UiO : Department of Informatics
University of Oslo

Making sense of the human genome using machine learning

Fredrik Haaland
Master Thesis Spring 2013

Making sense of the human genome using machine learning

Fredrik Haaland

April 30, 2013

Abstract

Machine learning enables a computer to learn a relationship between two assumingly related types of information. One type of information could thus be used to predict any lack of information in the other using the learned relationship. During the last decades, it has become cheaper to collect biological information, which has resulted in increasingly large amounts of data.

Biological information such as DNA is currently analyzed by a variety of tools. Although machine learning has already been used in various projects, a flexible tool for analyzing generic biological challenges has not yet been made.

The challenges of representing biological data in a generic way that permits machine learning is here discussed. A flexible machine learning application is presented for working on currently available biological DNA. Also, it targets biological challenges in an abstract manner, so that it may become useful for both current and future challenges.

The application has been implemented in The Genomic HyperBrowser and is publicly available. An use case inspired by a biological challenge demonstrates the application usage. A machine learned model is analyzed and used for making predictions. The results are discussed and further actions of how to improve the model is proposed.

The application offers a new way for researchers to investigate and analyze biological data using machine learning.

Contents

List of Figures

List of Tables

Preface

The thesis is written as if it were to be read by a fellow master student. By reading the thesis, the reader should get some background information into some of the main topics and challenges within both machine learning and bioinformatics.

Personally, I had little or no prior knowledge about biology, DNA sequencing and machine learning alltogether. It has been both interesting and challenging to get to know and learn about these complex fields. The learning curve has been long and steep. I am truely blessed by having had the opportunity to get to know the topics and to create and implement a publicly available application as a result. I had never been able to do it without the help and support from my supervisors, family, colleagues and friends.

First of all, I want to thank my head supervisor, *Geir Kjetil Sandve*, for giving me the chance to work on the thesis, and for his great support and direction along the way. Also, I want to thank my co-supervisors, *Eivind Hovig* and *Gunnar Liestøl* for sharing their knowledge and expertise. Thanks to *Nils Damm Christophersen*, *Gwyn Stephens* and *Peter Wilhelm Stray-Røding* for their input, help and direction. Thanks for all your help with detecting spelling errors.

I would like to wholeheartedly thank my in-laws, *Grete and Jon Baggetorp*, for having provided me with everything I have needed through countless visits and late night hours.

I am truly greatful for having the most wonderful parents in the world. *Grete and Geir Haaland*, you have always given me the chance to have faith in myself and the things I can achieve. You are the ones who have tought me the true meaning of life, and directed me towards our loving saviour Jesus Christ.

Finally, I will thank my beautiful wife *Therese* for her dedication and loving support throughout the whole process.

Psalm 28, 6-9 by David

Blessed *be* the LORD, because he hath heard the voice of my supplications.
The LORD *is* my strength and my shield; my heart trusted in him, and I am helped: therefore my heart greatly rejoiceth; and with my song will I praise him.
The LORD *is* their strength, and he *is* the saving strength of his anointed.
Save thy people, and bless thine inheritance: feed them also, and lift them up for ever.

King James Version

Part I

Introduction

Motivation

Biological challenges are interesting because they deal with the very foundations of mankind. Addressing the current challenges is key to develop medicine to both prevent and cure diseases. Machine learning is a tool, one among many, for addressing the challenges.

Machine learning aims to make computers learn models or patterns which could be used for analysis, intepretation and decision making. A computer may learn from mathemathical techniqes (i.e regression analysis), complex computer-algorithms (data-mining, artificial intelligence), amongst others. Regression analysis [12, 42] is a statistical technique for *understanding* and *interpreting* relationships between independent and dependent mathematical variables, by estimation. A (probable) relationship may be examined using various techniques which explains one or more dependent variables based on one or more independent variables using a statistical model. A model is deterministic if it explains (in a complete manner) the dependent variables based on the independent ones. In many real-world scenarios, this is not possible. Instead, statistical (or stochastic) models tries to approximate exact solutions, by evaluating probabilistic distributions. The decisions made by using such models may be supported by various indicators (e.g. a confidence interval). Creating models and using probability distributions and indicators for decision making and forecasting are closely related with machine learning, even though machine learning may be understood more widely since it also have a branch to artificial intelligence.

In 1962, Arthur Lee Samuel programmed a checkers game on his computer which later on beat him in the same game [30]. He accomplished this applying a technique, later known as *machine learning*. The computer *learned* checker moves (positions) which would maximize its likelihood of winning, by playing thousands of times against itself. Then, after learning the relationship between moves (independent variables) and outcomes (dependent variables), the author of the program stood no chance against the computer. Later on, the computer created a small revolution by winning against a self-proclaimed master player of checkers. This particular checkers program is thought to be the first *self-learned* program. By knowing all checker-states and outcomes, the computer would naturally select the best available move (from any given situation) which would maximize its possibility of winning the game.

If a computer knows all moves and outcomes, so that it always will select the optimal move, then the game is said to be *solved*. The question of whether a computer is able to learn, still remains a philosophical question. Enthusiast would probably underline that being able to predict a correct answer concerning a given problem most of the time, is the same as having learned how to deal with it. Others would say that making errors is what makes us human.

Personally, this has given me the passion to explore in what ways, and possibly how well, machine learning could be used to answer current challenges or problem formulations by using available genomic data.

Related Work

Various projects (e.g. [28, 31]) have already applied various machine learning approaches to challenges dealing with biological (genomic) data. Much effort has resulted in machine learning techniques applicable for dealing with genomic data in the sense of reading, storing, learning and analysing it. A common focus of such projects has mainly been towards creating, improving and optimizing one or more models for a specific case. Thus, because the project goal is closely related to the machine learning goal of learning and prediction data with a highest possible accuracy.

Little work, if any, has yet been focused on bringing such already known techniques together for building a general purpose tool for dealing with the challenges in a broader and more general sense, in the context of genomic data. Such a tool may be beneficial for users with little or no programming skills in order to perform dynamic machine learning investigations and seek answers to their specific questions. The potential power of machine learning might be hidden for users with none or little knowledge about it.

Focus and challenges

Challenges of building a generic and flexible machine learning application has been the following:

- The transformation and representation of genome (genomic) data in a way which enables standard machine learning algorithms to work on it.
- The adaptation of a tool implementation, within an already complex and existing framework.
- The adversity of building a tool which bridges the fields of machine learning and biology when not having dealt with any of it previously.

Covered topics

Topics of special interest, which are devoted extra attention are:

- How to build a machine learning tool, having the flexibility and power to solve a wide range of both current and future biological challenges.
- Creating measures which capture genomic data and which are both flexible and reusable in multiple genomic data contexts.
- The treatment of the enormous available amount of data, when dealing with sparse data (rare cases) and skewness (imbalanced data).

Uncovered topics

While the thesis deals with the already mentioned topics, the work does not involve developing any new algorithms, nor adjusting or extending any existing ones. Nor does it try to fit algorithms to already available feature data, or try to select algorithm to best-fit a given feature set.

Though some results are presented, the focus is only to explain general concepts or uses of mentioned algorithms and techniques. It has never been the intention to optimize results or fine-tune algorithms only for performance purposes.

Method

Extensive research has been done searching through online resources and libraries to find material on (prior) work within the field of data mining, machine learning and bioinformatics. Access to the libraries has been authorized by The University of Oslo[1].

To get hands-on experience with machine learning applications, an online course in machine learning[2], has been completed while writing the thesis. Five increasingly challenging tasks was early solved with the focus of learning both statistic linear- and multiple-regression as well as the development-framework where the development took place. Concretely, the tasks were in general solved by using `R` and `rpy`[3]. Each of the task results was displayed and described at a dedicated wikipedia website for internal use.

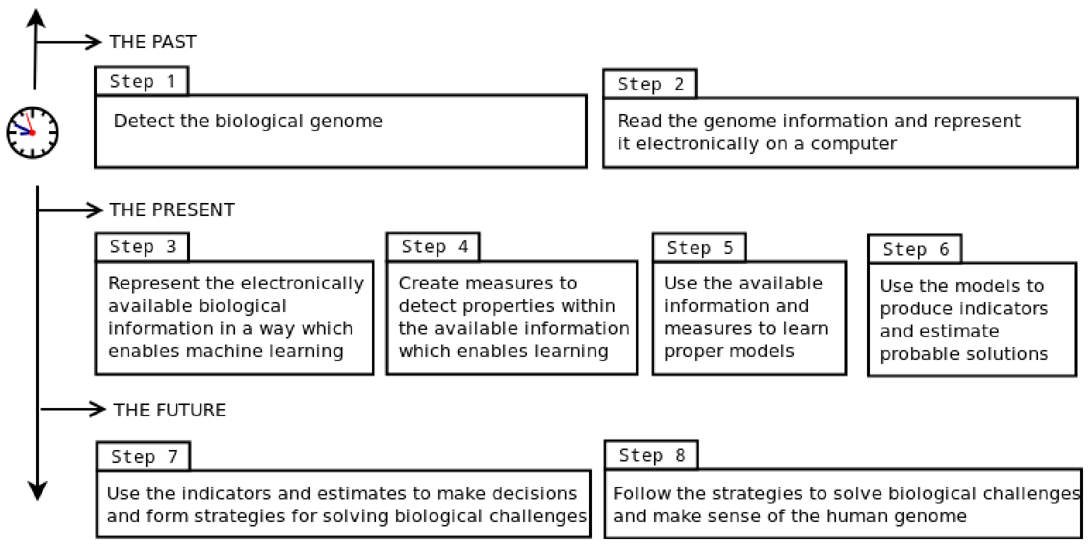

THE PAST
Step 1
Detect the biological genome
Step 2
Read the genome information and represent it electronically on a computer
THE PRESENT
Step 3
Represent the electronically available biological information in a way which enables machine learning
Step 4
Create measures to detect properties within the available information which enables learning
Step 5
Use the available information and measures to learn proper models
Step 6
Use the models to produce indicators and estimate probable solutions
THE FUTURE
Step 7
Use the indicators and estimates to make decisions and form strategies for solving biological challenges
Step 8
Follow the strategies to solve biological challenges and make sense of the human genome

Figure 1: An abstract 8-step methodology to make sense of the human genome.

The work of the thesis focuses on the 4 inner steps (3,4,5 and 6) of the abstract 8-step methodology for meaking sense of the human genome, shown in figure 1. The first covered topic (of building a machine learning tool) aims to investigate step 4, while the second topic (of creating measures) aims to investigate step 5. Step 6 and 7 are investigated through building an application for learning and

[1]The University of Oslo, website: `https://www.uio.no/` (2013-04-29)

[2] *Machine Learning* (`https://www.coursera.org/course/ml` (2013-04-29)) course, by Andrew Ng.

