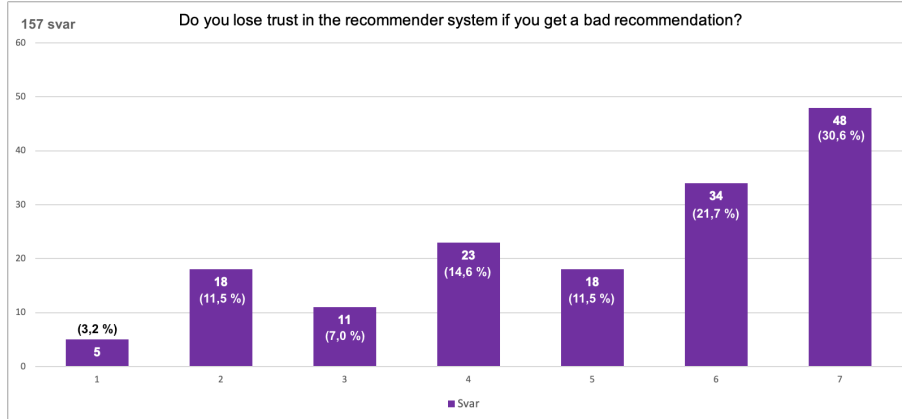


Figure 43: Diagram showing if people lose trust in recommender systems if bad recommendations are responded

38.7 % percent of the respondents say that they are sometimes annoyed by bad recommendations. 29 % percent say they are often annoyed, and 16.1 % percent say that they are annoyed very often. Only 6.5 % percent say that they are never annoyed.

When it comes to the question of whether people want to know if a recommender system is responsible for displaying content or not, the respondent's answers are evenly divided. 58.1 % percent say that it is in between important and not important that this is described (answering option 3, 4 or 5). The same goes for the question about whether people want to get an informative explanation to why they received a certain recommendation. The response is evenly divided were circa 60 % answered that it is reasonably important (answering option 3, 4 or 5).

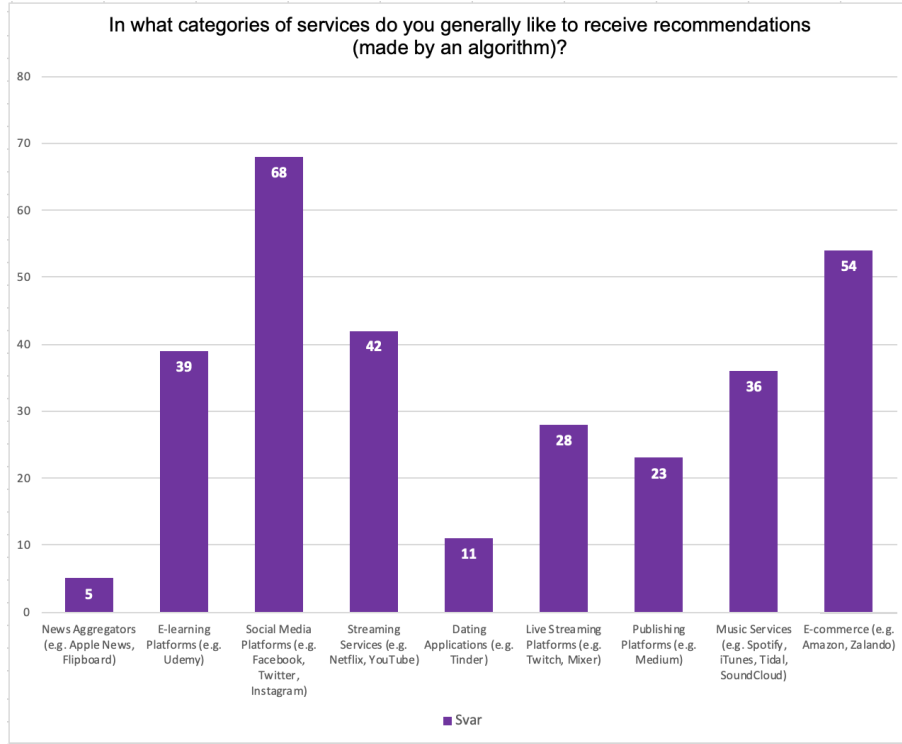


Figure 44: Diagram showing the distribution of different categories of services where people generally like to receive recommendations from

Figure 44 shows the respondents answers on which categories of service they generally like to receive recommendations from. On this question, respondents were allowed to chose as many answers as they wanted (hence more replies than 157). The majority of the answers are in social media services, with 68 answers. We also see a high amount of respondents answering E-commerce services (54 answers) and E-learning platforms (39 answers).

Summary

In our survey, we did not explicitly ask for the opinions of the respondents towards software product lines as we have noticed a general lack of knowledge on the topic, even in the IT-community. Therefore, we were most interested in opinions on the quality of recommender systems. Our purpose was to gain insights into how recommender systems influence the attitudes of the respondents. An issue that became clear by conducting a survey is that poor recommendations can be hurtful for business. As we will discuss in a later chapter, the goal of numerous software product line engineering approaches is automatic product derivation. This

leads us to the conclusion to look for an approach that will let us build software product lines in a flexible way and that will ensure product quality of the recommender systems. A sub-optimal or next-best approach will not suffice.

We state this because a majority of the respondents answered that they lose trust if bad recommendations are responded. This implies that it is important with accurate recommender systems for business to gain benefits from them. A majority of the respondent also say that they get annoyed by bad recommendations, which substantiate this argument. In the final question, we see that a big amount of the respondents answer that they generally receive recommendations from either E-learning platforms, publishing platforms or E-commerce services (a total of 116 of 156 respondents). This is a high number, and this is important to study because, in these services, SPLs have a great influence on the market, and are used frequently.

9 Lessons learned

In this chapter, we will discuss why we chose to use a reactive and feature-oriented composition-based approach to composing a software product line. Our discussion on the topic will be based on first-principles to establish why it serves as an excellent foundation to derive products in an SPL. Afterwards, we will discuss the building blocks of the composition-based approach, namely components and services. The discussion will consist of how to reuse and size the components. Our result on this topic is derived through an exploration of previous research articles and books on the matter.

Moreover, we will discuss version-control and feature tracking based on our own experience in building software. Lastly, we will discuss our judgments and results of recommender systems in software product lines. The results of the aforementioned subject will be heavily based on our survey results and experience gained through a DSR approach with the results from the initial and implementation phase. The discussion chapter will give a holistic view of our research results and propose what we consider an ideal approach to build software product lines with machine learning components.

9.1 Reactive and feature-oriented approach

The process of software development is to break down large problems into smaller problems by building smaller components that solve those smaller problems and then compose these components to form a complete application. The atomic units of composition are *functions* and *data structures*. The composition of these atomic units defines application structure. Gang of Four states "Favor object composition over class inheritance" (Gamma et al., 1994, p. 32). Class inheritance can be used to produce composite objects, where a class inherits parts of its functionality from a parent class and extends or overrides the parts. This approach causes a lot of design problems with an **is-a** thinking, e.g., "a duck is-a bird". A few changes to the base class can potentially break a large number of descendant classes due to tight coupling, also known as the *fragile base class problem*. Even breaking code that the author is not aware of. Besides, the *inflexible hierarchy problem* can arise with a single ancestor taxonomy. During time and evolution, all class taxonomies are eventually wrong for new use-cases. Due to inflexible hierarchies, new use cases are often implemented by duplication, rather than an extension. This can lead to similar unplanned classes which are variant. Once duplication occurs, it might not be obvious which class new classes should descend from—thus leading to the *duplication by*

necessity problem.

Duplication can be removed by writing a component (a function, module, class, etc.), then giving it a name (identity), and reuse it as many times as needed. Successful abstraction means that the result is a set of independently useful and recomposable components. *Abstrahere* is the Latin word for abstraction and means "to draw away" and in the context of software development; it is the process of considering something independent of its associations. Hence, software solutions should be de-composable into their component parts, and re-composable into new solutions, without changing the internal implementations details.

The process of abstraction is done by *generalisation* and *specialisation*. Generalisation implies the process of finding similarities in repeated patterns and hiding the similarities behind an abstraction. Specialisation is the process of supplying only the things that differ for each use case. Abstraction is the process of deducing the underlying essence of a concept. By leaving our headspace for a moment and viewing a problem from a different perspective, we can explore the common ground between various issues in different domains. Proceeding into this thesis, we aspired to see software product lines from first principles. First-principles thinking is the act of understanding a process from the fundamental parts that are true and then build from there (Clear, 2017). Building a software product line should emulate the process of building software by its most fundamental parts. In feature-oriented product line engineering, two approaches are used widely: *annotation-based* and *composition-based*. They differ in the way they represent variability in the code base and how they produce products. Component composition favours having all features as independent parts rather than merged in a single code base.

9.1.1 Composition-based approach

Consequently, a composition-based approach locates code belonging to a feature or feature combination in a dedicated file, container, or module. This can relieve the overall architecture of the duplication by necessity problem. The software has to be updated (or changed) to stay relevant and deal with varying conditions such as new customer requirements, changes to the environment, security threats, etc. We believe in the importance of the Agile Manifesto and staying agile as, during time and evolution, all taxonomies are eventually wrong for new use-cases. Issues arise as the whole software product line process involves a form of pre-planning, and information about anticipated variations is a prerequisite for the proper design of a product line. A way to deal with this is by anticipating potential requirements in terms of features over

the entire domain, instead of as individual software systems.

9.1.2 Reactive approach

A reactive approach has the closest bout to stay true to the Agile Manifesto, as it begins with a small and easy to handle product line (might only consist of a single product) and is extended incrementally with new features and implementation artefacts (extending the product line's scope). The software product line begins at SPL_0 as an initial version release of the software product line. Afterwards, the product line can progressively grow in incremental steps from SPL_i to SPL_{i+1} towards its ideal. The ideal would mean covering the full variation spectrum as defined during domain analysis (which might also be incremental).

The reactive approach allows for exploration and characterisation of the requirements leading to a new product currently not covered by the product line. Besides being an adoption path, the reactive approach is implicitly in its form the way the other two adoption paths would maintain and evolve its product line over a lifetime. A reactive approach is substantially simplifying the entire process.

Strunk Jr. states that "*...parallel construction requires that expressions of similar content and function should be outwardly similar. The likeness of form enables the reader to recognise more readily the likeness of content and function*" (Strunk JR., 1999, p. 35).

Through further reasoning, we may say that a reactive approach is more suitable as it requires less planning than a proactive approach. Despite the trade-off, by including a feature that may be invasive and expensive, as the SPL was not designed with that feature in mind. At the same time, the reactive approach is more structured than the extractive approach because each iteration follows clear planning steps.

9.1.3 Feature-orientation

The idea of feature orientation is to organise and structure the entire software product line process and software artefacts involved as features. The concept of a feature is used to distinguish the products of the product line. Following this concept, it is easy to trace the requirements of a customer to the software artefacts that provide the corresponding functionality. Features would be explicit in requirements, design, code, and testing — across the entire life cycle. The broad spectrum of requirements from the stakeholders is the desire for variable software. It allows for

tailoring variable software by purpose, rather than serving the needs of a particular stakeholder.

9.1.4 Software composition with mapping to SPL

In software composition abstraction may take many forms:

- Algorithms
- Data structures
- Modules
- Classes
- Frameworks

As mentioned earlier, the atomic units of composition are functions and data structures. The optimal functions for abstraction are pure. They share modular characteristics with functions from math. In math, a function given the same inputs will always return the same output. Functions can be seen as relations between inputs and outputs.

Given any input A, a function f will produce B as output. f defines a relationship between A and B:

$$f : A \rightarrow B \tag{11}$$

Likewise, we could define another function, g, which establishes a relationship between B and C:

$$g : B \rightarrow C \tag{12}$$

This implies another function h which defines a relationship directly from A to C:

$$h : A \rightarrow C \tag{13}$$

Useful abstractions simplify by hiding structure; the same way h reduces $A \rightarrow B \rightarrow C$ down to $A \rightarrow C$ (as shown in figure 45).

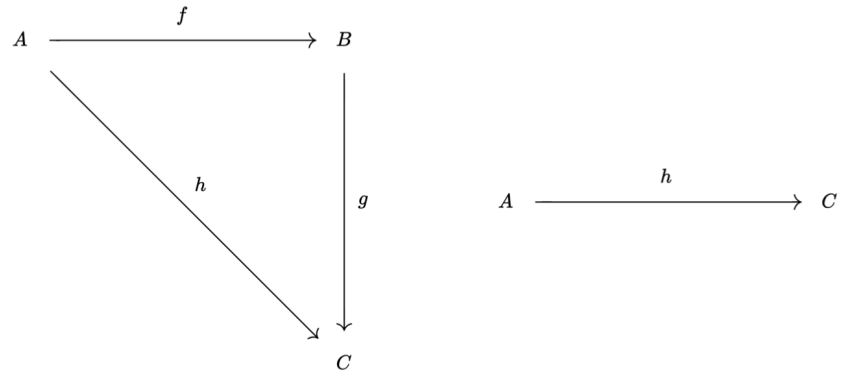


Figure 45: Function composition diagram

Those relationships form the structure of the *problem space*, and the way you compose functions in your application forms the structure of your application (Elliott, 2018). Features are, in fact, domain abstractions that characterise the problem space. In which we can map to the term *problem space* in SPL theory as the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products (Apel et al., 2013). In contrast, as discussed earlier, the *solution space* represents the vendor’s and developer’s perspectives. The *relationships* that form the structure of the problem space can be mapped to the solution space in SPL theory. Characterised by the terminology of the developer, such as names of functions, classes, and program parameters. Apel et al. states that “*the solution space covers the design, implementation, and validation and verification of features and their combinations in suitable ways to facilitate systematic ways*” (Apel et al., 2013).

Distinctions between domain and application engineering as well as problem and solution space leave us with four clusters of tasks in product-line development:

- Domain analysis
- Requirement analysis
- Domain implementation
- Product derivation

Domain implementation is the process of developing reusable artefacts that correspond to the features identified in domain analysis. Components help to break down complex problems that are easier to solve in isolation, so you can compose them in various ways to derive products according to requirement analysis. The component composition creates pipelines that application data flows through. Input added in the first stage of the pipeline is output at the last stage of the pipeline, transformed. Values will go through each component (or function, in function composition) like a stage in an assembly line, transforming the value before it is passed to the next function in the pipeline. For that to work, each stage of the pipeline must be expecting the data type the previous stage returned.

Alan Kay coined the term "object-oriented programming" (hereby referred to as OOP), and his idea was to use encapsulated mini-computers in software which could communicate via message passing rather than direct data sharing (Elliott, 2018). This was to stop having to break programs into separate "data structures" and "procedures". In an email exchange, Alan Kay addressed "OOP" in Smalltalk (Ram, 2003):

"OOP to me, means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things." - Alan Kay (Ram, 2003).

Which means essential ingredients of OOP are *message passing*, *encapsulation*, and *dynamic binding*. The state can be encapsulated by isolating other objects from local state changes. To affect another object's state is by sending it a message asking it to change. Thus, changes are controlled at local rather than being exposed to shared access. The message sender is only loosely coupled to the message receiver through the message API. Objects are decoupled from each other.

Furthermore, Alan Kay states that "*the whole point of OOP is not to have to worry about what is inside an object. Objects made on different machines and with different languages should be able to talk to each other*" (Elliott, 2018).

In a composition-based approach to SPL, the components don't need to know about the details of the other parts of the system. Only their individual, modular concerns. The system components are recomposable and decomposable and interoperate by standardised interfaces. The composition code is the *glue code* of the product derivation. As long as the interface is satisfied, decomposing and recomposing can even be achieved at run-time. And thus, enabling the

ultimate goal of feature-oriented product lines which is automatic product derivation. Besides, composition-based approach prevents monolithic systems, where parts can not be removed or changed without understanding or changing many other parts. These systems become a dense mass that is hard to learn, port, and maintain. It is important to note that all software development is composition. The distinction is between brittle and complicated ways to combine parts, or easy, flexible ways. Some forms of composition form very tight relations between components, and others form loose coupling. The fundamental principle is to identify the parts that are the same and build an abstraction that allows to only supply the parts that differ. This can adequately be mapped to the SPL terms of *commonality* and *variability*. *Product-specific* would, in this case, be the unique parts.

Summary

The idea of feature orientation is to organise and structure the entire software product line process and software artefacts involved, as features. Features specify and communicate commonalities and differences of the products between stakeholders, and to guide variation, reuse, and structure across all phases of the software life cycle.

9.2 Variability mechanisms

In this thesis, we have explored *language-based* variability mechanisms as opposed to *tool-based* variability mechanisms because the tool-based approaches use one or more external tools to implement or represent features in code to control the product derivation process. Language-based approaches are a better match when building applications with ML capabilities. The tool-market is not that useful for deriving product lines with ML code because products are limited and require substantial effort to use. Language-based approaches also allow us to more naturally view our research questions from first-principles by using mechanisms provided by a host of programming languages to derive features (and products).

9.2.1 Components and services

In our thesis, we have mainly focused on a classic language-based implementation approach: components and services (including the notion of a web service).

The key idea of a component is to form a modular, reusable unit. A component provides its functionality through an interface, seeing its internal implementation encapsulated (assuming a black-box component). Also, a class can be seen as a component that can be reused in

many applications. To reuse components, they have to be *composed* with other components in different combinations. By using components, it is attainable to implement a product line by encapsulating changing parts and hiding their internals (i.e., the concept of *information hiding*). Parts can then be removed and exchanged easily. Domain analysis is necessary to identify and separate the parts that differ between the products of the product line.

As a consequence of the proponent component-based approach, developers can implement and deploy components independently, and compose component from different sources. *Building markets* for ML algorithms and services are emerging: thus, developers can decide whether to implement their own components or whether to buy and reuse third-party components. Building markets open a new perspective: Companies with limited resources and knowledge can build applications with ML capabilities without needing to have a lot of dedicated expertise in the area. Components facilitate a best-of-breed approach, in which developers can decide, for each subsystem, whether to purchase a component from the market or to implement the functionality themselves.

Reusing components from markets has a fallacy, and that is components are plug-and-play compatible only if their interfaces are designed to existing standards. Architectural assumptions will mismatch unless components are planned together. Addressing these incompatible components will require a significant engineering effort to compose. Hence, we can establish that **a component is independent of a specific application or product line**. As an example, a component written in JavaScript can be packaged and distributed with npm. Developers can reuse it when implementing the feature for other applications. They would, however, have to write custom code to connect the implementation with the component, for example, additional code to call component methods. To reuse the component, only the public interface is of interest — implementation details do not have to be accessible.

Services

Services are a special form of software components. A service — likewise to a component — encapsulate functionality behind an interface. Services may take the form of a web service; thus, being reachable over a standardised Internet protocol and may run on remote servers. Moreover, services written in different languages can exchange messages because the communication between services is standardised. Through the use of service registries, even the lookup of service can occur at run-time. Services greatly simplify composition through standardisation,

interoperability and distribution.

Composing components

In the early era of product-line engineering, notably, the most common strategy was to develop product lines by constructing and composing reusable components. In the perspective of product lines with ML components that require extensive manual effort and has product-specific requirements to function accurately; components cannot be composed automatically. Component-based software engineering holds a more desirable approach, in which building units of functionality is the goal—fully automating the product delivery with plug-compatible components are not typically a priority. The design-for-change mindset with the component-based strategy is to exchange a component by another one with the same interface. This vastly simplifies A/B testing of recommendation systems or ML models in general.

In the component-based approach compared to a feature-oriented approach; manual developer intervention is required as no generator would build a product for a given *feature selection* (hence software product lines, throughout this section). When pursuing this strategy, to derive a product for a given feature selection during application engineering, a developer selects suitable components and then manually writes glue code to connect components for every product individually. Building product lines with components are suitable if feature selection is performed by developers (not customers) with a limited set of products. Developers can, in this case, construct new products from reusable parts and perform customisations beyond what was preplanned as a product line feature. Although manual effort (almost unavoidable anyways, see chapter 7.2) during application engineering increases the cost per product, implementing product lines with components, do not change the overall expected benefits. Automated product derivation is essential for effective, large-scale product-line development; but, it is not especially desirable or attainable when working with ML components.

Tensions between reuse and use

There is a trade-off between reuse and use: Developers need to provide a component small enough to be reused in many contexts but balance the need of being large enough to provide useful functionality. The smaller, reusable components can be combined flexibly, but, add a lot of overhead of using the component. Whilst, a large component that provides plenty of functionality is easy to integrate and use, might only fit a few applications. Maximising reuse minimises use (Szyperski, 1997, Chap.4). With small components, more programming

(or glueing) is left to connect the components with the base code and each other. In some extreme cases — with very small components — little remains to reuse and to hide behind their interfaces. Unsuitable component size will likely limit the success of component reuse. But, without knowing when and how a component will be reused, it is hard to decide on the suitable component size.

In product line development, domain analysis is used to solve the issue in how to size a component to balance use and reuse. During domain analysis, a domain expert investigates potential products within the scope of the product line. Having this information upfront about functionality reuse will ease the process preplanning reuse systematically by better determining the component size and coordinating architectural assumptions. When functionality is only used in a few products, it should be separated into its own components. In contrast, we know certain functionality is always used together; we can combine it in the same component to make it easier to use. Thus, domain analysis helps decide how to divide code into components.

9.2.2 Parameters

An advantage of passing configuration parameters as method arguments (in contrast to using global variables) is that different parts of the control flow can have different configurations.

Some drawbacks with this approach are the issue of dead-code. There are options to build compilers to deal with this issue, by removing corresponding conditional statements and their bodies if it is known at compile-time that the parameter is always deactivated. However, when to remove entire classes or methods is less obvious. Beyond manual dead-code optimisations, compilers to optimise this process is far from the mainstream or easy to use yet.

The parameter approach could use a type system (i.e., TypeScript) for checking type compatibility to statically guarantee invariants on feature selections (thus, activating features that are not compatible with each other).

Feature dependencies should be checked when the parameters are configured. Dependencies between features are rarely checked systematically when using parameters. This can lead to implementations that are error-prone and have poor code quality. Propagating method arguments may lead to methods with many parameters or unused parameters (considered as code smell by Al Mamun et al. (2019)), and can lead to undisciplined ad-hoc implementations that are hard to maintain and debug. The optimal approach is to pass a single *configuration object* that

encapsulates multiple configuration options because global parameters violate the separation of concerns and potentially breach information-hiding interfaces.

Summary

The key idea of a component is to form a modular, reusable unit. A component provides its functionality through an interface, seeing its internal implementation encapsulated (assuming a black-box component). Domain analysis is necessary to identify and separate the parts that differ between the products of the product line. Components facilitate a best-of-breed approach, in which developers can decide, for each subsystem, whether to purchase a component from the market or to implement the functionality themselves. Furthermore, it is important to establish that a component is independent of a specific application or product line. In the component-based approach compared to a feature-oriented approach; manual developer intervention is required as no generator would build a product for a given feature selection. Although manual effort during application engineering increases the cost per product, implementing product lines with components, do not change the overall expected benefits.

9.3 Version-control systems in software product lines

Version-control systems are used to track changes in source code and other development artefacts to track changes in source code and other development artefacts to facilitate collaborative development. Two popular options are *Subversion* and *git*. We have looked into using version-control systems for product line development as a part of software configuration management. Changes can be tracked in two dimensions: (i) *variation over time* (revisions of source code) and, (ii) *variations in space* (variants of the same file). In the case of variants of the same files, they can be changed independently. They exist in parallel as changes in one branch do not affect the files in other branches. Each of these branches can be terminated when it is no longer needed.

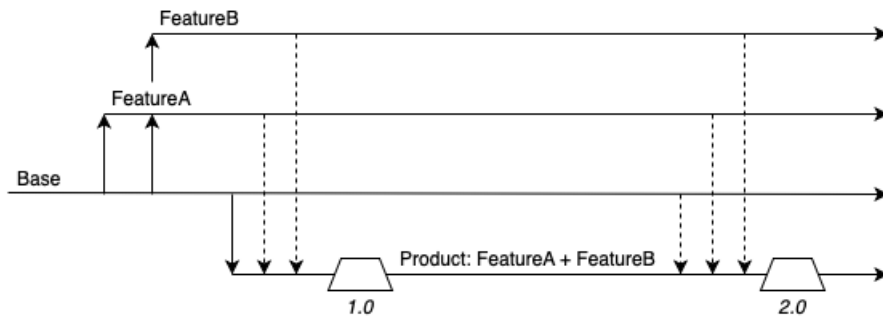


Figure 46: Merging feature branches

A product line may be implemented by merging branches in a version control system. As shown in figure 46, each feature in a distinct branch could create products by merging corresponding feature branches. Instead of merging feature branches to derive a product, each customer could get their own branch with custom modifications, as shown in figure 47. With the approach in figure 47, recent developments and bug fixes will be merged from the main branch into customer branches. While it is possible to implement product lines by merging branches in a version control system, problems will accumulate, as the product line evolves. The use of version-control systems should be restricted to revision control.

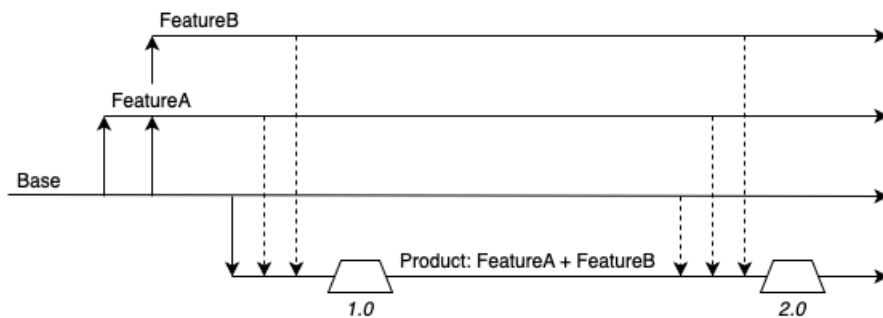


Figure 47: Merging feature branches with custom modifications

For purely revision control, a common approach should be considered, as shown in figure 48.

- **Development branch:** The central branch of the project.
- **Release branches:** For each planned release, a separate branch is created. These separate branches should be used to avoid daily interference from the main development branch to focus on getting a release stable.

- **Feature branches:** These branches should be used for larger blocks of functionality or develop a feature of a future release.
- **Bug fixes:** Bug fixes can be developed in separate branches and can be merged into one or multiple other branches.