[3] The `rpy` is an extension module (binding) of the programming language `R` and `Python`.

making estimates. Furthermore, the methodology relies on the accuracy of the existing finds and representation of genomic information.

The application development has throughout the thesis been implemented in `Python`[4] using the «extreme programming» methodology without the pair-programming part. Test driven development (TDD) has been followed on a case-to-case basis. It has been a firm focus on unit testing key components with `Python`'s built-in `unittest` package. Frequent releases has been followed by solving multiple minor problems, which later has been combined and refactored whenever it was found to be necessary. My supervisor has played the role as an active customer; asking for functionality which has directed me towards building a beneficial end product.

The application implementation relies on three `Python` libraries, namely `numpy`[5], `scipy`[6] and `sklearn`[7] (SciKit).

Modelling has mainly been done using the `Dia`[8] program for creating UML class diagrams, and Graphviz `dot`[9] for other illustrations.

Environment

Large scale analysis and simulations requires a solid and powerful development environment. Development, implementation and execution of tool functionality has been done on the infrastructure and installation of The Genomic HyperBrowser[10] (HyperBrowser)[34]. It is based on the Galaxy project[11], which is built for interactive large-scale genomic analysis. The HyperBrowser developer community is continuously improving and building new tools for genomic analysis, in cooperation with statisticians and biologists.

The HyperBrowser executes on the *invitro* computer-cluster, which is a subset of the Abel[12] computer cluster, located in Norway. The cluster was ranked (in June 2012) as the 96th most powerful computer system in the world according to Top500[13]. It has more than a total of 10000 cores (CPUs) and 40 TebiBytes of memory.

Application access and usage

The application is publicly available at `http://hyperbrowser.uio.no/ml/`. Anyone who finds this work interesting is encouraged to use the application on their biological challenges.

[4]Python (version 2.7.3), website: `http://www.python.org/` (2013-04-29)

[5]Numerical python, website: `http://numpy.org/` (2013-04-29)

[6]Scientific Python, website: `http://scipy.org/` (2013-04-29)

[7]SciKit (version 0.12), website: `http://scikit-learn.org/` (2013-04-29)

[8]Dia website: `http://projects.gnome.org/dia/` (2013-04-29)

[9]Graphviz website: `http://www.graphviz.org/` (2013-04-29)

[10]The Genomic HyperBrowser website: `http://hyperbrowser.uio.no` (2013-04-29)

[11]Galaxy website: `http://galaxyproject.org/` (2013-04-29)

[12]Abel homepage: `http://www.uio.no/english/services/it/research/hpc/abel/` (2013-04-29)

[13]The Top500 is an organization which ranks the worlds most powerful computer clusters at their website: `http://www.top500.org/` (2013-04-29)

Chapter 1

Background

1.1 Machine Learning

Machine learning techniques [1, 42] has been increasingly popular over the last decades, due to the large amounts of available data («big data»[21]) and the access to freely available tools, e.g. Hadoop[1]. The field of machine learning offers many multi-purpose algorithms for operating on both small, large and huge datasets. In addition to this, many smart processing approaches has been proposed. Machine learning can also be viewed as extracting knowledge from data. However, the objective is not to store it, but to detect and use patterns for prediction purposes.

The key idea is to make a machine (computer) *learn* a model (hypothesis) by enough data of a given type, so it becomes able to identify one or more patterns within it. Identified (learned) patterns may then be used for making estimates (predictions) on yet unseen data of similar type as the data which was used to learn the pattern. The amount of required data may vary based on the difficulty of the pattern to learn. The learning process is often referred to as *training*, while the process of making decisions is called *classification*.

There are mainly two types of learning. The first type, when data is given to the computer in addition to directly pointing out the pattern answer, is called *supervised learning*. The second type, when no such out-pointet answers are given, is called *unsupervised learning*. Sometimes, unsupervised learning is performed while providing answers at a later stage in the process to make adjustments or fine-tune one or more parameters. This is called semi-supervised learning, since it is a combination of the two main types. Notice that other machine learning variants of the types do exists (e.g. reinforcement learning), but are not discussed in the thesis.

1.1.1 Supervised Learning

Supervised learning is to learn an hypothesis (model) using «answers» to help the machine figure out patterns. By this, the patterns to be learned is assumed to be known when the learning process begins. For the learning to have any meaning,

[1]Hadoop website: `http://hadoop.apache.org/`

there must be at least one pattern to learn. Thus, the outcome of all instances (samples) could either represent the presence or absence of the pattern.

The supervision part, is (for each sample) to «tell» the machine if a pattern is present or not. A sample instance where a pattern is present is denoted a positive sample. Equally, a sample where a pattern is not present (absent) is denoted a negative sample.

As an example, imagine that a dog is trained to find drugs hidden in a vehicle. The goal is to make it bark whenever a positive sample, e.g. a car with drugs, is presented to it. The training process is done presenting the dog with enough cars with drugs and reward it accordingly with some great candy whenever it barks when discovering drugs. In the same way, the dog may be «punished» when barking whenever drugs are not present in a car, by doing something which is unpleasant for the dog but not harmful. The supervision is done by stimulating the dog to respond to positive samples by rewarding it, and equally not to respond to negative samples by «punishing» it. Hopefully, the dog then obtains a built-in feeling (hypothesis) for barking whenever drugs are present in various vehicles. The evaluation process is done presenting the dog with a car which has not been used in the training process. By this, it is regarded as unknown to the dog. But, the dog should by its built-in feeling for what a vehicle with drugs smells and looks like, be able to transfer this knowledge to the presented car. The dog must then consider and decide whether or not it wants to signal the presence of drugs by barking.

1.1.2 Binary- and multi-class classification

A model (or hypothesis) which is used to predict two outcomes is known as binary classification. For instance, it may predict or classify a sample to be either positive or negative. Multiclass classification is when there are more than two possible outcomes (classes). In general, a n-class classifier may classify n possible outcomes. There is no such thing as a one-class classifier ($n = 1$) since there is no classes to distinguish between.

In some cases a binary classifier may be used as a multiclass classifier. This is known as a «one-vs-all» or «one-vs-rest» classifier. The idea is to build a collection $\mathbb{C}$ of n binary classifiers (c), one for each class, and then select the i-th classifier which estimates the highests probability for a given sample x.

$$\arg\max_{x} c(x) = \left\{ c(x_i) \mid \forall j \, \exists \, c(x_j) \le c(x_i) \land c_i, c_j \in \mathbb{C} \right\}$$

1.1.3 Unsupervised Learning

Unsupervised learning encourages the computer to figure out patterns by itself and learn an hypothesis without explicitly pointing out any answers for it. Such learning is particularly good in discovering segments within a data set, often by exploring relationships between huge amounts of data (big data). Examples of such segmentation could be detecting customer groups for targeted marketing or discovering solar system relationships. Applications which applies techniques from this field are usually somewhat related to artificial intelligence.

As an example, imagine that a soccer team wants to order new shirts for their players. To save costs and the effort of measuring the players *exact* shirt-size, they have found a company which delivers the shirts they want, by only getting the height and weight of each player. The samples are shown (as points) in figure 1.1.A. The company have three types of shirt-sizes, namely small (S), medium (M), and large (L). This corresponds to three segments. The company runs the given data of heights and weights on their unsupervised learning algorithm, asking it to return the samples as a group of three segments. The three segments (S, M and L) are shown in figure 1.1.B.

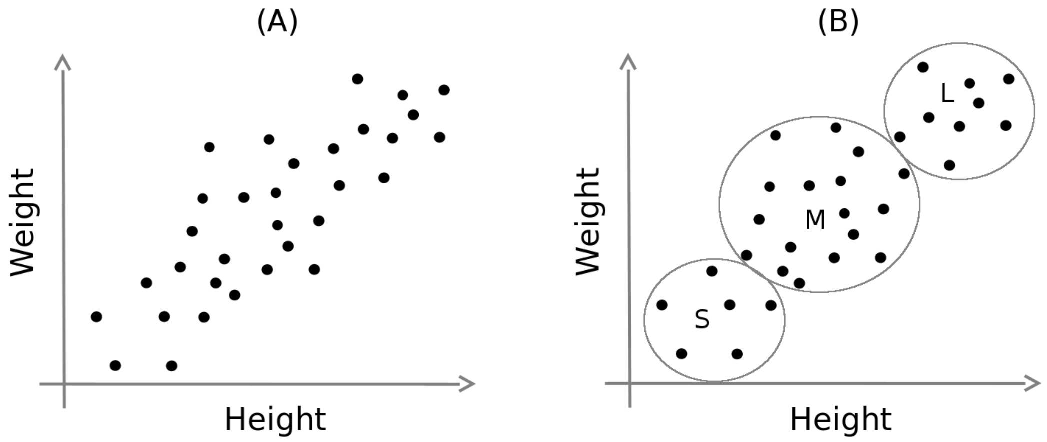


Figure 1.1: Segmentation of shirt-sizes for a soccer team. The dataset, illustration and the idea is partly taken from the machine learning course at Coursera.

1.1.4 Data mining

Data mining [37, 42] is used to detect patterns in data through analysis, to gain new insights. Patterns are interesting because they reveal information about trends and behaviors amongst the data, which are usually not visible by looking at the data separately. Thus, the insights reveal themselves when information is put together in a broader context. The insights reveal information «within» the information.

The data to be used in such an analysis should therefore, at least, be assumed to be a carrier of such information. Usually, the more data available at hand - the more precise the prediction model becomes. In addition, a data mining analysis should per definition be automatic[2], to be sure of that no man has added any interpretation. An analysis should then focus on carefully describing a series of the wanted mining steps, before execution time.

1.1.5 Concept

A specific pattern of interest is sometimes referred to as a concept or class. A machine learning concept [40, 42] can be viewed as a learned computational

[2]Automatic analysis usually means computer-driven.

understanding. It is a pattern which repeatedly appears or sticks out from the rest of the data. A concept is usually desirable and its discovery the goal of an analysis.

Usually, it proves difficult to fully represent a concept, due to the imperfection and variety of the world we live in. The goal, as in regression analysis, is to approximately estimate a concept as good as possible. If so, then there is a good chance of predicting the presence or absence of a concept in yet unknown data.

A concept may be represented differently by the algorithms that learns it, because of the way the algorithm is implemented. However, in abstract terms, a concept remains the same. How an algorithm represents it and how it is learned is much less important than how well it may be predicted in yet unseen data.