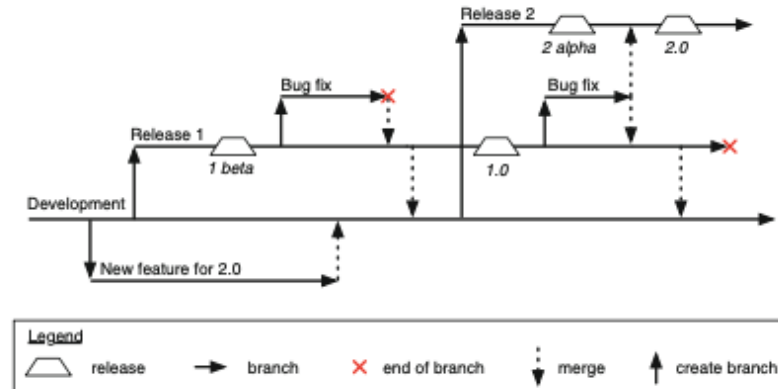


Figure 48: Revision control. Figure taken from Apel et al. (2013).

Like version-control systems, *build systems* are a part of software configuration management. Build systems support multiple build targets, dependency management, avoid unnecessary recompilation with incremental builds, download and updating required libraries and tools, and to create build reports. There are many tools to help the orchestration of version-control mechanisms and, maintaining a uniform application to source code and noncode artefacts. When developing a software product line with a component-based approach, utilising a *monorepo* (a single repository) for code sharing makes sense. Tools such as *Lerna* which optimises workflow around managing multi-package repositories allows for maintaining modules in the same repository. Having all modules in the same repo makes it easy to coordinate changes across the modules. There is just a single lint, build, test and release process and only a single place to report issues. This will allow for an easier development environment setup and better feature tracking. Besides, other benefits include that tests can run together across modules to find bugs that touch multiple modules more easily. Probably, the main benefit is the ability to split up large codebases into separate independently versioned packages, which is useful for code sharing and in line with a component-based product line approach. Using Lerna, our directory structure looks something like this:

```
spl/  
  packages/  
    core/  
    customerA/  
    customerB/  
    customerC/  
  package.json  
  lerna.json
```

In a multi-repo approach, all of our packages would either be library targets or dependencies for our library targets:

```
@spl/core  
@spl/customerA  
@spl/customerB  
@spl/customerC
```

With the multi-repo approach, given that most of the library logic resides in the `@spl/core` package, every change that occurs in the `core` package and all other packages had to be updated to point to the new version. Should a bug occur in `customerA`, in many cases, the change will have to be fixed in `core`, but the update will then have to be reflected in all other targets. There is a good chance with the multi-repo approach with many customers; a target will be missed. In other words, a change in the library will result in changes to multiple packages. It is revealing a tedious process to update all targets and being prone to a developer making a mistake. However, there are usually build lint, test, and build processes to help and avoid errors from occurring. These lint, common tests and build processes could in sporadic cases differ but are highly unlikely, and thus, revealing the biggest drawback of having a multi-repo. While allowing the sharing of common packages and commands across all applications are extremely beneficial in a monorepo — the adoption of any solution do not have to be binary. At scale, how well an organisation does with code sharing, collaboration, thigh coupling, etc. is a direct result of engineering culture and leadership and whether the used solution does not transform these values in its own right.

Summary

While it is possible to implement product lines by merging branches in a version control system,

problems will accumulate, as the product line evolves. The use of version-control systems should be restricted to revision control. Tools such as Lerna which optimises workflow around managing multi-package repositories allow for maintaining modules in the same repository. Having all modules in the same repo makes it easy to coordinate changes across the modules.

9.4 Recommender Systems in SPL

Recommender systems can predict whether a particular user would prefer an item or not based on the user's profile. They are information filtering systems that handle the problem of information overload by filtering information of generated information according to user's preferences, interest, or observed behaviour about an item.

Recommender systems support users by allowing them to move beyond catalogue searches. Our research survey indicates that respondents believe that RS enables them to find relevant information regarding their preferences more efficiently. Furthermore, respondents indicated that poor recommendations was annoying and could make them lose trust in the recommendation system. Hence, it is paramount to use efficient and accurate recommendation techniques within a system that will provide relevant and dependable recommendations for the users.

Recommendation systems shall assist and augment the social process of using recommendations of others to make choices when there is little personal knowledge or experience of alternatives. There are various approaches for building recommendation systems which utilise either collaborative filtering, content-based filtering or hybrid filtering. Despite the success and adoption of collaborative filtering and content-based filtering techniques, there are problems associated with either technique. Collaborative techniques suffer from problems such as cold-start, sparsity and scalability problems and the content-based filtering techniques are limited content analysis, over-specialisation and sparsity of data.

These techniques can form a hybrid-approach by combining two or more filtering approaches to harness their strengths and reducing their weaknesses. Hybrid approaches can be classified based on their operations into a weighed hybrid, mixed hybrid, switching hybrid, feature-combination hybrid, cascade hybrid, feature-augmented hybrid and meta-level hybrid. Inspecting each hybrid approach is out of the scope of this thesis. Nevertheless, we will discuss a way to deal with software product line feature combinations containing machine learning components.

9.4.1 ML and SPL

More companies are eager to develop applications with ML capabilities, but the ML models end up in internal use. There is a gap to practical deployment developers struggles to bridge. McCaw states that “*only 22 percent of companies using machine learning using machine learning have successfully deployed a model*” (McCaw, 2019). As mentioned earlier we believe a problem should be solved at the lowest level that it can be solved at. Bottom-up allows for self-organised behaviour in a way that top-down doesn’t. We will look at the root causes. Before anything else, note that there is a fundamental difference between traditional software and machine learning: ML is more than code; it is code plus data.

The ML artefact that ends up in production is created by applying an algorithm to a mass of training data, which will affect the behaviour of the model in production. The behaviour of the model will also be affected by the input data that it receives at prediction time, which can not be known in advance. This is shown in figure 49.

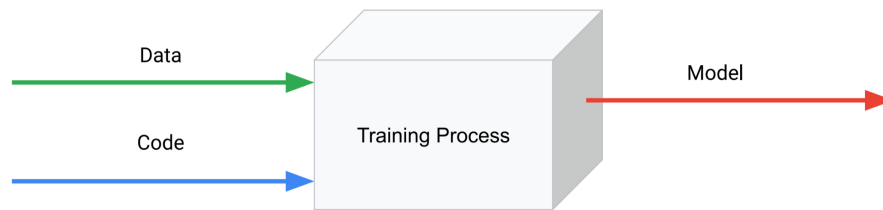


Figure 49: ML consists of code and data. Figure taken from Breuel (2020).

Code can be developed in a controlled development environment, but data never stops changing, and it hard to control how it will change. Data and code evolve independently. Then the dilemma of a machine learning process is to bridge the data and the code in a controlled way. Some of the causes for ML not being deployed in the production lies with lack of reproducibility and, slow and inconsistent deployment.

9.4.2 Phases of recommendation process

One of the core concepts is the data pipeline, shown in figure 50. ML models will always require a type of data transformation. Using data pipelines provides advantages in code reuse, management, run-time visibility and scalability. It is possible to argue that ML training also can be considered as a data transformation, and hence, can benefit from including the specific

ML steps into the data pipeline becoming an ML pipeline as we have done with our CSV loader. The ML Pipeline is independent of specific data instances and consequently becomes a pure code artefact. This means it is possible to break it down into components, track its versions with source control and deploy with a regular CI/CD pipeline.

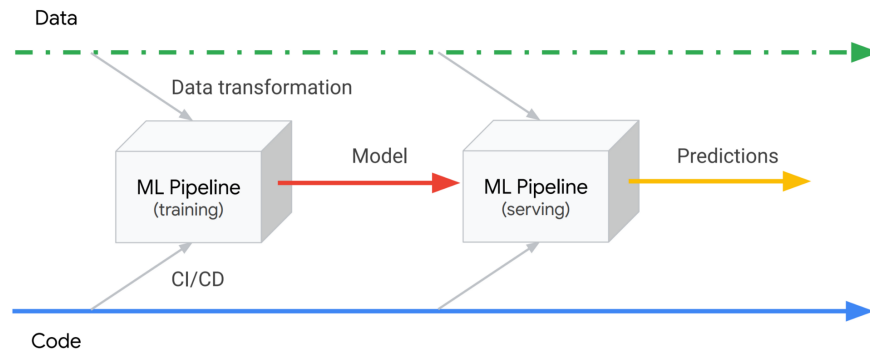


Figure 50: ML Pipelines connect data and code to produce models and predictions. Figure taken from Breuel (2020).

As discussed earlier, the recommendation process is sub-divided into three phases: (i) information collection phase, (ii) learning phase, and (iii) prediction recommendation phase.

Information collection phase

During the information collection phase, relevant information about users is collected to generate a user profile or model for the prediction tasks. Information includes a user's attribute, behaviours or content of the resources the user accesses. As discussed in chapter 3.3, recommender systems rely on different types of input, such as inferred user preferences through user behaviour observation (*implicit feedback*) and explicit input by users (*explicit feedback*). Hybrid feedback can be obtained through the combination of both explicit and implicit feedback. In a fully-fledged E-learning platform, a user profile will be a collection of personal information associated with a specific user. This information includes interaction with the system, learning styles, interest, preferences, cognitive skills and intellectual abilities. The success of any recommendation system depends on its ability to represent a user's current interests. Information to build up a model of the user is retrieved from the user profile. We can say that the user profile describes a simple user model. Accurate models are by those means key to obtaining relevant and precise recommendations from any prediction technique.

The information collection phase will mostly be a manual process requiring a lot of glue code to construct the moving product-specific parts to build a data pipeline (or integrated into ML pipeline) feeding the recommender system. However, some components can easily be generalised and support variability, such as a system interface that prompts the user to provide ratings for items to construct and improve the model. Variability in this sense would be the contents of the form submission. The rating accuracy of recommendation depends on the quantity of ratings provided by the user. Nonetheless, the component itself can easily be generalised for the software product line. On a side note, explicit feedback requires more effort from the user but also provides transparency into the recommendation process that results in a somewhat higher perceived recommendation quality. Components to automatically infer the user's preferences can also be constructed as monitoring components to record actions of the users such as navigation history, time spent on courses and links followed by the user. Implicit and explicit feedback can be composed to minimise their weaknesses and create a performing system. For example, explicit feedback can only be given when some implicit criteria are met.

Learning phase

The learning phase applies a learning algorithm to filter and apply the user's features from the feedback gathered.

In our approach, we used a model-based technique in an effort to provide useful recommendations to the users. The model-based technique employs previous rating to learn a model in order to improve the performance of the collaborative filtering technique. We decided on this approach because of the model-based techniques can recommend a set of items quickly because they use a pre-computed model. Model-based techniques also alleviate the sparsity problems associated with recommendation problems. There are many facets of model-based techniques that use the user-item matrix to identify relationships between items. We specifically used a model-based regression technique as it is a powerful process for analysing associative relationships between the dependent variable and one or multiple independent variables.

The models built in this phase can be composed into components and reused as seen fit. This is ideal considering we would want to reuse as much as possible.

Prediction / Recommendation phase

In this third phase, it recommends or predicts what kind of items the user may prefer.

This is the top-N list that shall be passed to the components responsible for serving the result to the display layer. Ideally, after the top-N cut is predicted, it should be filtered through a stoplist. The stoplist is responsible for removing any prediction that may be offensive, inappropriate, etc. Our respondents answered that they had been offended by recommendations before, and a few of them deemed the content inappropriate.

9.4.3 Composing components in an SPL ML pipeline

Smaller components are easier to reuse. They do less and make fewer assumptions. They are thus having the benefit of being easier to understand and maintain. Design is about pulling things apart. It can always be put back together. Components should have a few related or ideally single responsibilities. Then we will find that they are easy to organise around concepts.

Sources of coupling dependencies are important to understand in order to build independent components. According to Nodejs best practices, this is roughly the order of how tight the coupling is (Goldberg et al., 2020):

- Class inheritance (coupling is multiplied by each layer or inheritance and each descendant class)
- Global variables
- Other mutable global states (browser DOM, shared storage, network, etc...)
- Module imports with side-effects
- Implicit dependencies from compositions
- Control parameters
- Mutable parameters

Loose coupling:

- Module imports without side-effects
- Message passing / publish-subscribe
- Immutable parameters

Most sources to reduce coupling are mechanisms originally designed to reduce coupling. Smaller problem solutions need to integrate and communicate to be able to recompose into a complete solution. Sources of tight coupling should be avoided as much as practically possible.

9.4.4 SPL feature interaction

When features (or components) are developed independently, an issue can occur where it is difficult to predict their mutual interactions combined. Inadvertent behaviour occurs due to missing coordination when combined with other features. For a software product line to be successful: features, have to be understood by their interactions.

Feature interactions can be inadvertent or undesired in the sense that they are hard to predict when planning and implementing features in isolation. This especially holds true to for ML components that need to interact. These feature interactions must, in many cases, to be planned in advance. Besides that, each feature has its own code; there might be code that does not belong to a single feature, but to a combination. This may harm feature modularity.

Along with the feature interactions, there are also features that can be optional such as a stoplist or components for monitoring. The problem of optional features is an implementation-level instance of feature interaction. The optional-feature problem is concerned with incorrect implementations of variability that reduce the intended variability or have negative effects such as non-modular code (Apel et al., 2013).

The optional-feature problem happens because of a misalignment between variability specified in the feature model and the variability provided by the specific implementation. Essentially, when coordination code is hardwired inside a feature such as our `loadcsv`. Down the line, `loadcsv` might have problems in the form of type errors because of mismatches. Coordination or glue code should not be implemented in a way that it impairs variability. Of course, our `loadcsv` can be refactored at later stages, which we will discuss later, should it pose problems for variability.

Knowing how two features interact — a strategy to deal with the interaction which does not introduce the optional-feature problem should be devised. All combinations of feature interactions can be described to need coordination code to use two independent features together. Our problem-solving strategy for the issue should: (i) not reduce variability merely due to implementation issues, (ii) not require overwhelming implementation effort, (iii) not decrease the performance of products compared to an individual implementation of each product, and (iv)

should not reduce code quality, making the product line harder to maintain.

Change the feature model

The simplest solution would be to add constraints restricting the feature model to exclude problematic feature selections. This will limit variability related to what should be valid products in the domain. None implementation changes are needed — instead of adding glue code or reimplementing features — thus, does not affect performance or code quality. There is a trade-off to be considered in whether the reduced variability can be acceptable or will have damaging effects on the strategic value of the product line.

Multiple implementations

The strategy is based on providing multiple implementations of a feature, one with and without glue code. This does not align well with the prime benefit of product line development, code reuse. In fact, it encourages code replication instead as features must be implemented multiple times — one for each feature combination. This leads to scalability issues if a feature interacts with multiple other features.

Move code

We could instead implement feature code in either of the features or create a third feature to which both refer. Drawbacks with this approach include: (i) glue code does not belong to a single feature, but combinations of features, (ii) does not acknowledge the principle of separation of concerns, and (iii) unnecessary code is included in some products.

Distinct module for coordination code

Coordination code could be separated into a *module for coordination* code and composed with the implementation of both features. It is essentially scaling for higher-order interactions with modules for coordination code. There are drawbacks to this way of handling interactions as the number of derivatives may grow out of control in cases where many interactions exist. This may lead to a large number of additional but potentially small modules that can be overwhelming and hard to understand in isolation. Creating distinct modules for coordination code may be elegant but can increase complexity.

Comparison of solutions

Above strategies except for changes to the feature model support full variability. Creating distinct modules or creating multiple implementations per features can cause a lot of overhead.

The multiple implementation strategy, will also cause code replication which is something to avoid. In practice, these strategies are mixed and matched according to needs.

In regards to recommender systems, we can discuss interactions between the algorithms in hybrid filtering techniques. These techniques combine different recommendation techniques to gain better system optimisation to avoid limitations and problems of pure recommendation systems. The idea is that a combination of algorithms will provide more accurate and useful recommendations than a single algorithm. Other algorithms can solve the disadvantages of a single algorithm. Isinkaye et al. states that *"the combination of approaches can be done in any of the following ways: separate implementation of algorithms and combining the result, utilising some content-based filtering in a collaborative approach, utilising some collaborative filtering in content-based approach, creating a unified recommendation system that brings together both approaches"* (Isinkaye et al., 2015).

Summary

Data and code evolve independently. Then the dilemma of a machine learning process is to bridge the data and the code in a controlled way. When features (or components) are developed independently, an issue can occur where it is difficult to predict their mutual interactions combined. Inadvertent behaviour occurs due to missing coordination when combined with other features.

9.5 Summary

The process of software development is to break down large problems into smaller problems by building smaller components that solve those smaller problems and then compose these components to form a complete application. Software systems should be constructed from reusable components because then the systems can be tailored to the requirements of the customers; where the customer can select from an ample space of configuration options. This provides a form of mass customisation by constructing individual solutions based on a portfolio of reusable software components. It introduces individualism into software production but is still able to retain the benefits of mass production. The need for individualism arises from the different software requirements regarding functionality, nonfunctional properties, and target platforms. The idea of feature orientation is to organise and structure the entire software product line process and software artefacts involved, as features. Features specify and communicate commonalities and differences of the products between stakeholders, and to guide variation, reuse,

and structure across all phases of the software life cycle. We have utilised first-principles toward composing software to abstract and reuse the concepts to shape a foundation for composing software product lines.

Consequently, we have delved into functional thinking and explained why we believe the composition-based approach is the ablest path to compose software product lines with machine learning components. This approach has several strong points including a well-known and established implementation technique, reuse within and beyond the product line, reuse of third-party implementations, reuse in distributed environments, uniformly applicable to many languages, and with separations of concerns (information hiding). We have also revealed a few weak-points with this approach by having no automated product derivation, thus requiring glue code and that pre-planning is necessary to size components. We believe the benefits of the architectural structure and method outweigh the drawbacks because when working with machine learning, we don't want to add more complexity into the problem at hand. The strategy of implementation can almost be summed up entirely in just one sentence: *The system components are re-composable and de-composable and interoperate by standardised interfaces*, which is principled on the original idea Alan Kay had for "OOP" in its purest form. Composition-based software engineering does not regard automating product delivery with plug-compatible components a priority. While it may be attainable to automate the entire process, it is not necessarily desirable. As discussed, producing accurate predictions that serves the customers and end-users needs requires extensive manual effort. We would not want to encourage a human-zero mindset but rather a human plus AI mindset. Viewing algorithms, tech and, people and process as three distinct parts; algorithms should account for 10%, the tech should account for 20%, and people and process should be responsible for 70% of the process to maximise the real outcome. Winning organisations will invest in human knowledge, not just AI and data. A component-based approach aligns well with this goal by having a *design-for-change* mindset as during development components can be exchanged as long as they adhere to the same interface. This substantially simplifies multiple aspects of software development with ML capabilities as A/B testing can be achieved in a breeze (doing partial rollouts, reverting changes, etc.).

For a software product line to be successful: features, have to be understood by their interactions. Understanding features by interaction can be hard when planning and implementing in isolation. Without proper design, interactions can lead to negligent behaviour. In ML components, more often than not, code belongs to a combination of features. This will harm our goal of feature

modularity. Thus, we may take actions to address this issue and understand it in connection with the concept of optional-feature problem, which is concerned with incorrect implementations of variability that reduce the intended variability or have adverse effects such as non-modular code.

Strategies to mitigate this problem includes:

- Changing the feature model,
- having multiple implementations,
- move code,
- or utilise a distinct module for coordination code.

There are benefits and drawbacks with each of these methods, but in practice, they should be mixed and matched to provide one coherent functioning strategy.

10 Discussion

In this chapter, we present our theoretical contributions through answering the research questions, then we offer practical contributions discovered through our research, then we discuss validity threats to our research process and evaluation methods, and finally, we present related work.

10.1 Theoretical contributions

The theoretical contributions we have made through our research in this thesis is presented in the following sections by answering each research question.

RQ1: How possible is it to create machine learning components that work for multiple products in a software product line?

The essence of development is composition. The essence of software design is problem decomposition. You can not assemble what you can not take apart. In our effort to figure out whether it is possible to create components with machine learning capabilities for multiple products in a software product line; we researched several engineering methodologies and settled on a composition-based approach because of our statement above: *the essence of development is composition*.

Software should strive for simplicity and, be aided by good mental models and abstractions. *Software architecture* has come to mean how parts of a system work together and communicate with each other. The most straightforward architecture is the “pipe-and-filter”, in which an operator send output from one program (filter) into another. A pipeline based on this notion is usually used to describe the overall process of the tasks, that has to be done in succession when building machine learning applications. Whether it being a *data pipeline* or an *ML pipeline*, we believe its internal complexities can be componentised according to our findings in chapter 9.2.1. Architecture then describes the system properties, how components cooperate to meet the overall requirements. Interface matching is a system property, but the descriptions used to check a match are at the component level (i.e., component model), and the system semantics applies to the component connections themselves, describing and constraining the interfaces.

We are not trying to impose any strict architectural decisions to the software product line other than it should consist of composable components. Then the only restrictions would be of how ML components need to interact, to yield a useful result. With a pipeline: a system is a single

line of components, each “provides” joined to the following “requires”. This can, in essence, yield a web-service, consisting of multiple minor components. Two or more abstract approaches are useful for software product line engineering with machine learning components: components as building blocks (lower level) to describe the system architecture (higher-level) of the system. Then there is room for abstracting as you would like with additional middle-layers to adapt for forming even larger, more complex systems according to our findings in chapter 9.1. As mentioned, the understanding of the system behaviour will come from the interactions and relationships between the component models observed at different levels. Successful abstraction means that the result is a set of independently useful and recomposable components. The ideal approach strikes the right balance in abstraction between too much abstraction, where insights would not be useful; and, too little abstraction, where little will be learned that come from observing reality.

Our point is that nothing is hindering creating reusable machine learning components as long as parts can be divided into smaller standardised components. Thus, it becomes more of a question of how reusable are they?

RQ2: How reusable are machine learning components in a software product line?

A composition-based software engineering approach is desirable because building units of functionality is the goal. The attempt to automate the product delivery with plug-compatible components entirely is not typically a priority. This aligns well with our knowledge of how machine learning models behave and their fragility. Small changes to code or data can drastically alter how the model performs. When we remove the trait of some software product lines that they shall be fully automated, the question is more of *whether the component* itself can be reused for other applications. As we have seen is that code can be reused, it is merely the fact that the data that are accompanying the component that cannot unless it is a component that supports *transfer learning*. There is no catch-all optimisation that works on a single component that will work on others, and hence, lies the problem with reusing machine learning components. We could optimise the component to the best possible solution for a given configuration. The configuration would though change depending on customer and context; thus, we would rather want to look at an adaptive solution since we will do better than a solution that is optimal sometimes and creates havoc at other times. Methods for optimisation can be effective for problems where the domain is reasonably static. Problems occur when the domain changes, to cope with these changes, solutions should be constantly approached actively.

Although manual effort during application engineering increases the cost on a per-product basis, it does, however, not change the overall expected benefits according to findings in chapter 9.2.1. There is still the benefit of having the possibility to build new systems using the components developed for their older brothers. The factors that might cripple reuse are things that require data to be handled in a certain way (that cannot be defined as a variability) and how the components interact. Inadvertent behaviour occurs due to missing coordination between interacting components. In order to avoid that, machine learning components must, more often than not, be preplanned in advance to ensure their compatibility. Besides that, there might be components that have code that does not belong to a single component, but to a combination given the linear architecture of data- and ml pipelines. This may harm modularity.

Building software product lines with machine learning components are suitable should feature selection (feature selection in terms of SPL) be performed by developers (not customers) with a limited set of products. Machine learning components add restrictions to how a software product line can be architected. With the separation of concerns between code and data, components may be reused according to findings in chapter 9.4.

RQ3: How feasible is it to create and consume reusable machine learning models in a software product line?

Creating and consuming reusable machine learning models are feasible as we have shown earlier in our theoretical discussion about the composition-based approach. Nevertheless, this approach hinges on the success and viability of the responsiveness between the interfaces of the components. In our understanding of the topic and from our evaluation, it is both feasible to create and consume given the right time and resources to implement a proper solution. However, this thesis has had recommender systems and general web-development in mind and has been implemented with JavaScript, and its surrounding shared libraries. We are not familiar with the language designs in, for example, programming languages such as R. Thus, we find it hard to conceive a catch-all solution to API design that apply to ML components. However, an expert interviewed noted that in a few cases it is beneficial to include default options that can be overridden by values passed as parameters as it is easier to get the ML component quickly up-and-running and then time can be spent adapting the component interaction (see chapter 8.3).

As an afterthought of the interviews, we understood that in some scenarios a composition-based approach might lead to sub-system optimisation, in which, developers tasked with developing

a component spends too much time just optimising that particular component. This can be unbeneficial in companies where the same developers handle both application- and domain engineering because the software product line is best of, understood as a whole. There is a fine line between too little and too much optimisation. It is also important to call into question the constraints of a component. Developers might design around the constraints of that component, instead of questioning them. It is important as developers to question the constraints of the component because there is plenty of research that indicates that product errors reflect organisational errors. In this thesis, we focused mostly on the *Business, Code* and *Architecture* in the BAPO model, and left out the *Organisational* part. This is a limitation of our study but which we made as we worked with a small company, but other researchers should further investigate this aspect.

RQ4: How can you support a software product line evolution containing machine learning components?

We have discussed a *design-for-change* mindset that is key to support the evolution of software product lines containing machine learning components. Along with the standard approaches to refactoring a software product line, we believe it is essential to replace components when making changes, and we believe this will be beneficial in the long term (see chapter 6.6). Let us assume you want to adjust or add functionality to a component. The entire component should be replaced by a newer version containing the changes, and properly A/B tested because of fragility in ML components (described in chapter 7.2.4). A small change in code or input data can cause irreversible effects on the output (Ghorbani and Zou, 2017). In the typical training process, training data is collected, and a model is trained to recognise patterns in this data, in order to make predictions about new data. In other words, ML systems learn about the data they are trained on, and learning algorithms are constructed to generalise from that data. The end result is that the models can be fragile and unpredictable. Thus, we would not want to make changes and deploy our models directly without proper testing to ensure relevant and bias-free results.