In statistical terms, a concept is a dependent variable on the independent variables inside the data. The explanation of a concept is learned by certain characterizations or measures of the concept, created from the data, and is known as features.

1.1.6 Features

When a machine learning algorithms learns a concept, it relies on learning from *features* that represents the data in a manner which explains its nature. Such features are usually created according to the data to be learned, on a case-to-case basis. Thus, features are used to distinguish between concepts.

Even though features are vital for any algorithm to be able to make good predictions, it is not always intuitive to know what a good feature looks like. Thus, creating proper features is a key challenge.

Feature example

A newly wedded couple enters a bank to ask for a loan of 2,000,000 NOK to buy the house of their dreams. The banker behind the desk looks through their financial situation, as displayed in table 1.1, in order to estimate (and decide) if the couple is able to repay the loan or not. Each column may represent the corresponding feature so that F_1 = Savings, F_2 = Yearly income, F_3 = Yearly car costs and F_4 = Private loan. The two possible outcomes ($\mathbb{C}$) are to give the loan ($c_{\text{yes}} \in \mathbb{C}$) or not ($c_{\text{no}} \in \mathbb{C}$). The banker may use a (machine) learned model h, which is learned from collected historical data of other customers and their repayment capability using the same features. The model is in this case represented as an vector $h = [1,4,-2,-1]$. The table row (sample) could also be represented as a vector $x = [150000, 550000, 25000, 350000]$.

Savings (F_1)	Yearly income (F_2)	Yearly car costs (F_3)	Private loan (F_4)
150000	550000	25000	350000

Table 1.1: Feature example of the financial situation of a newly wedded couple.

$$\begin{aligned}
maxAmountToLend(x|h) &= x_1 \times h_1 + x_2 \times h_2 + x_3 \times h_3 + x_4 \times h_4 \\
&= x_1 \times 1 + x_2 \times 4 + x_3 \times -2 + x_4 \times -1 \\
&= 150000 \times 1 + 550000 \times 4 + 25000 \times -2 + 350000 \times -1 \\
&= 150000 + 550000 \times 4 + 25000 \times -2 + 350000 \times -1 \\
&= 150000 + 2200000 - 50000 - 350000 \\
&= 1900000
\end{aligned}$$

$$giveLoan(x, sum) = \begin{cases} c_{\text{no}}, & \text{maxAmountToLend(x) < sum} \\ c_{\text{yes}}, & \text{otherwise} \end{cases}$$

The calculation result `maxAmountToLend` show that the maximum amount to lend the couple is 1,900,000 NOK. However, it does not look like the banker is able to give the newly wedded couple the loan to buy the house of their dreams since `giveLoan(x, 20000000)` returns negative (c_{no}).

To underline the importance of selecting "proper" features, imagine that another model h' was learned based on a different set of features. Concretely, only using the three first features of the ones used in h (without the private loan feature, F_4). Then, h' would not have taken the major drawback of the -350,000 NOK in prvate loan into account. As a result, h' might be more generous towards the newly wedded couple, since all of their other features are more positive than negative, in general.

1.1.7 The learnable

When dealing with patterns and concepts, the question of whether it is possible to learn a given concept arises. A learnable concept has been defined by Valiant [40] by following these criteria:

1. It must be characterized in order to know how to what situations it applies.
2. It must be computable in a feasible amount of steps. It should also be computable in polynomial time. If it is not computable in reasonable amount of time, it is per definition not learnable.
3. It must be non-trivial and for general purpose knowledge. It should be valuable and handle at least some variety of situations.

1.1.8 Data partitioning

Learning an hypothesis h and measuring its performance, usually makes use of data from the same source of origin. The data is therefore often partitioned in a beneficial manner which seeks to optimize the performance of h. Often, the total amount of available data is partitioned into 80% training data and the remaining 20% to test data. Both partitions should represent a relatively equal distribution of samples, so that training data may be verified by the test data.

In situations which requires tuning of certain parameters, an additional cross-validation partition may be beneficial. Its role is to evaluate and fine-tune parameters of h to improve the fitting of the training data. Afterwards, the test data may (as usual) be used to predict its performance. The partition sizes are then usually 60% training data, 20% cross validation data and 20% test data.

1.1.9 Learning curve

A learning curve is a graphical tool for evaluating a learning progress based on the amount of added samples. It is often used in error analysis when optimizing a model. Typically, the progress is evaluated for a every k added samples. Thus, if $k = 1$ the learning progress would evaluate the hypothesis for every single sample. Usually, k is selected as a larger number on large sample sets to reduce computation time.

The learning progress usually targets the evaluation score of a learned hypothesis on both the added training samples and on the cross-validation set. Ideally, the prediction error in both the training and cross-validation set would approach 0 by adding «enough» samples. If the graph fails to meet this objective, it indicates that the sample set contains unclear representations of its concepts.

In addition, the visualization helps interpreting the results of the learning progress. Consequently, it may imply *how* the dataset would perform if more samples or more accurate samples were to be used. Futhermore, adjusting learning parameters such as the regularization parameter (for smoothening decision boundaries) could be another alternative to improve the learning process.

Figure 1.2 show a learning curve with 100 ($k = 100$) intervals. It seem that adding increasingly more samples are lowering the cross validation error, but is gaining more tarining error at the same time. Furthermore, the lines (error rates) are moving towards a common error rate, by the amount of samples that are added. Thus, adding even more samples (if possible) could be fruitful path to follow. Another path could be to inspect the mis-classified samples and create features that target these specifically.

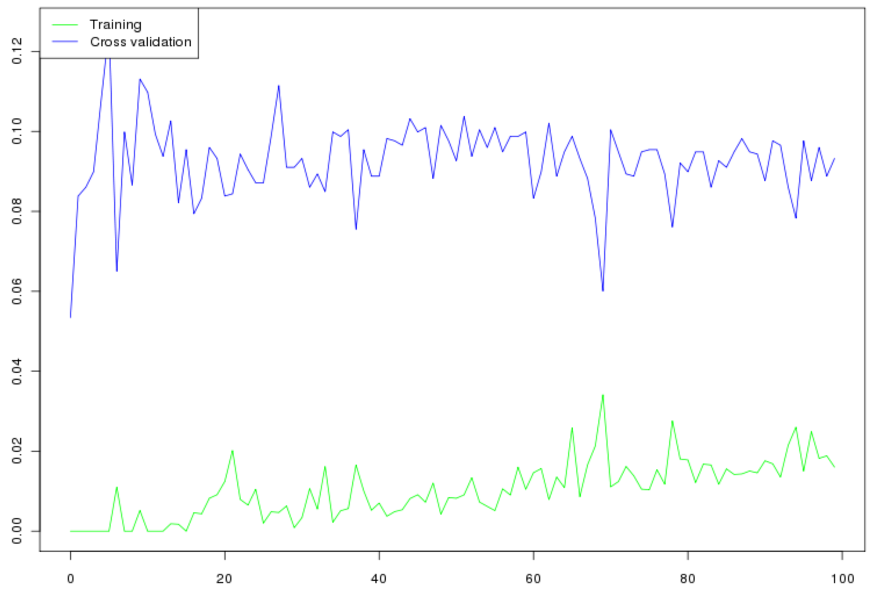


Figure 1.2: A learning curve illustration.

1.2 Bioinformatics

Computer science, dealing with biological challenges is known as bioinformatics. Biologists and computer-scientists work together, using computer power, to gain insights of how the human body operates (internally). The insights might be used to create even better medicine to cure or prevent diseases. A key challenge has been to figure out what normal DNA looks like. Having proper understanding of what normal DNA is, facilitates detection of anomalies and changes.

1.2.1 DNA and the human genome

The human genome consists of about 3 billion base-pairs, known through the language of DNA (deoxyribonucleic acid) which consists of 4 bases, namely A, C, T and G as shown in figure 1.3. The genome contains all our genes. More precisely, it contains the alleles which codes for the genes, but may differ on base-pairs, due to changes such as mutations.

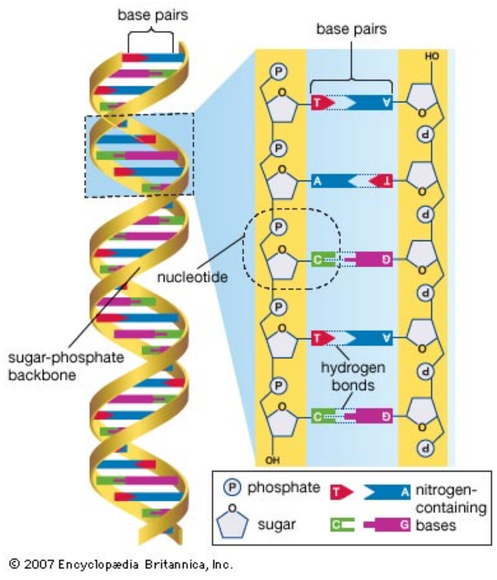


Figure 1.3: The DNA bases are adenine (A), thymine (T), guanine (G), and cytosine (C). Image taken from Encyclopedia Britannica.

1.2.2 DNA sequencing

DNA sequencing [25] is the process of *reading* biological material and translating it into a computer readable data representation which may be used by scientists and researchers for a multiple of analytical purposes. The sequencing process is complex and introduces many challenges such as gaps between reads, lack of coverage and various other sequencing errors.

In 2001, a draft sequence of the human genome[27] was published for the first time using a technique based on the one introduced by Sanger in 1977 [35]. Earlier, only fragments of smaller reads had been examined. The larger continuous read (in 2001) created an overview or landscape [26], enabling prior fragments to be put into a context. Such a context is still of great use in many areas, such as detecting gene transcription, chromatin structure, genetic variation, association to inherited diseases and more.

The following decade after the draft sequence breakthrough, more precise sequencing techniques was formed, resulting in increasingly large amounts of available reads, as well as more accurate reads. In 2004, the read of 2001 was greatly improved, covering ~99,7% of the genome, having only 300 gaps. The first-generation sequencing technique introduced by Sanger was able to read about 25,000 bases in a week [26], having 250,000 gaps and ~90% genome coverage. In contrast, the current, third-generation, sequencing techniques makes it possible to read 250 billion bases per week, with a nucleotide errors rate of only 1/100,000. The field is continously trying to detect more effecient ways of performing sequencing, where «cloud» sequencing[38] is one of the newest approaches. It offers great scalability and «pay-as-you-go» solutions.