The overall process of building the software product line can benefit from using a reactive approach (as described in chapter 9.1.2). During time and evolution, all taxonomies are eventually wrong for new use-cases. Therefore, being agile and adaptable will help to cope with change in demands. One expert also stated that dividing the development into a domain- and application-specific teams can give better structure on areas of responsibilities, and make the process more

efficient according to findings in 8.4. A reactive approach is many ways an ideal path to success when building software product lines because the developer cannot *a priori* conceive of all possible configurations, purposes, or problems the system might be confronted with. That said, components should, to a large extent, be preplanned when having machine learning capabilities.

RQ5: How does a software product line affect the quality of its recommender system?

We will ultimately believe a software product line will aid the speed and effort required to implement components with ML capabilities. Remember, the software product line is more than code; it is also processes and practices in the surrounding organisation. Building a software product line for a recommender system will help to establish repeatable procedures, resources and principles streamlined to ensure a satisfying result. Automatic derivation, however, can be compared to a horse wearing blinders; you might run faster, but will also lose track of your surroundings. According to our survey in chapter 8.5, 54.6% of respondents expressed that they will likely lose trust in a recommender system if it yields bad predictions. Each configuration should, therefore, be optimised for the customer and the context to yield relevant results. Other than the manual labour required to optimise (and adapt) the solution having components to mix-and-match, will ease the process of building the *right* recommender system. Trial and error through partial rollouts with good A/B testing are ultimately how recommender systems are deployed today, and a software product line does not have to change that.

10.2 Practical contributions

We have established a theoretical foundation in the thesis of *how* and *why* a composition-based approach to creating software product lines is ideal when the product line shall contain machine learning components. In practical terms, this implies building reusable components that can be composed to create an ML pipeline. The design-for-change mindset mentioned in chapter 9.2.1 allows for quick replacement of components for testing and improvement. The factor of reusability comes from thinking functional when developing the components by splitting data and code and create suitable interfaces for the components to interact. Overall the task of introducing machine learning components to a software product line, and having to maintain and evolve them might not seem too daunting anymore.

10.3 Ethical Considerations

We have taken precautions to ensure that research is carried out in accordance with good research practices and the University of Oslo's ethical guidelines (Ethical guidelines, 2020). All interviews with the respondents have been conducted through voluntary participation and the respondents have had the right to withdraw from our study at any stage. They have all participated on the basis of informed consent. We have anonymised individuals and organisations participating in the research. Furthermore, any use of offensive or unacceptable language has been avoided in the formulation of the questionnaire and the interviews. All of our communication in relation to the research has been done with honesty and transparency. We have worked to ensure that we avoid biased data findings and we have not used any deception or exaggeration to fulfil the aims and objectives of this thesis. John of Salisbury expressed the metaphor of **dwarfs standing on the shoulders of giants**, in essence, meaning "*discovering the truth by building on previous discoveries*" (Standing on the shoulders of giants, 2020). Acknowledgement of the works of other authors used in any part of the dissertation has been done with the use of Harvard referencing system.

10.4 Threats validity

When conducting an evaluation, it is relevant to assess the threats to the validity of the study and the results. For the study, we have conducted we find this very important. This is because of the number and magnitude of threats that exist when conducting empirical studies (Feldt and Magazinius, 2010). Validity in research is concerned with how the conclusion might be wrong, hence does not correlate with the reality. A specific action used to increase validity by addressing a specific threat is called a *mitigation strategy* (Feldt and Magazinius, 2010).

A number of models and lists of validity threats exist to help to realise and mitigate the threats. Feldt and Magazinius present a list of 7 validity threats that exist when conducting qualitative evaluation methods, shown in table 12.

No.	Validity threat type	Example of typical questions to be answered
V1	Conclusion validity	Does the treatment/change we introduced have a statistically significant effect on the outcome we measure?
V2	Internal validity	Did the treatment/change we introduced cause the effect on the outcome? Can other factors also have had an effect?
V3	Construct validity	Does the treatment correspond to the actual cause we are interested in? Does the outcome correspond to the effect we are interested in?
V4	External validity, Transferability	Is the cause and effect relationship we have shown valid in other situations? Can we generalize our results? Do the results apply in other contexts?
V5	Credibility	Are we confident that the findings are true? Why?
V6	Dependability	Are the findings consistent? Can they be repeated?
V7	Confirmability	Are the findings shaped by the respondents and not by the researcher?

Table 12: Main types of validity threats. Table taken from Feldt and Magazinius (2010), p. 376.

There were several threat validities to our evaluation methods. Finding our test subjects was an example of theoretical sampling because the diversity and number of test objects were limited. This is a threat validity that lowers the credibility (V5), and if we had the time and capacity, we should, therefore, have interviewed more people with a higher diversity in technical competence and roles to mitigate this. We mitigated the *confirmability* (V7) threat by having a less strict interview and allowing for more dialogue with open questions. Having these interviews with other software developers increases the *confirmability* (V7) because the conclusion and findings are based on not only research and reflections from ourselves but from other developers.

The *dependability* (V6) threat was affected in the way that there were only one of us that took notes during the interviews, while the other asked questions. However, we mitigated this threat by discussing the results right after the interview was done, to evaluate the data we had collected

and see if something was missing or was unclear. We also changed who were taking notes every other interview, to avoid biased notes.

We see a few threats to the statistical analysis we did of the survey results. But there is a *credibility* (V5) and *confirmability* (V7) threat in that the participants of the survey had only read a short background at the beginning prior to answering the questions, so their answers may be based on too little knowledge and may be a bit random. Hence their answers can be biased. We mitigated these threats by consolidating with an expert on statistics when formulating the questions.

Based on the number of participants we had on the survey, the *credibility* (V5) threat is lowered. Considering the interview, it was lowered because of the experience the experts we chose had with software engineering. And because of their heir roles, they were highly relevant for our case study, hence increases the *construct validity* (V3). The *dependability* (V6) threat is a threat that could be lowered a lot by conducting more interviews with more experts. But seeing a correlation between the results from all the experts imply that the results are consistent.

Prototyping with a recommender system is a specific approach of ML, but our evaluations are also based on a general and high-level view of having ML components in an SPL. This was to increase the *external validity* (V4), so we would generalize the results outside just the prototype scope. Researching solutions that allow for multiple products to consume the same ML models is to increase the *transferability* (V4), so that our research can be relevant to apply to other companies. We can also offer “lessons learned” and “good practices” when having ML components in general with multiple products sharing functional requirements or functionalities.

Observing while the experts studied our prototype was to evaluate the *internal validity* (V2) of the prototypes. We asked questions related to the prototype afterwards to get results on whether this is a good solution and if our design and implementation of the prototype can be applied for multiple products, hence solving some of the problems introduced in the research questions. We also asked for improvements and changes that could affect how an SPL can interact with the ML components we have made, to see if this affects the results.

Conclusion validity (V1) is a result of knowledge gained through research, development and evaluation. We do not present any new theories, we simply present lessons learned and a process of how to reuse ML components in a product line, based on research and theories already made.

10.5 Related work

We have been researching to see if similar problems have been solved or conducted research on. As we have been reading articles and journals, nothing directly similar has been found. However, we see that Apel et al. (2013) introduces a feature-oriented approach when developing an SPL, they do not; however, focus on implementing ML components into it. Sculley et al. (2015) discuss the maintenance and risks of having ML components and study how ML components affect a larger system; they do not focus on having ML components serving multiple products in an SPL. Bacciu et al. (2015) suggest a machine learning approach when analysing patterns in different products to predict features for new products; they do not focus on reusability of code or components. Camillieri et al. (2016) also focuses on having a machine learning workflow when designing SPLs but focuses more on evolution mechanisms for software; they do not focus on implementing ML components, but rather uses an ML pipeline workflow. Temple et al. (2016) discuss the use of a machine learning approach to create constraints when configuring different products in the same SPL; they as well do not focus on implementing ML components in the SPL and do not mention anything about reuse in any sense.

We have been using these sources (among others) as a foundation and combining the lessons learned from the papers to solve our problems. No articles or journals have been published on creating a generic CSV loader or ML components that work in an SPL. This requires further research to see what the possibilities are, and the quality of an API that allows multiple products to consume the same ML model. We see this as an interesting field to further explore.

11 Conclusion

This research aimed to identify whether it was possible to create machine learning components in a reusable manner for a software product line. Based on our research, we found that it is possible to accomplish, but there are drawbacks and limitations to be considered. These include that the goal of automatic product derivation might not be attainable depending on the type of ML problem the practitioner is trying to solve. In our case, a recommender system might not give the best predictions unless manual quality assurance is involved in testing the end-product. The experience and results gained in this research indicate that a composition-based approach with components may serve as an excellent starting point which the architecture and its components are easily reusable, maintainable, scalable and evolvable.

While our limited prototype (or artefact) and experience in machine learning limit the generalisability of the results, this approach provides new insights to the topic on how to combine machine learning engineering with software product line engineering. This research clearly illustrates that a composition-based approach with a design-for-change mindset will work. Still, it raises the question of whether it is impossible to have an automatic product derivation.

Based on these conclusions, practitioners should consider whether they spend effort in creating a reusable ML model or that time is better served by having a focus on making product-specific implementations that increase stability and could yield better results. Using software product lines in an all-encompassing strategy to handle all resources and process in a software life-cycle may be beneficial. Providing a *reusable* process to develop, maintain, quality assurance, and evolve will yield significant benefits outside the scope of *pure code* and *component* reuse. To better understand these results, future studies could address other ways on how overall software systems with machine learning capabilities could grow in complexity and how to build them sustainably. Further research is needed to determine the effects of testing, and the relationship between refactoring and coherence from results in regards to ML models and the overarching topic.

In essence, our research helps to solve the problem by providing a first investigation of an approach to engineer the software product line architecture providing lessons learned and practical insights. As time of writing this thesis, there is next-to-none literature on the topic of software product lines in combination with machine learning. Therefore, we believe we addressed a gap in knowledge that might prove useful in the years to come as more and more companies adopt

ML into their organisations.

12 References

Aggarwal, C., Hinneburg, A. and Keim, D., 2001. *On the Surprising Behavior of Distance Metrics in High Dimensional Space*. In Proceedings of the 8th International Conference on Database Theory (ICDT '01)., 1973, pp.420-434.

Al Mamun, M.A. et al., 2019. *Evolution of technical debt: An exploratory study*. Joint Proceedings of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM Mensura 2019): Haarlem, The Netherlands, October 7-9, 2019, pp.87–102.

Anguera, M., Portell, M., Chacón-Moscoso, S. and Sanduvete-Chaves, S., 2018. *Indirect Observation in Everyday Contexts: Concepts and Methodological Guidelines within a Mixed Methods Framework*. *Frontiers in Psychology*, 9.

Apel, S., Batory, D., Kästner, C. and Saake, G., 2013. *Feature-Oriented Software Product Lines*. pp.1-278.

Arya, D., Venkataraman, G., Grover, A. and Kenthapadi, K., 2017. *Candidate Selection for Large Scale Personalized Search and Recommender Systems*. Association for Computing Machinery, pp.1391-1393.

Assawiel, N. (2018). *What to do when your training and testing data come from different distributions*. [online] freeCodeCamp.org. Available at: <https://www.freecodecamp.org/news/what-to-do-when-your-training-and-testing-data-come-from-different-distributions-d89674c6ecd8/> [Accessed 11 Sep. 2019].

Bacciu, D., Gnesi, S., & Semini, L. (2015). *Using a Machine Learning Approach to Implement and Evaluate Product Line Features*. 11th International Workshop on Automated Specification and Verification of Web System (WWV'15). EPTCS 188, 2015, pp. 75-83. DOI: 10.4204/EPTCS.188.8

Bajo, A. (2020). *A Production Machine Learning Pipeline for Text Classification with fastText*. [online] Medium. Available at: <https://towardsdatascience.com/a-production-machine-learning-pipeline-for-text-classification-with-fasttext-7e2d3132c781> [Accessed 5 Feb. 2020].

Bhatia, M., 2018. *Your Guide To Qualitative And Quantitative Data Analysis Methods*.

[online] Atlan — Humans of Data. Available at: <https://humansofdata.atlan.com/2018/09/qualitative-quantitative-data-analysis-methods/> [Accessed 23 April 2020].

Bosch, J., 2017. *Structure Eats Strategy – Software Driven World*. [online] Janbosch.com. Available at: <https://janbosch.com/blog/index.php/2017/11/25/structure-eats-strategy/> [Accessed 7 April 2020].

Breuel, C., 2020. *ML Ops: Machine Learning As An Engineering Discipline*. [online] Medium. Available at: <https://towardsdatascience.com/ml-ops-machine-learning-as-an-engineering-discipline-b86ca4874a3f> [Accessed 24 May 2020].

Brownlee, J. (2020). *A Tour of Machine Learning Algorithms*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/> [Accessed 30 Jan. 2020].

Brownlee, J., 2019. *A Gentle Introduction To Cross-Entropy For Machine Learning*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/cross-entropy-for-machine-learning/> [Accessed 14 April 2020].

Camillieri, C., Parisi, L., Blay-Fornarino, M., Precioso, F., Riveill, M., Cancela Vaz, J., Mayerhofer, T., Pierantonio, A., Schätz, B. and Tamzalit, D., 2016. *Towards a software product line for machine learning workflows: Focus on supporting evolution*. Journal of Proceedings of the 10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems and Software, 1706, pp.65-70.

Cardoza, C., 2018. *Google Introduces Tensorflow Hub For Reusable Machine Learning Models - SD Times*. [online] SD Times. Available at: <https://sdtimes.com/ai/google-introduces-tensorflow-hub-reusable-machine-learning-models/> [Accessed 19 March 2020].

Chauhan, N. (2019). *Decision Tree Algorithm — Explained*. [online] Medium. Available at: <https://towardsdatascience.com/decision-tree-algorithm-explained-83beb6e78ef4> [Accessed 5 Feb. 2020].

Chen, L., Ali Babar, M. and Nuseibeh, B., 2013. *Characterizing Architecturally Significant Requirements*. IEEE Software, 30(2), pp. 38-45.

Chen, Long-Sheng & Hsu, Fei-Hao & Chen, Mu-Chen & Hsu, Yuan-Chia. (2008). *Devel-*

oping recommender systems with the consideration of product profitability for sellers. Information Sciences. 178. pp. 1032-1048.

Clear, J., 2017. *First Principles: Elon Musk On The Power Of Thinking For Yourself.* [online] James Clear. Available at: <https://jamesclear.com/first-principles> [Accessed 4 May 2020].

Covington, P., Adams, J. and Sargin, E., 2016. *Deep Neural Networks for YouTube Recommendations.* Association for Computing Machinery, 110, pp.191-198.

Crang, M. and Cook, I., 2007. *Doing Ethnography.* SAGE Publications Inc.,

Das, A., Mathieu, C. and Ricketts, D. (2009). *Maximizing profit using recommender systems.* ArXiv, abs/0908.3633, pp.1-5.

Deng, H. (2019). *Recommender Systems in Practice.* [online] Medium. Available at: <https://towardsdatascience.com/recommender-systems-in-practice-cef9033bb23a> [Accessed 8 May 2019].

Deshpande, M. and Karypis, G., 2004. *Item-based top- N recommendation algorithms.* ACM Transactions on Information Systems (TOIS), 22(1), pp.143-177.

Di Stefano, J. and Menzies, T., 2002. *Machine Learning for Software Engineering: Case Studies in Software Reuse.* IEEE Transactions on Applications and Industry, pp.246-251.

Eichberg, M., Klose, K., Mitschke, R. and Mezini, M., 2010. *Component Composition Using Feature Models.* Component-Based Software Engineering, pp.200-215.

Elliott, E., 2018. *Composing Software: An Exploration of Functional Programming and Object Composition in JavaScript.* Leanpub, pp.1-246.

En.wikipedia.org. 2020. *Comma-Separated Values.* [online] Available at: https://en.wikipedia.org/wiki/Comma-separated_values [Accessed 30 March 2020].

En.wikipedia.org. 2020. *Generic Programming.* [online] Available at: https://en.wikipedia.org/wiki/Generic_programming [Accessed 23 March 2020].

En.wikipedia.org. 2020. *Pearson Correlation Coefficient.* [online] Available at:

https://en.wikipedia.org/wiki/Pearson_correlation_coefficient [Accessed 24 March 2020].

En.wikipedia.org. 2020. *Sigmoid Function*. [online] Available at:
https://en.wikipedia.org/wiki/Sigmoid_function [Accessed 14 April 2020].

En.wikipedia.org. 2020. *Standing On The Shoulders Of Giants*. [online] Available at:
https://en.wikipedia.org/wiki/Standing_on_the_shoulders_of_giants [Accessed 1 June 2020].

Feldt, R. and Magazinius, A., 2010. *Validity Threats in Empirical Software Engineering Research - An Initial Survey*. SEKE, pp.374-379.

Ferguson, R., 2018. *Decisions For Sustaining A Software Product Line*. [online] Insights.sei.cmu.edu. Available at:
https://insights.sei.cmu.edu/sei_blog/2018/10/decisions-for-sustaining-a-software-product-line.html
[Accessed 12 April 2020].

Fleder, Daniel and Hosanagar, Kartik. (2007). *Recommender systems and their impact on sales diversity*. EC'07 - Proceedings of the Eighth Annual Conference on Electronic Commerce. pp. 192-199.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, p.32.

Gandhi, R. (2018). *Naive Bayes Classifier*. [online] Medium. Available at: <https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c> [Accessed 7 Feb. 2020].

Ghorbani, A. and Zou, J., 2017. *Why Are Deep Networks Fragile: Deformation of Intertwined Data*. International Conference on Machine Learning (ICML).

Goasduff, L. (2015). *Taking a First Step to Advanced Analytics*. [online] Gartner.com. Available at: <https://www.gartner.com/smarterwithgartner/taking-a-first-step-to-advanced-analytics/>
[Accessed 4 Sep. 2019].

Gohrani, K., 2019. *Different Types Of Distance Metrics Used In Machine Learning*. [online] Medium. Available at: https://medium.com/@kunal_gohrani/different-types-of-distance-metrics-used-in-machine-learning-e9928c5e26c7 [Accessed 30 March 2020].

Goldberg, Y., 2020. *Node.js Best Practices*. [online] GitHub. Available at:

<https://github.com/goldbergryoni/nodebestpractices> [Accessed 23 May 2020].

Gowda, D., 2017. *Data Shuffling - Why It Is Important In Machine Learning & How To Do It?*. [online] LinkedIn.com. Available at: <https://www.linkedin.com/pulse/data-shuffling-why-important-machine-learning-how-do-deepak-n-gowda/> [Accessed 13 April 2020].

Gregor, Shirley and Hevner, Alan. (2013). *Positioning and Presenting Design Science Research for Maximum Impact*. MIS Quarterly. pp. 337-356.

Gupta, A. (2019). *ML — Semi-Supervised Learning*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/ml-semi-supervised-learning/> [Accessed 4 Feb. 2020].

Gupta, A., 2018. *Understanding Logistic Regression*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/understanding-logistic-regression/> [Accessed 15 April 2020].

Hamlet, D., 2010. *Composing Software Components*. New York: Springer.

Heller, M. (2019). *Machine learning algorithms explained*. [online] InfoWorld. Available at: <https://www.infoworld.com/article/3394399/machine-learning-algorithms-explained.html> [Accessed 7 Jan. 2020].

Hevner, Alan & R, Alan & March, Salvatore & T, Salvatore & Park, & Park, Jinsoo & Ram, & Sudha,. (2004). *Design Science in Information Systems Research. 1st ed.* Management Information Systems Quarterly, pp. 75-105.

Hosch, W., 2011. *Euclidean Space*. [online] Encyclopedia Britannica. Available at: <https://www.britannica.com/science/Euclidean-space> [Accessed 24 March 2020].

Isinkaye, F., Folajimi, Y. and Ojokoh, B., 2015. *Recommendation systems: Principles, methods and evaluation*. Egyptian Informatics Journal, 16(3), pp.261-273.

Jannach, D. and Adomavicius, G. (2017). *Price and Profit Awareness in Recommender Systems*. ArXiv, abs/1707.08029, pp.1-6.

Kundu, S., 2019. *Various Types Of Distance Metrics Machine Learning*. [online] Medium. Available at: <https://medium.com/analytics-vidhya/various-types-of-distance-metrics-machine-learning-cc9d4698c2da> [Accessed 25 March 2020].

Liao, K. (2018). *Prototyping a Recommender System Step by Step Part 1: KNN Item-Based Collaborative Filtering*. [online] Medium. Available at: <https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-1-knn-item-based-collaborative-filtering-637969614ea> [Accessed 13 Feb. 2020].

Loshin, D. (2016). *Supervised vs. Unsupervised Learning (Part 3 in a Series)*. [online] Transforming Data with Intelligence. Available at: <https://tdwi.org/articles/2016/03/17/supervised-vs-unsupervised-learning.aspx> [Accessed 13 Feb. 2020].

Luo, S. (2018). *Intro to Recommender System: Collaborative Filtering*. [online] Medium. Available at: <https://towardsdatascience.com/intro-to-recommender-system-collaborative-filtering-64a238194a26> [Accessed 12 Feb. 2020].

Masters, M., 2010. *An Overview Of Requirements Elicitation*. [online] Modernanalyst.com. Available at: <https://www.modernanalyst.com/Resources/Articles/tabid/115/ID/1427/An-Overview-of-Requirements-Elicitation.aspx> [Accessed 7 April 2020].

McCaw, B., 2019. *The Batch: Companies Slipping On AI Goals, Self Training For Better Vision, Muppets And Models, China Vs US?, Only The Best Examples, Proliferating Patents*. [online] Blog.deeplearning.ai. Available at: <https://blog.deeplearning.ai/blog/the-batch-companies-slipping-on-ai-goals-self-training-for-better-vision-muppets-and-models-china-vs-us-only-the-best-examples-proliferating-patents> [Accessed 27 May 2020].

Mitra, A., 2020. *Methods Of Study Designs- Observational Studies & Surveys*. [online] Medium. Available at: <https://towardsdatascience.com/methods-of-study-designs-observational-studies-surveys-22f0a04c7446> [Accessed 21 April 2020].

ML Glossary. 2017. *Loss Functions*. [online] Available at: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html [Accessed 14 April 2020].

Morisio, M., Ezran, M. and Tully, C., 2002. *Success and failure factors in software reuse*. IEEE Transactions on Software Engineering, 28(4), pp.340-357.

Mwiti, D. (2018). *How to build a Simple Recommender System in Python*. [online] Medium. Available at: <https://towardsdatascience.com/how-to-build-a-simple-recommender-system-in-python-375093c3fb7d> [Accessed 10 Sep. 2019].

Northrop, L., 2010. *Introduction to Software Product Lines*. Software Product Lines: Going Beyond, pp.521-522.

Orlikowski, W. and Iacono, C. (2001). *Research Commentary: Desperately Seeking the "IT" in IT Research—A Call to Theorizing the IT Artifact*. Information Systems Research, 12(2), pp.121-134.

Pandey, P. (2019). *The Remarkable world of Recommender Systems*. [online] Medium. Available at: <https://towardsdatascience.com/the-remarkable-world-of-recommender-systems-bff4b9cbe6a7> [Accessed 14 Feb. 2020].

Pandey, P., 2019. *Understanding The Mathematics Behind Gradient Descent*. [online] Medium. Available at: <https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e> [Accessed 14 April 2020].

Peppers, K., Tuunanen, T., Rothenberger, M. and Chatterjee, S. (2008). *A Design Science Research Methodology for Information Systems Research*. Journal of Management Information Systems, 24(3), pp.45-77.

Pollock, T., 2019. *The Difference Between Structured, Unstructured & Semi-Structured Interviews*. [online] Oliver Parks Consulting LLC - Technology Sector Recruitment Experts. Available at: <https://www.oliverparks.com/blog-news/the-difference-between-structured-unstructured-amp-semi-structured-interviews> [Accessed 19 April 2020].

Presthus, Wanda and Munkvold, Bjørn Erik. (2016). *How to frame your contribution to knowledge? A guide for junior researchers in informaton systems*. Paper presented at NOKO-BIT 2016, vol. 24. pp.1-14.

Qaheaven.blogspot.com. 2011. *BAPO Model*. [online] Available at: <http://qaheaven.blogspot.com/2011/01/bapo-model.html> [Accessed 16 April 2020].

Ram, S., 2003. *Dr. Alan Kay On The Meaning Of "Object-Oriented Programming"*. [online] Userpage.fu-berlin.de. Available at: http://www.purl.org/stefan_ram/pub/doc_kay_oop_en [Accessed 4 May 2020].

Ray, S. (2017). *Essentials of Machine Learning Algorithms (with Python and R Codes)*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2017/09/common->

machine-learning-algorithms/ [Accessed 4 Feb. 2020].

Robinson, A., 2017. *How To Calculate Euclidean Distance*. [online] Sciencing.com. Available at: <https://sciencing.com/how-to-calculate-euclidean-distance-12751761.html> [Accessed 22 March 2020].

Rocca, B. (2019). *Introduction to recommender systems*. [online] Medium. Available at: <https://towardsdatascience.com/introduction-to-recommender-systems-6c66cf15ada> [Accessed 11 Nov. 2019].

Sander, K., 2019. *Induktiv Og Deduktiv Studie*. [online] eStudie.no. Available at: <https://estudie.no/induktiv-deduktiv/> [Accessed 16 April 2020].

Sanjay, M., 2018. *Why And How To Cross Validate A Model?*. [online] Medium. Available at: <https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f> [Accessed 14 April 2020].

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J & Dennison, D. (2015). *Hidden Technical Debt in Machine Learning Systems*. Advances in Neural Information Processing Systems 28 (NIPS 2015), pp. 1-9.

Seif, G., 2020. *An Easy Introduction To Machine Learning Recommender Systems - Kdnuggets*. [online] KDnuggets. Available at: <https://www.kdnuggets.com/2019/09/machine-learning-recommender-systems.html> [Accessed 7 April 2020].

Sharma, N., 2019. *Importance Of Distance Metrics In Machine Learning Modelling*. [online] Medium. Available at: <https://towardsdatascience.com/importance-of-distance-metrics-in-machine-learning-modelling-e51395ffe60d> [Accessed 22 March 2020].

Soni, D. (2020). *Supervised vs. Unsupervised Learning*. [online] Medium. Available at: <https://towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d> [Accessed 30 Jan. 2020].