1.2.3 Reference genomes

A practical way of acquiring and assembling genomic information of whole genomes is often done by *genome assembly*. Genome assembly is the process of assembling multiple fragments (often millions) of DNA sequenced reads to one single continuous read. This is mostly done by comparing overlaps amongst the fragments and combining them in a smart way. In addition to the large number of fragment-comparisons, the accuracy of keeping track of *genome coordinates* (fragments positions) inside the genome is a huge challenge. It was, in fact, the work on genome assembly which led to the first draft sequence of the human genome in 2001[27], which created a reference genome for later genome-assemblies. Various assembly techniques usually achieve good accuracy, though the greatest challenge is in regards of speed.

Multiple techniques and software programs are currently available for performing genome assemblies. Each of them tries to propose a smart way of overcoming the huge task of fragment-comparisons (string comparisons), while at the same time aiming at increased accuracy. An assembly job is usually ran on a larger computer cluster with lots of resources. One of the most famous assembly strategies is the whole-genome shotgun[41] (WGS) strategy, used by e.g. PCAP[23] for assembling whole genomes.

The WGS strategy clones a given genome a fixed number of times, and breaks each of the clones into fragments through *shearing*. By this, the fragments (splits) may be read individually. Parallel sequencing of individual fragments leads to a tremendous increase of speed. The resulting sequence reads are then re-assembled by a computer, looking for overlaps in the reads. Ideally, all fragments should overlap in a manner which results in a single (correct) continuous read. Special cases such as gaps and sequencing errors are handled in final post-processing steps.

Genome assemblies has been used in many projects[5, 6, 11] which have aimed to build a human «reference genome». Such a genome reference should have specific information about the genes, the functional elements, known variations and more. In 2003, the «Human Genome Project» [5] identified 3,2 billion base pairs and about 25,000 genes, while the ENCODE[11] project identified functional elements in 1% of the human genome[10].

1.2.4 Genomic annotation tracks

Genomic annotation tracks are computer-readable files, formatted in a structured way to provide information about one or more features across a genome or piece of DNA. An annotation track usually specifies which features it represents, and provides meta-data to specify genome coordinates and more. The UCSC

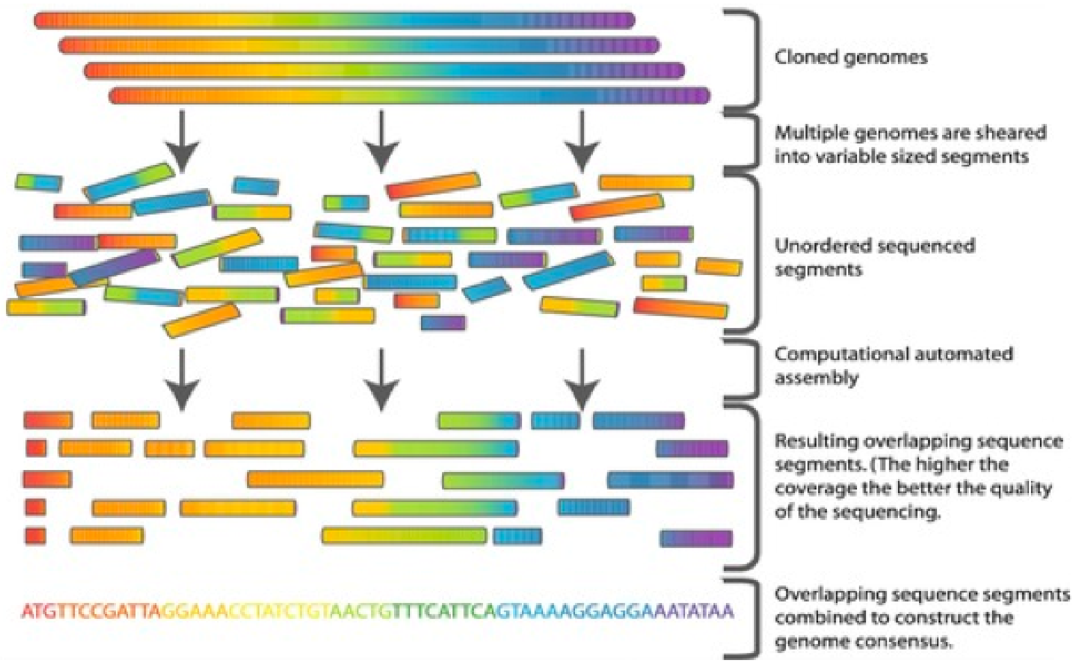

Cloned genomes
Multiple genomes are sheared into variable sized segments
Unordered sequenced segments
Computational automated assembly
Resulting overlapping sequence segments. (The higher the coverage the better the quality of the sequencing.
ATGTTCCGATTAGGAAACCTATCTGTAACTGTTTCATTCAGTAAAAGGAGGAAATATAA
Overlapping sequence segments combined to construct the genome consensus.

Figure 1.4: An illustration of the whole-genome shotgun strategy (WGS). Image taken from Commins, J., Toft, C., Fares, M. A. CC-BY-SA-2.5, via Wikimedia Commons.

(University of California, Santa Cruz) genome browser[3] is an open source, interactive, browser for inspecting and querying data based on annotation tracks, amongst other functionality. It was created for the Human Genome Project[5] and was used to compute the draft sequence of the human genome in 2001[27]. The browser offers a huge database of annotation tracks, and enables users to both download and upload annotation tracks and inspect the tracks with some provided visualization tools. The ENCODE project has systematically mapped regions of transcriptions, transcription factor associations, chromatin structures and histone modifications[14], using annotation tracks, to assign biochemical functions for 80% of the genome.

A large number of data formats exists due to researchers need to represent, arrange and store their content. By this, features have been represented in a multitude of ways through various data formats.

1.2.5 Data formats

DNA sequencing information of a whole genome is often stored as a single file on a computer. The file may also contain meta-information like positions of chromosomes and genes, depending on the data-format specification. There are many formats, but the most popular are BED[32] and WIG[24].

In 2011, the GTrack format[15] was published to represent many of the various formats uniformly. The format is carefully specified[4] to support an easy

[3]USCS genome browser, website: `http://genome.ucsc.edu/`.

[4]GTrack specification is avalable at `http://hyperbrowser.uio.no/hb/static/`

conversion from both BED, WIG and others using freely available tools at the Hyperbrowser's website[5].

1.2.6 Analyzing genomic data

Projects which deals with genomic data, usually analyze the representation formats (e.g. annotation tracks) in one way or another. For instance, the UCSC genomic browser allows anyone with educational (academic) purposes to freely download and possibly create their own tools if needed.

The Genomic Hyperbrowser is an online application, built for statistical analysis of large-scale genomic data. The process of using it to analyze a biological question, is illustrated in figure 1.5. The data representation («Data») uses two annotation tracks («Track 1» and «Track 2») and is represented in the GTrack format. The «Analysis» consists of selecting a proper statistic, possibly selecting a question to be answered and defining hypotheses. The «Result» may be interpreted by inspecting measurements such as correlation, p-value or others.

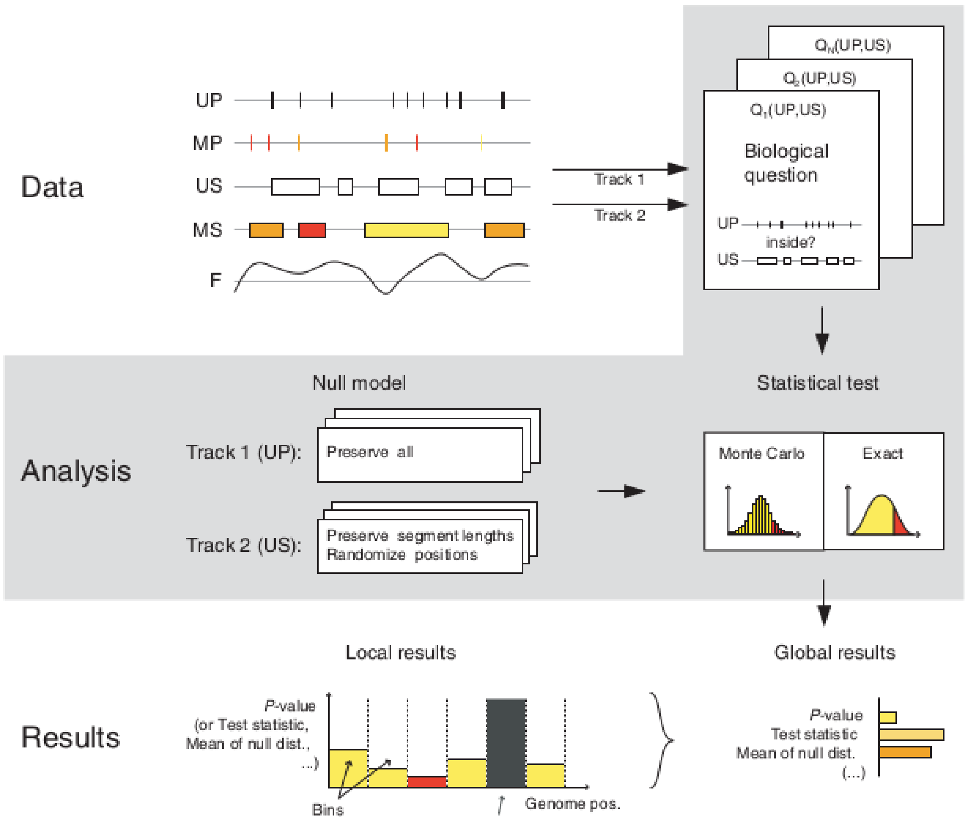


Figure 1.5: Illustration of analysing genomic data in the HyperBrowser. The image is taken from the HyperBrowser article[34]. It illustrates the three main Galaxy parts, namely «Data», «Analysis» and «Result».

hyperbrowser/gtrack/GTrack_specification.html.

[5]Left panel of http://hyperbrowser.uio.no/ contains GTrack tools.

1.3 Challenges and solutions in related research

In bioinformatics, much research has already applied machine learning techniques on genomic data and the human genome [28, 31]. Common to most of the projects, are the focus on optimizing performance within a single or specific context. A common challenge which often arises, is the challenges of «imbalanced data».

1.3.1 Imbalanced data

The imbalanced data problem [20] is the challenge of learning from data where there are more samples of a given class (or concept) than others. The imbalance between two or more classes are called *between-class* imbalance, while imbalance inside a given class is called *within-class* imbalance. The class with most examples are denoted the *majority class*, while a class with relatively few samples are denoted the *minority class*. The distinction between majority and minority classes are usually done when the imbalance reaches a certain (imbalance) *ratio*, e.g. 1:2, 1:10, 1:100, 1:1000 or more.

A high imbalance ratio tend to give majority classes a high accuracy ($\tilde{1}00\%$), while minority classes are left with much less accuracies ($\tilde{0}$-10%). A key challenge is to improve the accuracy of the minority classes without «hurting» (lowering) the accuracy of the majority class too much.