Srivastava, T. (2018). *Introduction to k-Nearest Neighbors: A powerful Machine Learning Algorithm (with implementation in Python & R)*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/> [Accessed 5 Feb. 2020].

Starkweather, D. and Kay Moske, D., 2011. *Multinomial Logistic Regression*. Science Direct, pp.1-6.

Statistics How To. 2020. *Correlation Coefficient: Simple Definition, Formula, Easy Calculation Steps*. [online] Available at: <https://www.statisticshowto.datasciencecentral.com/probability-and-statistics/correlation-coefficient-formula/> [Accessed 24 March 2020].

Strunk JR., W., 1999. *The Elements of Style, Fourth Edition*. Longman, p.35.

Swaminathan, S. (2018). *Logistic Regression — Detailed Overview*. [online] Medium. Available at: <https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc> [Accessed 15 Apr. 2020].

Tan, L., Lin, Y. and Ye, H., 2012. *Quality-Oriented Software Product Line Architecture Design*. Journal of Software Engineering and Applications, 05(07), pp.472-476.

Teknomo, K., 2015. *Chebyshev Distance*. [online] People.revoledu.com. Available at: <https://people.revoledu.com/kardi/tutorial/Similarity/ChebyshevDistance.html> [Accessed 31 March 2020].

Temple, P., Galindo, J., Acher, M. and Jézéquel, J., 2016. *Using machine learning to infer constraints for product lines*. Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16,.

TensorFlow. 2020. *Tensorflow Tensors*. [online] Available at: <https://www.tensorflow.org/guide/tensor> [Accessed 14 April 2020].

Uio.no. 2020. *Ethical Guidelines*. [online] Available at: <https://www.uio.no/english/about/regulations/ethical-guidelines/> [Accessed 1 June 2020].

van der Linden, F., Schmid, K. and Rommes, E., 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer Berlin Heidelberg, pp. 1-333.

Walsham, G., 2006. *Doing interpretive research*. European Journal of Information Systems, 15(3), pp.320-330.

Weisstein, E., 2020. *Covariance*. [online] Mathworld.wolfram.com. Available at:

<https://mathworld.wolfram.com/Covariance.html> [Accessed 24 March 2020].

Zinkevich, M. (2019). *Rules of Machine Learning: Best Practices for ML Engineering*.
[online] Google Developers. Available at: <https://developers.google.com/machine-learning/guides/rules-of-ml/> [Accessed 30 May 2019].

Appendices

Appendix A

Product requirements from Company A and Company B

Appendix B

Interview templates

Appendix C

Survey questions

Appendix D

Code extract from the prototype

A Product requirements

A.1 Company A - product requirements

The product requirements for Company B's product. Written by Sebastian for Snapper Net Solutions AS.

Company A - Produktkrav

Brukerhistorier

Håndbok

Som en bruker ønsker jeg en rask tilgang på informasjon som er relevant for meg.

Som en bruker ønsker jeg å lese informasjon om ulike arbeidsoppgaver i butikken.

Som en bruker ønsker jeg å få informasjon om varer vi selger i butikken.

Som en bruker ønsker jeg å få informasjon om hvordan applikasjonen (Product A) fungerer.

Som en bruker ønsker jeg å få informasjon om våre tjenester og tilbud.

Som en bruker ønsker jeg å få informasjon om hvordan våre andre applikasjoner fungerer (f.eks. Course A).

Som en bruker ønsker jeg å få informasjon om konseptet vårt (e.g. konsepthåndbok).

Som en bruker ønsker jeg å kunne søke opp (filtrere) den informasjonen jeg trenger.

Kataloger

Som en bruker så ønsker jeg å få informasjon som er relevant for meg og butikken hver innværende uke.

Som en bruker ønsker jeg å se statistikk på kundetilfredshet.

Div

Som en bruker ønsker jeg å være selvstendig og ikke avhengig av en overordnet for å fullføre elementære oppgaver i applikasjonen.

Kurs

Som en bruker ønsker jeg å få tilgang til alle kurs som er relevant for meg.

Som en bruker ønsker jeg å få en oversikt over hvor mange kurs jeg har gjennomført.

Highscore

Som en bruker ønsker jeg å se hvordan jeg gjør det i forhold til mine kollegaer.

Som en bruker ønsker jeg å se hvordan jeg gjør det i forhold til ansatte i samvirkelaget.

Som en bruker ønsker jeg å se hvordan jeg gjør det i forhold til andre på landsbasis.

Som en bruker ønsker jeg å se hvordan butikken min gjør det i forhold til andre butikker i samvirkelaget.

Som en bruker ønsker jeg å se hvordan butikken min gjør det i forhold til andre butikker på landsbasis.

Løype

Som en bruker ønsker jeg å se min egen progresjon gjennom løypen.

Som en bruker ønsker jeg å se hvilke *nano-kurs* jeg har tilgjengelig.

Som en bruker ønsker jeg å vite når neste løype blir lansert.

Som en bruker ønsker jeg å kunne gå tilbake å se hvor mange poeng jeg fikk på et *nano-kurs*.

Innlogging

Som en bruker ønsker jeg å kunne få hjelp til få tilbake passord om jeg har glemt det.

Butikksjefs administrasjon:

Som en butikksjef ønsker jeg å se fremgangen til alle mine ansatte.

Som en butikksjef ønsker jeg å se hvilke ansatte som ikke er aktive.

Som en butikksjef ønsker jeg å se butikken sin gjennomføringsgrad.

A.2 Company B - product requirements

The product requirements for Company A's product. Written by Sebastian for Snapper Net Solutions AS.

Company B - Produktkrav

Brukerhistorier

Håndbok

Som en bruker ønsker jeg en rask tilgang på informasjon som er relevant for meg.

Som en bruker ønsker jeg å lese informasjon om ulike arbeidsoppgaver i butikken.

Som en bruker ønsker jeg å få informasjon om varer vi selger i butikken.

Som en bruker ønsker jeg å få informasjon om hvordan applikasjonen (Company B App) fungerer.

Som en bruker ønsker jeg å få informasjon om våre tjenester og tilbud.

Som en bruker ønsker jeg å få informasjon om hvordan våre andre applikasjoner fungerer.

Som en bruker ønsker jeg å få informasjon om konseptet vårt (e.g. konsepthåndbok).

Som en bruker ønsker jeg å kunne søke opp (filtrere) den informasjonen jeg trenger.

Div

Som en bruker ønsker jeg å være selvstendig og ikke avhengig av en overordnet for å fullføre elementære oppgaver i applikasjonen.

Kurs

Som en bruker ønsker jeg å få tilgang til alle kurs som er relevant for meg.

Som en bruker ønsker jeg å få en oversikt over hvor mange kurs jeg har gjennomført.

Highscore

Som en bruker ønsker jeg å se hvordan jeg gjør det i forhold til mine kollegaer.

Som en bruker ønsker jeg å se hvordan jeg gjør det i forhold til ansatte i samvirkelaget.

Som en bruker ønsker jeg å se hvordan jeg gjør det i forhold til andre på landsbasis.

Som en bruker ønsker jeg å se hvordan butikken min gjør det i forhold til andre butikker i samvirkelaget.

Som en bruker ønsker jeg å se hvordan butikken min gjør det i forhold til andre butikker på landsbasis.

Løype

Som en bruker ønsker jeg å se min egen progresjon gjennom løypen.

Som en bruker ønsker jeg å se hvilke *nano-kurs* jeg har tilgjengelig.

Som en bruker ønsker jeg å vite når neste løype blir lansert.

Som en bruker ønsker jeg å kunne gå tilbake å se hvor mange poeng jeg fikk på et *nano-kurs*.

Kart

Som en bruker ønsker jeg å kunne velge hvilken løype jeg skal kunne se.

Som en bruker ønsker jeg å kunne opprette flere kart (forskjellige visninger).

Company B

Innlogging

Som en bruker ønsker jeg å kunne få hjelp til få tilbake passord om jeg har glemt det.

B Interview templates

B.1 Architecture interview template

Interview template from architectural viewpoint.

Subject-Matter Experts Analysis - Code & Architecture

Interviewer:

Date:

Time:

Observer:

Place:

Code being analysed:

Expert:

Gender:

1. Are there any repetitive lines of code or opportunities for useful abstraction?

-

2. How much would you estimate you needed to create a separate version of this product?

-

3. Do you understand how to use [this] interface?

-

4. How easy is it to understand?

-

5. Does this code accomplish the author's purpose?

-

6. Do you see potential for useful abstractions?

-

7. Does this code follow standard patterns/guidelines?

-

8. Is the code maintainable easily?

-

9. Can I unit test / debug the code easily to find the root cause?

-

10. Is this function or class too big? Does it have too many responsibilities?

-

11. Is the code thoroughly commented?

-

a. - Does the intent comment match the logic?

-

b. - Does the intent comment have any typos?

-

12. Does this code change compile-time or run-time dependencies between projects?

-

13. Are any of your objections based on subjective preferences?

-

14. Would you have written it in the same way?

-

15. Is the performance acceptable with huge data?

-

16. Is the code meeting non-functional requirements?

-

17. Does the code follow the defined architecture?

-

18. Is the code following OOAD (Object Oriented Analysis & Design) principles?

-

a. - Does the code follow the single responsibility principle?

-

b. - Open Closed Principle

-

c. - Dependency Injection

-

19. Is there any hard-coding? Should use constants or config values.

-

20. Does it use framework features, wherever possible?

-

21. Is the code reusable?

-

a. - DRY (Do not Repeat Yourself) principle: The same code should not be repeated more than twice.

-

b. - Consider reusable services, functions and components.

-

c. - Consider generic functions and classes.

-

B.2 Business and process interview template

Interview template from business and process point of view.

Subject-Matter Experts Analysis - Business & Process

Interviewer:	Date:	Time:
Observer:	Place:	
Code being analysed:		
Expert:	Gender:	

Business view

1. How will the software product line prototype impact business performance?

-

2. How can software product line engineering be introduced in the company?

-

3. How marketable would our software product line prototype be for the company?

-

4. How would the software product line prototype evolve over time?

-

5. What is the relationship between the individual products in the software product line?

-

6. Can any of the products from the same product line supplant each other over time on the market?

-

7. Can any of the products from the same product line support each other over time on the market?

-

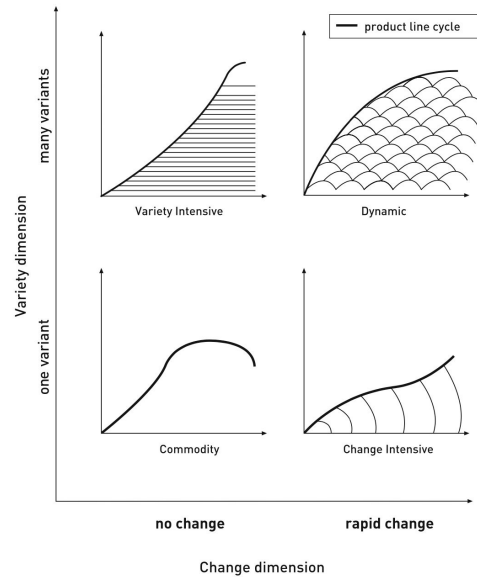
8. Based on the prototype we have made, how long time-period do you think it will take to mature the software product line?

-

9. In a young market, the required product variety is usually much lower than in a mature market. How mature would you consider the market our Software Product Line competes in?

-

The first dimension describes how many different products are available on the market at the same time. The second dimension describes how fast a new product is supplanted by a newer one.



10. The overall product line life-cycle can exhibit different forms of dynamism as shown in the figure above. Which kind of life cycle is most appropriate for our software product line in regards to the characteristics of the market?

-

Process view

1. Do you think the process of adopting new products to the model would be cost-effective in the long term?

-

2. Do you think the process of adopting new products to the model would be time-consuming and demanding?

-

3. Does it require a cross-functional team to make instances of products to the SPL (that can use the recommender system)?

-

- 3.1. What competence is required in the cross-functional team?