There are some common naming conventions to group «related» or «similar» imbalance problems together. If a ratio between majority and minority classes are (almost) constant when acquiring new samples, then the imbalance is *relative*. If acquiring more samples only increase the amount of samples of a majority class (leaving minority classes unchanged), then the balance is said to be *absolute*. Imbalance, due to the nature of the problem where the samples are gathered from, is called *intrinsic* imbalance. If a problem is not intrinsic, then it is *extrinsic*. If a minority class has a large enough sample data variety, it is a *rare case problem* and is a type of within-class imbalance.

1.3.2 Over and under-sampling

Two techniques for balancing an imbalanced data set are *oversampling* and *undersampling*[9]. When undersampling, some of the available samples from the majority class, are dismissed, in order to make the imbalance smaller. A potential downside of this, is loosing valuable samples for learning a subconcept of the majority class. When oversampling, the minority class samples are replicated by making some (randomized) small changes to feature values and adding the resulting «new» (synthetic) samples to the minority class. Oversampling may raise new challenges (or hinders) in classification, like *overfitting*.

1.3.3 Over and under-fitting

Heavily optimizing an algorithm may cause negative effects on predicting unseen data in a learned hypothesis h. If such cases, h is said to be *overfitted*[1]. Overfitting if often caused by learning a too specific model with small or no "margins" between the concepts to be learned; it does not generalize its hypothesis to fit unseen data. To prevent overfitting, more data (samples) is often used to train a more nuanced hypothesis. A second option is to introduce a *smoothening* parameter, known as the *regularization* value. A higher regularization value would then lead to increasingly larger margins between classes. If the margins becomes too large, then the possibility of under-fitting h arise. When a model is under-fitted[1], it leaves unnecessary large margins between classes, which could lead to bad generalization of h and thus hurt its performance.

1.3.4 Synthetic sampling and data generation

Replicated samples (from oversampling) are called *synthetic* samples because they try to mimic the absence or fill the void of a «thought» original sample. Creating synthetic samples for balancing out the minority and majority classes, has been approached in a multitude of ways, e.g. SMOTE, SMOTEBoost[4], ADASYN[19] or DataBoost-IM[16].

The Synthetic Minority Over-sampling TEchnique (SMOTE)[3] over-samples minority classes by adding synthetic samples. It claims that creating synthetic samples provides better results than replacing or adjusting existing samples.

The algorithm is inspired by the work on handwritten numeral recognition[17], where synthetic handwritten digits samples were created using various perturbation types (e.g. rotation) to achieve an increase in performance (e.g. prediction accuracy).

The SMOTE algorithm operates is *feature space* and not in *data space*, by adding synthetic samples after a dataset has been created. By contrast, adding more samples before sampling equals operating in data space. A synthetic sample S_{new} is created based on the k nearest neighbours of a sample S_i from the minority class. The sample S_{new} is then randomly created in-between S_i and the k nearest neighbours. The amount of samples to be created is based on a given percentage p. If $p = 100$, then 100% synthetic samples are created, meaning the minority class would double its size. Figure 1.6 displays a set of samples, where (a) is before and (b) is after (post) performing synthetic sampling.

The ADAptive SYNthetic sampling (two-class classification) algorithm (ADASYN)[19] is inspired by SMOTE, but offers an approach that focuses on improving sample balancing. It aims to both reduce «the bias introduced by class imbalance» and «shifting the classification decision boundary toward the difficult examples».

Balancing the samples are done by first calculating the degree of imbalance. If the degree is less than a set degree d, then the amount of samples needed to establish balance, are calculated. The amount of synthetic samples to be created for each of the minority samples are then carefully calculated by the use of a density distribution.

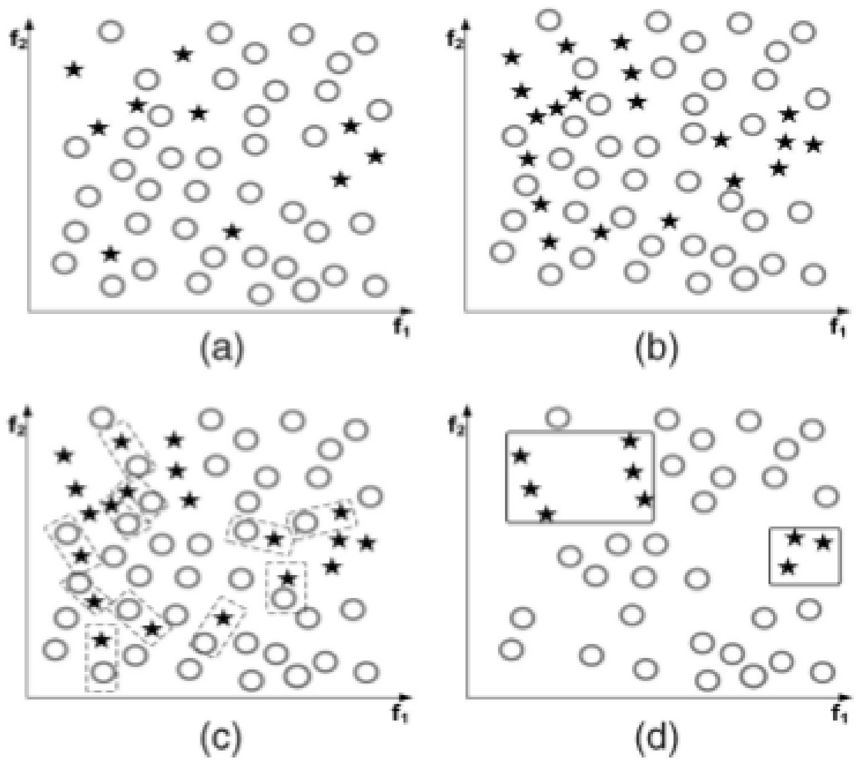


Figure 1.6: The image show (a) Original data set distribution. (b) Post-SMOTE data set. (c) The identified Tomek Links. (d) The data set after removing Tomek links. Image is taken from [20].

The tomek links data cleaning technique

Contrary to adding more samples, is reducing repetitive elements. Data cleaning techniques both aims to reduce less important samples and clean up misplaced (noise) samples or confusing border samples.

The tomek links[9, 20, 39] is an algorithm for detecting closely related n-vector pairs in $\mathbb{R}^n$ of different classes. A tomek link is a pair of samples from the full sample set, which has the smallest possible (euclidean) distance between them, and which has a non-equal class. Such a pair may exist by either being a misplaced (noise) sample, or a border-sample. Thus, the two aims of the data cleaning techiques are met. However, removing the pair would ease the classification. At the same time, it would introduce the risk of removing samples carrying critical classification information. Figure 1.6 displays the identification of such pairs in (c) and the sample set after their removal in (d).

1.3.5 Cost-Sensitive Learning

In contrast to editing a dataset, another approach called cost-sensitive learning has been proposed[20]. The key idea is that the cost for misclassifying a minority class sample could be given a higher cost, than misclassifying a majority class sample. A higher cost would result in larger margins «around» associated concepts by expanding or moving decision boundaries.

Currently, the work on this has only been case-specific. Cost-sensitivity improvements are often specific to individual learning algorithms, and little, if any, theoretical general-purpose improvements have yet been published.

Part II

Work

Chapter 2

Methods

A list of definitions may be found at the end of the chapter, at section 2.3.

2.1 Representing genomic data for use with machine learning algorithms

The human genome may be represented computationally in many ways and formats, e.g. as annotation tracks. The challenge is to «translate» available formats into a format which is both readable and understandable for machine learning algorithms, in order to make use of them. An essential step towards creating such a format is to use the common matrix representation which all algorithms understands. A matrix $\mathbb{S}$ may represent its m samples as rows, and its n measures as columns[42]. Notice that supervised learning algorithms use the last (n-th) measure column to store the class (or concept). The n measurements are then computed as the sum of the feature measures, possibly plus a response measure.

$$ {}_m\mathbb{S}_n = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,1} & s_{m,2} & \cdots & s_{m,n} \end{pmatrix} = \mathbb{S}_{m,n} $$

Furthermore, the representation of (human) genomic data in a matrix form relies on the definition of what a *sample* and a *measure* is. However, the smallest common representation in most formats deals with base-pair positions. Representing each base-pair, or its position, as a sample would then enable both learning and predicting the smallest entities, and probably be a safe and proper definition. Naturally, if samples represents baise-pair positions, the measures should focus on capturing its information.

2.1.1 Abstracting and grouping genomic data challenges

Even though genomic challenges may appear complex and unique, there is a chance that any challenge share common problem formulations with another genomic challenges. Grouping (abstractly) equal challenges together could thus be a done in a manner similar to computational complexity.

Computational complexity theory[2, 7] deals with grouping challenges of shared difficulty. A challenge may be «reducible» to a group, which means that the challenge is only «disguised» as another challenge. Thus, it follows that solving any one of these challenges will automatically solve all the others within the group.

By the same analogy as for computational complexity, genomic challenges may share common abstract problem formulations. Representing the genomic data in a generic way could be a tool to group equal abstract problem formulations together. The work on solving a common (group) challenge could then at the same time work to solve the generic challenge, abstractly.

The «worst case» scenario of grouping together (abstractly) equal challenges occur when there are no common challenges to be grouped. Grouping challenges into only separate groups would not do any direct harm, other than the effort of grouping them. On the other side, the potential of benefitting from such a grouping is potentially relatively large.

Another great benefit of reasoning about generic problem formulations is that it helps detect the underlying natures of the challenge without being distracted or limited by any difficulties of any concrete case. Concerns which may be constrained such a concrete case is thus not a hinder in the abstract context. Solid work on creating an abstract model will rather incorporate and express such concerns as «tradeoffs» between two or more options.

2.1.2 Strategy for representing samples and features uniformly

An electronic genomic data representation d, in a format such as an annotation track, could be «translated» into a matrix representation S. Any genomic data to be translated must specify its content in a structured way and provide genome coordinates. The genomic context of d could for instance specify the whole genome, a chromosome or a range of a chromosome. A translation strategy could be based on three principles:

1. Each position in the context of d is represented as an unique sample s (matrix row) in $\mathbb{S}$.
2. Every sample s ($\in \mathbb{S}$) must provide exactly one value v for each of the n assigned feature (and possibly class) measures.
3. When a response measure exists, the response column must contain at least two ($n = 2$) distinct values for $\mathbb{S}$ to be a learnable dataset.