-

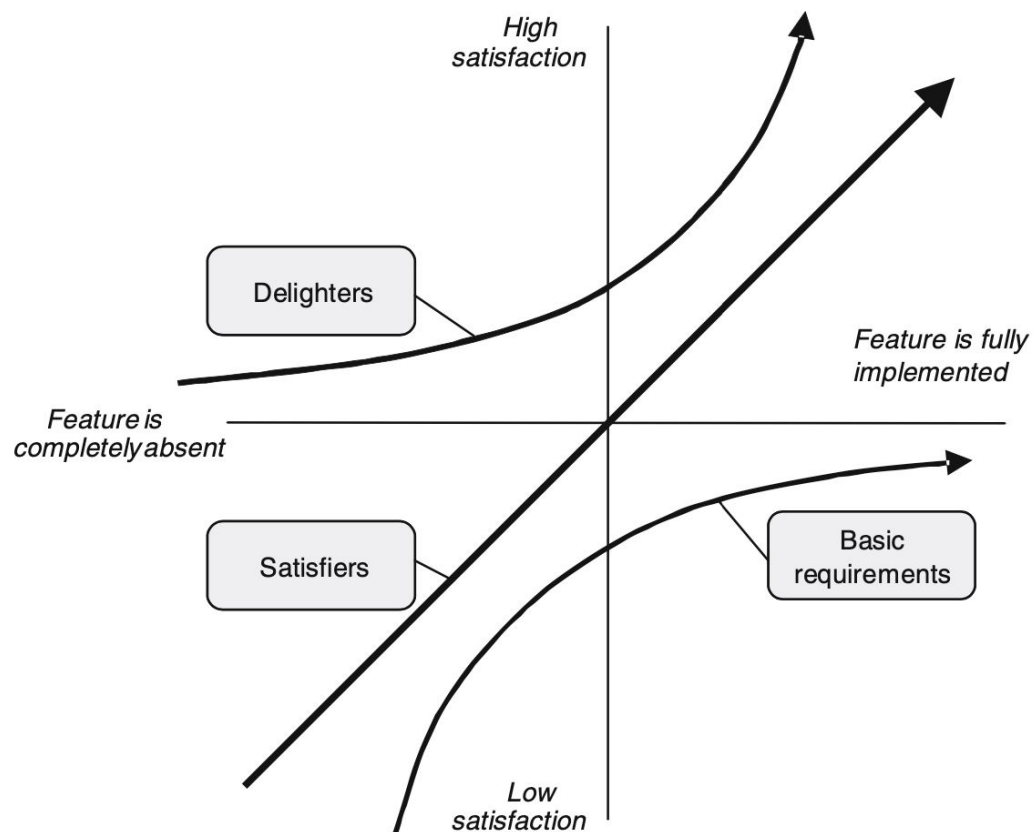
- 3.2. Would the team be divided into a domain- and application-engineering groups?

-

4. In terms of reliability, safety and usability, how big of an impact would a change to a software product line approach benefit these product qualities?

-

The KANO model is an approach used to define new products on the market. The distinction between delighters, satisfiers, and basic requirements provides the basis for determining the chances of a product on the market.



- *Delighters: These go beyond standard customer expectations – and thus beyond the competition. In order to be successful, a product should have some delighters.*
- *Satisfiers: Customer satisfaction is roughly proportional to the degree of satisfaction of the requirement.*
- *Basic requirements: These correspond to the fundamental expectations of the customers. As a consequence, these must be fulfilled in order for a product to be successful.*

5. Which category would you classify the software product line prototype to be in?

-

6. How many products do you envision the software product line to support?

-

7. How many products would you estimate to be similar? And how many products would require product-specific configurations?

-

C Survey - Google Form

The survey we published through Google Forms, containing 11 questions on recommender systems and how people interact with them.

Survey on Recommender Systems

Dear respondent, thank you for participating in this study.

The study aims to investigate respondents' attitudes towards recommender systems. It will only take about 5 minutes.

The survey is anonymous, and all data will be treated confidentially. There are no right or wrong answers, we are merely interested in your honest opinions.

Participating in this study is voluntary, and you have the right to withdraw at any time.

Kind regards,
Sebastian and Jørgen
MSc students at University of Oslo

***Må fylles ut**

Background

We interact with recommender (or recommendation) systems (RS) on a regular basis, when we use digital services and apps.

For example Amazon, Medium, Facebook Feed, YouTube, Tinder and Netflix.

These services and apps use algorithms to make suggestions about what a user may like, such as a specific movie, an article or a product. More formally, the algorithms use information about a user's preferences (e.g. about movies) to predict the rating a user would give the item under evaluation, and predict how they would rank a set of items individually or as a collection.

To work effectively and efficiently, recommender systems collect, curate and act upon vast amounts of personal data. Inevitably, shaping the individual experience of digital environments and social interactions.

How old are you? *

- Under 18
- 18 to 24
- 25 to 34
- 35 to 44
- 45 to 54
- 55 to 64
- 65 or older

Do you understand what a recommender system (RS) is? *

- Yes
- No
- Gist of it

How often do you think you are exposed to a recommender system?

- Annually
- Monthly
- Weekly
- Daily
- Multiple times daily

Do you think recommendations made by a recommender system will help you make decisions more effectively (on a general basis), e.g. in terms of increased satisfaction?

- 1 2 3 4 5 6 7
- Not at all likely Likely

Do you lose trust in the recommender system if you get a bad recommendation?

- 1 2 3 4 5 6 7
- Not likely Likely

Have you been offended by a recommendation before?

- Yes
- No

No

Do bad recommendations annoy you?

- Not at all
- Sometimes
- Often
- Very often
- Always

Have you been recommended inappropriate content before?

- Yes
- No

How important is it for you to know when an algorithm is responsible for displaying (or recommending) you a certain item, e.g. movie or product?

- 1 2 3 4 5 6 7
- Not important Important

How important is it for you to get an informative explanation to why you received a certain recommendation?

- 1 2 3 4 5 6 7
- Not important Important

In what categories of services do you generally like to receive recommendations (made by an algorithm)?

- News Aggregators (e.g. Apple News, Flipboard)
- E-learning Platforms (e.g. Udemy)

1 2 3 4 5 6 7

Not important ○ ○ ○ ○ ○ ○ ○ Important

In what categories of services do you generally like to receive recommendations (made by an algorithm)?

- News Aggregators (e.g. Apple News, Flipboard)
- E-learning Platforms (e.g. Udemey)
- Social Media Platforms (e.g. Facebook, Twitter, Instagram)
- Streaming Services (e.g. Netflix, YouTube)
- Dating Applications (e.g. Tinder)
- Live Streaming Platforms (e.g. Twitch, Mixer)
- Publishing Platforms (e.g. Medium)
- Music Services (e.g. Spotify, iTunes, Tidal, SoundCloud)
- E-commerce (e.g. Amazon, Zalando)
- Andre:

Send

Send aldri passord via Google Skjemaer.

Dette innholdet er ikke laget eller godkjent av Google. [Rapportér misbruk](#) - [Vilkår for bruk](#) - [Retningslinjer for personvern](#)

Google Skjemaer

D Code extract from the prototype

D.1 Engine

Code extract of the engine.

```

1 const tf = require("@tensorflow/tfjs");
2 const _ = require("lodash");
3
4 class Engine {
5   constructor(features, labels, options) {
6     this.features = this.processFeatures(features);
7     this.labels = tf.tensor(labels);
8     this.costHistory = [];
9     this.bHistory = [];
10
11    this.options = Object.assign(
12      { learningRate: 0.1, iterations: 1000, decisionBoundary: 0.5 },
13      options
14    );
15
16    // Diff to regular Logistic Regression is referencing the number
17    // of columns here
18    this.weights = tf.zeros([this.features.shape[1], this.labels.shape[1]]);
19  }
20
21  gradientDescent(features, labels) {
22    const currentGuesses = features.matMul(this.weights).sigmoid();
23    const differences = currentGuesses.sub(labels);
24
25    const slopes = features
26      .transpose()
27      .matMul(differences)
28      .div(features.shape[0]);
29
30    this.weights = this.weights.sub(slopes.mul(this.options.learningRate));
31  }
32
33  train() {
34    // How many total batches there are
35    const batchQuantity = Math.floor(
36      this.features.shape[0] / this.options.batchSize
37    );
38
39    for (let i = 0; i < this.options.iterations; i++) {
40      for (let j = 0; j < batchQuantity; j++) {
41        const startIndex = j * this.options.batchSize;
42        const { batchSize } = this.options;
43        const featureSlice = this.features.slice(
44          [startIndex, 0],
45          [batchSize, -1]
46        );
47        const labelSlice = this.labels.slice([startIndex, 0], [batchSize, -1]);
48
49        this.gradientDescent(featureSlice, labelSlice);
50      }
51      this.recordCost();
52      this.updateLearningRate();
53    }
54  }
55 }

```

```
56 predict(observations) {
57   return this.processFeatures(observations).matMul(this.weights).sigmoid();
58 }
59
60 test(testFeatures, testLabels) {
61   const predictions = this.predict(testFeatures);
62   testLabels = tf.tensor(testLabels); //argMax(1);
63
64   const incorrect = predictions.sub(testLabels).abs().sum().get();
65   console.log(incorrect);
66
67   return (predictions.shape[0] - incorrect) / predictions.shape[0];
68 }
69
70 // Put features into a Tensor and add 1's
71 processFeatures(features) {
72   features = tf.tensor(features);
73
74   // If these are defined properties, we need to re-apply
75   // to re-standardize the features, else, standardize for
76   // the first time
77   if (this.mean && this.variance) {
78     features = features.sub(this.mean).div(this.variance.pow(0.5));
79   } else {
80     features = this.standardize(features);
81   }
82
83   // Has to be defined after standardization to prevent
84   // 1's being standardized into -0,9999995
85   features = tf.ones([features.shape[0], 1]).concat(features, 1);
86
87   return features;
88 }
89
90 standardize(features) {
91   const { mean, variance } = tf.moments(features, 0);
92
93   this.mean = mean;
94   this.variance = variance;
95
96   return features.sub(mean).div(variance.pow(0.5));
97 }
```

```

119 recordCost() {
120   // - (1/n) * (Actual ^T * log(Guesses) + (1 - Actual)^T log(1 - Guesses))
121   // Guesses - Make one guesses array instead of calculating it twice
122   const guesses = this.features.matMul(this.weights).softmax();
123
124   // (Actual ^Transpose * log(Guesses))
125   const termOne = this.labels.transpose().matMul(guesses.log());
126   // (-1 Actual) ^Transpose log(1 - Guesses)
127   const termTwo = this.labels
128     .mul(-1)
129     .add(1)
130     .transpose()
131     .matMul(guesses.mul(-1).add(1).log());
132
133   // - (1 / N) then add the two terms of equation
134   const cost = termOne
135     .add(termTwo)
136     .div(this.features.shape[0])
137     .mul(-1)
138     .get(0, 0);
139
140   this.costHistory.unshift(cost);
141 }
142
143 updateLearningRate() {
144   if (this.costHistory.length < 2) {
145     return;
146   }
147   if (this.costHistory[0] > this.costHistory[1]) {
148     this.options.learningRate /= 2;
149   } else {
150     this.options.learningRate *= 1.05;
151   }
152 }
153 }
154
155 module.exports = LogisticRegression;
156

```

D.2 CSV Loader

Code extract of the CSV loader.

```

1 const fs = require("fs");
2 const { Transform } = require("stream");
3 const _ = require("lodash");
4 const shuffleSeed = require("shuffle-seed");
5
6 function readStream(
7   filename,
8   {
9     converters = {},
10    dataColumns = [],
11    labelColumns = [],
12    shuffle = true,
13    seed = "standard",
14    splitTest = false,
15  }
16) {
17  return new Promise((resolve, reject) => {
18    let data = [];
19    stream = fs
20      .createReadStream(filename, { encoding: "UTF-8" })
21      .pipe(clean(converters));
22    stream.on("data", (chunk) => {
23      data.push(chunk);
24    });
25
26    const transformedData = () => {
27      data = data.flat(1);
28      let labels = extractColumns(data, labelColumns);
29      data = extractColumns(data, dataColumns);
30
31      data.shift();
32      labels.shift();
33
34      if (shuffle) {
35        data = shuffleSeed.shuffle(data, seed);
36        labels = shuffleSeed.shuffle(labels, seed);
37      }
38
39      if (splitTest) {
40        const trainSize = _.isNumber(splitTest)
41          ? splitTest
42          : Math.floor(data.length / 2);
43
44        return {
45          features: data.slice(0, trainSize),
46          labels: labels.slice(0, trainSize),
47          testFeatures: data.slice(trainSize),
48          testLabels: labels.slice(trainSize),
49        };
50      } else {
51        return { features: data, labels };
52      }
53    };
54
55    stream.on("end", () => resolve(transformedData()));
56    stream.on("error", (error) => reject(error));
57  });
58 }

```



```
60 function clean(converters) {
61   return new Transform({
62     objectMode: true,
63     transform(row, encoding, callback) {
64       row = row.split("\n").map((l) => l.split(","));
65       row = row.map((l) => _.dropRightWhile(l, (val) => val === ""));
66
67       const headers = _.first(row);
68
69       row = row.map((l, index) => {
70         if (index === 0) {
71           return l;
72         }
73
74         return l.map((element, index) => {
75           if (converters[headers[index]]) {
76             const converted = converters[headers[index]](element);
77             return _.isNaN(converted) ? element : converted;
78           }
79
80           const result = parseFloat(element);
81           return _.isNaN(result) ? element : result;
82         });
83       });
84
85       callback(null, row);
86     },
87   });
88 }
89
```

```
90 function extractColumns(data, columnNames) {
91   const headers = _.first(data);
92
93   const indexes = _.map(columnNames, (column) => headers.indexOf(column));
94   const extracted = _.map(data, (row) => _.pullAt(row, indexes));
95
96   return extracted;
97 }
98
99 async function loadCSV(filename, options) {
100   const result = await readStream(filename, options);
101   const { features, labels, testFeatures, testLabels } = result;
102
103   console.log("Features:", features);
104   console.log("Labels:", labels);
105   console.log("TestFeatures:", testFeatures);
106   console.log("TestLabels:", testLabels);
107 }
108
109 loadCSV("data.csv", {
110   dataColumns: ["valueA", "valueB"],
111   labelColumns: ["valueC"],
112   seed: "seeded",
113   shuffle: true,
114   splitTest: 2,
115   converters: {
116     valueC: (val) => (val === "TRUE" ? 1 : 0),
117   },
118 });
119
```