2.1.3 Representing samples

A practical way of working with the multitude of electronic genomic data representations, is to represent the collection of various data format information uniformly. Such a representation enables working with the genomic data in a generic way, through defined structures.

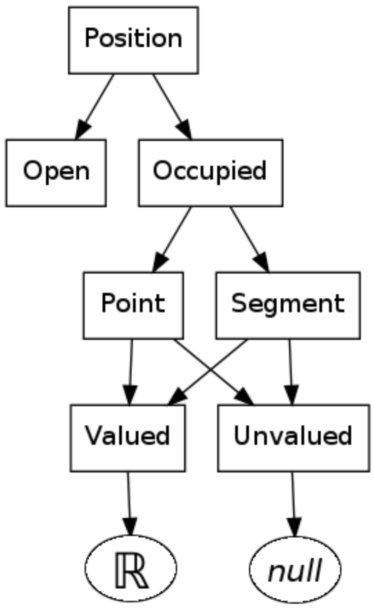


Figure 2.1: The position of a track and its possible values.

2.1.4 The track structure

A *track* T is an ordered collection of n subsequent, non-negative integers $\mathbb{N}_0 = \{0, 1, 2, 3, 4, ...\}$, denoting its members as *positions*. The theoretical upper bound of n is ∞ (infinity). The length of T is n, expressed as $|T| = n$. The first position, the *start position*, is always 0. By this, the set of positions equals the interval $[0, n-1]$ for $n > 1$, with the position at $n-1$ as the *end position*. If $n = 1$, then both the start and end position equal the very same 0 position. T is undefined for $n < 1$, since a track without any content does not make any sense, and certainly cannot be used to learn anything. T must have minimum one position, and at least one of its position must be *occupied*. A position k in a track T could be occupied by either a *point* or a *segment*, or is otherwise left *open*.

2.1.5 The track elements

A point or a segment are the only possible *elements* of a track T. Only one element may occupy a position at any given time, meaning two elements can never share a position k.

The *length* of an element equals the number of positions it occupies. A point can only occupy a single position. Thus, the length of a point is always 1. By contrast, a segment is a sequence of points, and may therefore occupy two or more connected, open, positions.

The *distance* between any two given elements is calculated by subtracting the end position of the preceding element by the start position of the subsequent element. By this, the distance between any two subsequent elements is always 0. Informally, the distance between two elements are the number of open positions between them.

An element e can either be *valued* or *unvalued*. In any case a value is assigned e. If unvalued, e is assigned the *undefined value* (*null*). If e is valued, a value v ($v \in \mathbb{R}$) is assigned to all the positions occupied by e.

2.2 Creating measures

Any given track T contains (by its definition) one or more elements. All elements have the properties of position, length and value. In addition, elements may have distance properties in regards to other elements. Thus, the challenge becomes to define measures which capture the available information afforded by the properties of T.

2.2.1 Capturing properties

The strategy of section 2.1.2 states that a sample s_i is created for every position i in a track T, providing a value $s_{i,j}$ for every j-th measure in the generated matrix representation ${}_{|T|}\mathbb{S}_n$ ($i \leq 1 \leq n, 1 \leq j \leq |T|, s \in \mathbb{S}$). Ideally, all «available» information should be accessable at any position i in T. A list[1] of standard properties could be:

- The current position i.
- A condition, determining if the current position is *open* or *occupied*.
- A condition, determining if the current position is a *point* or a *segment*.
- A condition, determining if the current position is *valued* or *unvalued*.
- The assigned value v.
- The *length* of the element which occupies the current position i. If i is open, then no element is present and the element length is undefined.
- The *distance* to the preceding element. If no preceding element exists, the distance is undefined.
- The *distance* to the subsequent element. If no subsequent element exists, the distance is undefined.

2.2.2 Using properties to create measures

A measure can make use of as many of the available properties as it desires. Any measure is only limited by the available properties and the creativity of the user. Designing «all» measures which *might* become useful is an overwhelming task. However, enabling dynamic creation of measures on a case-to-case basis might be a more fruitful path to follow.

2.2.3 Measurement utilization

Creating «good» features plays a key role in retrieving information from genomic data to learn concepts for classification. A good feature[18] collects unique data, which is not yet captured by any other feature (feature-feature inter-correlation), while having a high correlation to a concept (feature-class correlation).

[1]The list does not claim to be complete. More «available» properties may be added in the future.

The features are grouped based on their *nature*, which means that they act in a certain way which is common for other features within the same group. The groups are «distance», «value» and «condition».

Distance group
Features of the «distance» group deals with both distances between genomic elements (points and segments), as well as any other range related distances.

Value group
Features of the «value» group targets function tracks, where all positions inside the track is both connected, occupied an valued.

Condition group
Features of the «condition» group outputs only a discrete number of values. The theorethically minimum amount of values are two, in order to represent the presence or absence of a property.

2.2.4 Transformations

A *transformation* t adds an (optional) extra layer of flexibility to and reuse of a measure m. It is a function which enables the output of m to be changed dynamically, rather than statically (programatically). It is optional, because not all measures need to change its output. The purpose of t is to «map» a single real number x_{unmapped} to a corresponding real number x_{mapped} number, using a *mapper*. Concretly, if t is added to m, then any ouput x_{unmapped} of m is mapped into x_{mapped} before stored in the dataset $\mathbb{S}$.

$$x_{\text{mapped}} = T(x_{\text{unmapped}}) \mid x_{\text{unmapped}}, x_{\text{mapped}} \in \mathbb{R}$$

A *mapper* can either be *static* or *dynamic*, depending on if it's inner (mapper) variables is *regulated* by m. Variables used by the mapper of t may either be set *dynamically* or *statically*. A transformation is only static if all its variables are static. Otherwise, it is dynamic. A dynamic variable is a variable which may be set by m at runtime[2], where a static variable (by contrast) can not. A static variable must be *set* before runtime. A dynamic variable must allways have a specified default value, in case it is not set at runtime.

Regardless of whether a mapper is static or dynamic, the output must be a consistent equivalence relation to be reliable. All flexibility must be achieved by using a combination of the available structures, and not left to a single structure. Thus, a dynamic variable may not be changed after it has been used at runtime (in a way which changes this relation) during runtime.

There is no theorethical upper limit for how many transformations which could be added to m. Every transformation which is added to m, is to be chained. Thus, the output of one transformation would simply be the input of the next. By keeping track of the order of which n transformations are added, a mapping «pipeline» of all transformations would look like:

[2]Runtime is when the program executes.

$$x_{mapped} = \mathbb{T}_n(\mathbb{T}_{n-1}(\ldots \mathbb{T}_1(x_{unmapped}))) \text{ , where } |\mathbb{T}| = n$$

A transformation t could also be configured to *respond* differently to various *situations* and still uphold the equivalence relation. Thus, it is yet another way of affording flexibility. A *situation* s is a conditional evaluation offered by m for every output as a dynamic variable. Thus, t may (by its configuration) either choose to respond by either *executing* or *skipping* its mapping. Thus, it is possible to configure transformations to only execute under certain situations. However, t should execute in all situations by default if not explicitly configured otherwise.

2.2.5 Transformation utilization

Transformations may be generalized into groups by the nature of their operations, e.g. exponential, polyomial, etc. Operations should aim to enhance certain genomic behaviour, in a generic and flexible way.

Favouring transformation
A favouring transformation aim to favour a part of a genomic range over another part of the same range. The favouring may be beneficial when there exists insights in regards to how certain areas should be weighted over other areas.

Relativity transformation
A relativity transformation makes use of the mathemathical similtude property, for comparing values found in a single situation to another similar situation using a relatively equal scale.

Condition transformation
A condition transformation «clamp» a measurement range into a discrete amount of values. Thresholds are used to evaluate the selection of the discrete values.

Logarithmic transformation
A logarithmic transformation aim to improve help distinguis between distinct values which lie relatively close to each other. It aims to make the distance of relatively close elements smaller, while keeping relatively distant elements relatively further away.

2.3 Definitions

position A genomic coordinate (integer), specified as an offset inside a chromosome or the genome as a whole, depending on the *genomic context*. May be occupied by an *element* or is otherwise left *open*.

element A genomic element may occupy (be present) at a *position*. There are two types of elements, namely *points* and *segments*.

point A genomic *element*, which may occupy a single *position*.

segment A genomic *element*, which may occupy two or more *positions*. A sequence of *points*.

genomic context A specified genomic enviroment, e.g. the whole genome, one or more chromosomes or a *range* within a chromosome.

range An interval of all positions from a given *start position* to an *end position* within a genomic context. The *genomic context* selects the first position as the *start position* and the last position as the *end position* by default.

start position The first *position* of a range.

end position The last *position* of a range.

length The amount of positions in a range. Computed by subtracting the *end poisition* by the *start position*. Assumes the actual positions are from the same *genomic context*.

value A number $n \in \mathbb{R}$.

undefined value A specially set *value* (*null*), which denotes the absense of another *value*.

gap A gap is a series of one or more connected, open, positions between two elements (regardless of whether it is a point-point, segment-segment, segment-point or point-segment relation). The gap between two connected occupied positions are always 0.

Chapter 3

Implementation

This chapter describes the implemented machine learning *application* based on the methods and components of chapter 2. The application aims to offer a flexible toolset for solving generic and abstract genomic problem formulations, and thereby making sense of the human genome.

The application is integrated into the HyperBrowser as illustrated in figure 3.1, but is designed to be a stand-alone application. It is implemented in `Python` and interacts with the existing HyperBrowser framework by having a factory-inspired API. It relies on three `Python` libraries, namely `numpy`, `scipy` and `sklearn`.

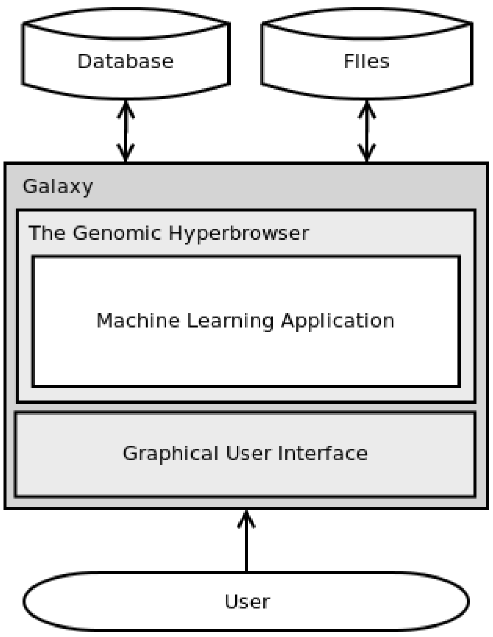


Figure 3.1: An illustration of the implemented machine learning application, fitted within the existing HyperBrowser.

The first section introduces a generic data format used as an annotation track for (internal) use within the application. Furthermore, the implemented structures for creating measures and samples from annotation tracks will be presented.

The second section introduces a genomic data representation technique for using the measures and structures to generate a dataset to be used by the application. The technique is both described and exemplified.

The third section introduces an adaptation of a selected set of existing machine learning algorithms within the application. The algorithms performance on the generated (genomic) dataset is illustrated.

The fourth section reason about the application design. It establishes design goals and guidelines while giving an overview of the various implementations.

The fifth section introduces a custom made programming language for enabling the application users to create, modify and thus optimize performance in learnability and accuracy by creating custom measures.

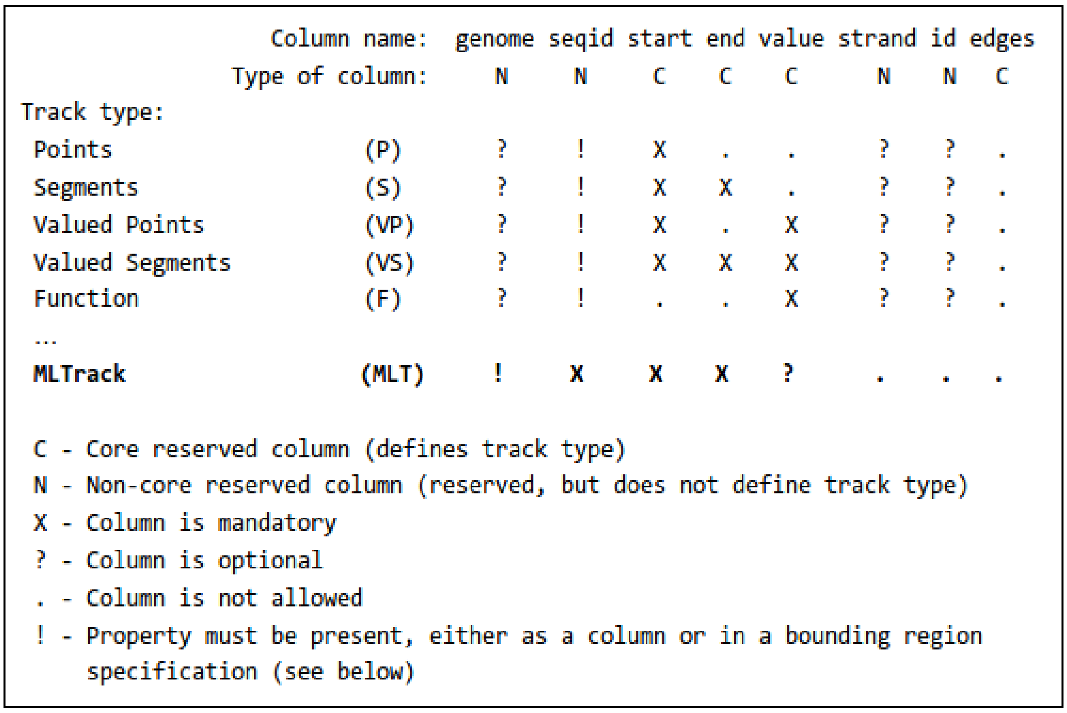
3.1 Machine learning components

The application to be presented is built using various components, which is defined in this section.

3.1.1 The machine learning track

A machine learning track (MLTrack) T implements the *track* structure presented in 2.1.4. It is implemented to be a generalization of the GTrack format. The track elements (points and segments) are therefore equivalently supported as in the GTrack implementation.

The representation of T uses a file-like notation for internal use, but it may also be useful for purposes of storage and sharing. All occupied positions of T are ordered by start position and written as separate lines. There are two types of lines, namely the «comment» line and the «data» line.



		Column name:	genome	seqid	start	end	value	strand	id	edges
		Type of column:	N	N	C	C	C	N	N	C
Track type:										
Points	(P)		?	!	X	.	.	?	?	.
Segments	(S)		?	!	X	X	.	?	?	.
Valued Points	(VP)		?	!	X	.	X	?	?	.
Valued Segments	(VS)		?	!	X	X	X	?	?	.
Function	(F)		?	!	.	.	X	?	?	.
...										
MLTrack	**(MLT)**		**!**	**X**	**X**	**X**	**?**	**.**	**.**	**.**

C - Core reserved column (defines track type)
N - Non-core reserved column (reserved, but does not define track type)
X - Column is mandatory
? - Column is optional
. - Column is not allowed
! - Property must be present, either as a column or in a bounding region specification (see below)

Figure 3.2: An overview of the eight reserved columns in the GTrack format and their associations to the different track type. The overview is a subset of Table 1 from the GTrack specification, with an additional row for the added MLTrack (in bold).

The comment line is optional, but may only occur once in any MLTrack file. If present, it must be the first line within the file and must start with four '#' signs (e.g. "####") followed by a minimum of one *key-value pair* separated with the '=' symbol. Multiple key-value pairs may be separated with the semicolon ';' sign. The commont line notation is, in fact, a generalization of the «bounding region specification line» (3.B) of the GTrack specification. By this, any valid GTrack notation will pass as a MLTrack notation, but not necessarily the other way around.

A data line must contain the four tabular-separated columns *seqid*, *start*, *end* and *val*. By this, the format may be seen as an extension of the GTrack format, since the following 5 GTrack formats (P,VP,S,VS,F)[1] may be mapped to the MLTrack. Concretely, since valued segments (VS) and functions (F) contain the four columns explicitly, they may be directly mapped only adding a proper comment line (header). The points (P) and valued points (VP) both lack the end column in the GTrack specification, but this may dynamically be created by the earlier definition that a point always have length of 1, meaning the end position is 1 more than the start position in the start column (which is present). Furthermore, the points (VP) and segment (S) is unvalued and may therefore be assigned the *null* value directly or equivalently omitted.

The formats (P,VP,S,VS,F) lays the foundation for learning any abstract relationships. Any explanatory track format could theorethically be related to any response track format. Consequently, a total of 25 (5^2) possible and abstract relationships exists.

Figure 3.2 on the facing page places the MLTrack in the context of the other mentioned GTrack formats. Note, that the function format is actually a special case of the valued point type, where all positions of a MLTrack T is occupied and none of the assigned values is the *null* value. The similarities or differences between two MLTrack's is usually represented as a MLTrack itself, based on the same reasoning as for GTrack. Note that the representation of the MLTrack does not guarantee to be a valid GTrack. In fact, the only situation where it is valid is in the case of segments (S) or valued segments (VS). Otherwise, if a MLTrack is not a valid GTrack, it should always be possible to reduce it into one (from the three remaining types P, VP or F. This is done by rearranging or removing content which is not supported by the GTrack specification. This is possible, because the mapping from GTrack to MLTrack is a *lossless* process, meaning all prior data may be recovered from the available data.

In the rest of the chapter, whenever a (explanatory or response) *track* is mentioned - the MLTrack is understood, if not explicitly stated otherwise.

MLTrack example 1: Unvalued points

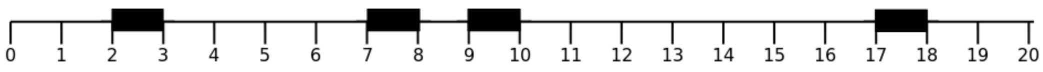


Figure 3.3: An example MLTrack of the unvalued points (UP) type.

Example 1 as GTrack

```
##gtrack version: 1.0
##track type: points
###seqid          start
####genome=hg18;start=0;end=20
chr1      2
chr1      7
chr1      9
chr1      17
```

Example 1 as MLTrack

```
####genome=hg18;start=0;end=20
chr1      2         3
chr1      7         8
chr1      9         10
chr1      17        18
```

[1]Allthough there are more GTrack formats than (P,VP,S,VS,F), the MLTrack only targets these.

MLTrack example 2: Valued points

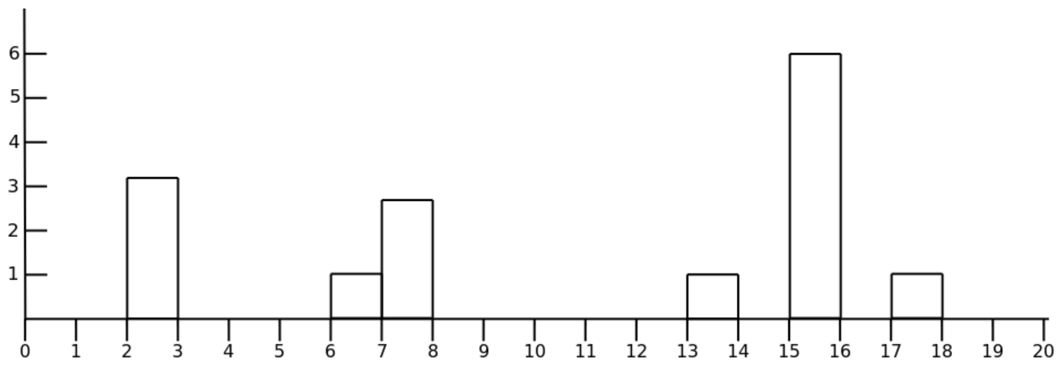


Figure 3.4: Visualization of an example MLTrack of the valued points (VP) type.

Example 2 as GTrack

```
##gtrack version: 1.0
##track type: valued points
###seqid          start    value
####genome=hg18;start=0;end=20
chr1      2        3.141579
chr1      6        1.0
chr1      7        2.718281828
chr1      13       1.0
chr1      15       6.0
chr1      17       1.0
```

Example 2 as MLTrack

```
####genome=hg18;start=0;end=20
chr1      2        3        3.141579
chr1      6        7        1.0
chr1      7        8        2.718281828
chr1      13       14       1.0
chr1      15       16       6.0
chr1      17       18       1.0
```

MLTrack example 3: Unvalued segments

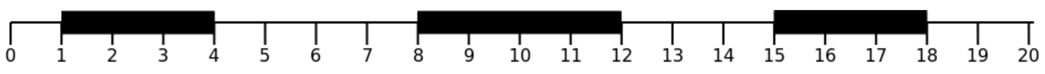


Figure 3.5: Visualization of an example MLTrack of the valued segments (VS) type.

Example 3 as GTrack

```
##gtrack version: 1.0
##track type: segment
###seqid          start    end
####genome=hg18;start=0;end=20
chr1      1        4
chr1      8        12
chr1      15       18
```

Example 3 as MLTrack

```
####genome=hg18;start=0;end=20
chr1      1        4
chr1      8        12
chr1      15       18       null
```

3.1.2 The machine learning track state

A machine learning track state (`MLTrackState`[2]) S gathers information from any given MLTrack T. S may be positioned at any position k in T ($0 \leq k < |T|$). Wherever S is positioned, it gathers and provides the «available» properties of the current position, according to section 2.2.1 on page 26.

3.1.3 The machine learning measure

A machine learning measure (`MLMeasure`[3]) M is inspired by the traditional way of reasoning about features (1.1.6), making use of the information provided by a MLTrackState S on a MLTrack T. The objective of M is to use the properties of S from T and produce a corresponding numeric representation of it, where equal information share relative or absolute number-equality. A machine learning feature (`MLFeature`[4]) F is a subclass of M and is used to generate explanatory data. A machine learning response (`MLResponse`[5]) measure R is a subclass of M as well, but outputs only numerical classes (representing concepts) so they may be either learned or predicted. Figure 3.6 shows the class relationship between M, F and R. Futhermore, the MLMeasure is implemented to enable adding an unlimited number of transformations.

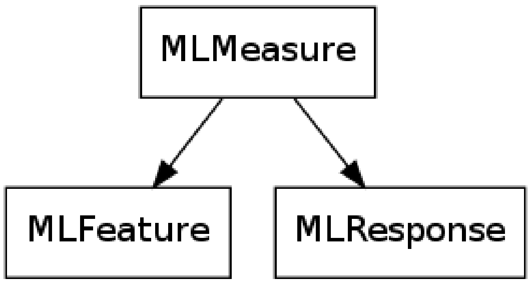


Figure 3.6: The relationship of the MLMeasure API and its two subclasses, namely MLFeature and MLResponse.

3.1.4 The machine learning transformation

A machine learning transformation (`MLTransformation`[6]) T implements the transformation structure introduced in 2.2.4.

To enhance the wanted transformation flexibility, any MLMeasure must (of interface reasons) have one or more transformations. To comply with this, there exists an empty transformation which implicitly is added if no transformations are explicitly added. The empty transformation T_{empty} simply outputs the input floating number.

$$x_{\text{unmapped}} = T_{\text{empty}}(x_{\text{unmapped}}) \mid x_{\text{unmapped}} \in \mathbb{R}$$

[2]Implementation details are available in appendix A.1.1.
[3]Implementation details are available in appendix A.1.2.
[4]Implementation details are available in appendix A.1.3.
[5]Implementation details are available in appendix A.1.4.
[6]Implementation details are available in appendix A.1.5.

Dynamic variables are implemented by allowing any transformation to read the «meta-data» of the measure it is attached to. A *meta-data instance* is a key-value pair, where the *key* is the (string) name of a specific dynamic variable reference and the *value* is the set value to be used by the mapper.

3.1.5 Situation dependence

The MLTransformation implements situation dependence by using the existing meta-data implementation. If no situation is explicitly offered by a measurement m, then the *all* situation is assumed, meaning the transformation would execute at all positions of a track. Two specific situations are implemented and used within the application, namely the *inner* and *outer* situations. The inner situation is set by m if a MLTrackState S is inside a segment. Otherwise, the outer situation is set if S is outside a track segment.

The inner situation
E occur (condition is met) whenever a MLTrackState S is inside an element at position k, and may be formally expressed as:

$$E(S,k) = \begin{cases} True, & \text{when } S \text{ is inside an element at position } k \\ False, & \text{otherwise} \end{cases}$$

The outer situation
E occur (condition is met) whenever a MLTrackState S is not inside an element at position k, and may be formally expressed as:

$$E(S,k) = \begin{cases} True, & \text{when } S \text{ is not inside an element at position } k \\ False, & \text{otherwise} \end{cases}$$

3.1.6 Combining transformations

A MLMeasure M may use a combination of a set of transformations $\mathbb{T} = \{T_1, T_2, \dots, T_{n-1}, T_n\}$. The transformations would then operate as a (transitive) transformation pipeline, mapping any given real number value in the order the transformations was added, taking any situation dependence into account. By this, it is possible to add transformations (with situation dependance) to the pipeline, and let the application take care of it. A transformation which is «skipped», corresponds to skipping a pipeline step, as illustrated in figure 3.7. The evaluation result of the pipeline may be expressed as:

$$x' = T_3(T_2(T_1(x))) \mid x, x' \in \mathbb{R} \mid T_2 \equiv T_{\text{empty}}$$

since T_2 is skipped (replaced by T_{empty}).The act of putting the transformations together is an act of design. It is left to the user to decide how many transformations and in what order. It is no maximum number of transformations which may be added, and an transformation may be added multiple times.

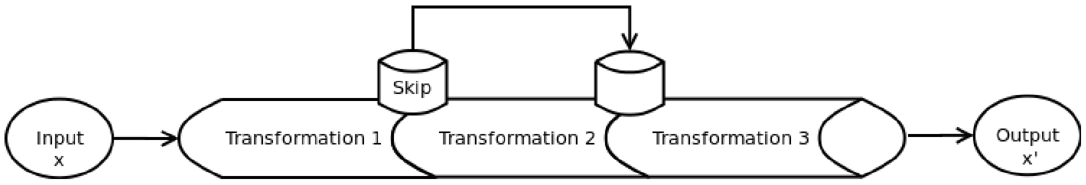


Figure 3.7: A transformation pipeline, of three transformations. The original input x is the measurement value, and x' is the transformed measurement value. The pipeline first executes «Transformation 1», since this "pipe" was the first one added. Then, the «Transformation 2» is skipped (possibly due to a situation condition). Finally, the last transformation («Transformation 3») transforms the output of «Transformation 1» and outputs the end result x'.

3.1.7 Feature measures

Generic feature implementations which use the available properties, offered by a MLTrackState S is presented in their natural groups, as described in section 2.2.3. Hopefully, these features may be used as a basis for creating «good» features on a case-to-case basis.

Distance features

The relative distance features[7] (`MLFeaturePositionRelative` and `MLFeaturePositionRelativeInverted`) aim to capture the samples relative position inside the length of the track (T) it operates on. It makes the assumption that related tracks share a mathematical similitude property. At any given position k, the feature value is computed to be the relative distance inside the track. The track length $|T|$ is denoted n.

$$F_{\text{DistanceRelative}}(k,n) = \begin{cases} \dfrac{k}{n-1}, & n>1 \\ 0, & n \leq 1 \end{cases}$$

$$F_{\text{DistanceRelativeInverted}}(k,n) = \begin{cases} 1-\left(\dfrac{k}{n-1}\right), & n>1 \\ 0, & n \leq 1 \end{cases}$$

The relative center distance features[8] (`MLFeaturePositionRelativeCenter` and `MLFeaturePositionRelativeSides`) aim to capture the samples relative position to the center of the track (T) it operates on. The distance to the sides is thus the inverse of the distance to the center. The center position is calculated by $\frac{n}{2}$, where $n = |T|$. It makes the assumption that related tracks share the mathematical symmetric and similitude property. At any given position k, the feature value is computed to be the relative distance from the center of T.

[7]See appendix A.2.2 for implementation details.
[8]See appendix A.2.2 for implementation details.

$$F_{\text{DistanceRelativeCenter}}(k,n) = \begin{cases} \dfrac{k}{n-1}, & k < \frac{n}{2}, n > 1 \\ \dfrac{n-k}{n-1}, & k > \frac{n}{2}, n > 1 \\ 0, & \text{otherwise} \end{cases}$$

$$F_{\text{DistanceRelativeSides}}(k,n) = \begin{cases} 1 - \left(\dfrac{k}{n-1}\right), & k < \frac{n}{2}, n > 1 \\ 1 - \left(\dfrac{n-k}{n-1}\right), & k > \frac{n}{2}, n > 1 \\ 0, & \text{otherwise} \end{cases}$$

Element-Element distance features includes two rather similar features based on the same idea. Both the «last distance» (`MLFeaturePointDistanceLast`[9]) and the «future distance» (`MLFeaturePointDistanceFuture`[10]) aims to capture the notion (distribution) of element-element gaps (distances). The feature is created based on the assumption that it may be of interest to have an numeric way of measuring how large the gap is between any position k and the closest element e surrounding it.

The gap between k and the last (subsequent) element l may be done computing the absolute difference $|k-l|$. It is also possible to compute the relative distance by dividing by the range length n. If no such element l exists, the undefined value is returned.

$$F_{\text{DistanceLast}}(k,l) = \begin{cases} |k-l|, & \text{if } l \text{ is defined} \\ undefined, & \text{otherwise} \end{cases}$$

$$F_{\text{DistanceLastRelative}}(k,l,n) = \begin{cases} \dfrac{|k-l|}{n}, & \text{if } l \text{ is defined and } n > 0 \\ undefined, & \text{otherwise} \end{cases}$$

Capturing the gap between the current position k and the future (preceding) element f may be done computing the absolute difference $|f-k|$. It is also possible to compute the relative distance by dividing by the range length n. If no such element f exists, the undefined value is returned.

$$F_{\text{DistanceFuture}}(k,f) = \begin{cases} |f-k|, & \text{if } f \text{ is defined} \\ undefined, & \text{otherwise} \end{cases}$$

$$F_{\text{DistanceFutureRelative}}(k,f,n) = \begin{cases} \dfrac{|f-k|}{n}, & \text{if } f \text{ is defined and } n > 0 \\ undefined, & \text{otherwise} \end{cases}$$

[9]See appendix A.2.2 for implementation details.

[10]See appendix A.2.2 for implementation details.

The closest outer element distance (`MLFeaturePointDistanceOuter`[11]) feature is a combination of the «last distance» ($F_{\text{DistanceLast}} = F_{\text{DL}}$) and «future distance» ($F_{\text{DistanceFuture}} = F_{\text{DF}}$) features. It aims to capture the minimum distance to the closest element from a given position k. A relative positioning could be achieved by dividing the result by the track length, if not undefined.

$$F_{\text{DistanceClosest}}(k,l,f) = \begin{cases} undefined, & \text{if both } l \text{ and } f \text{ is undefined} \\ F_{\text{DF}}(k,f), & \text{if } f \text{ is defined and } l \text{ is undefined} \\ F_{\text{DL}}(k,l), & \text{if } l \text{ is defined and } f \text{ is undefined} \\ min(F_{\text{DF}}(k,f), F_{\text{DL}}(k,l)), & \text{otherwise} \end{cases}$$

The element inner distance (`MLFeaturePointDistanceInner`[12]) feature aims to capture the distance inside of segments (only). Using the feature for points has no meaning, since a point has no inner length, resulting in a distance of 1. The inner distance of a segment is calculated as the distance away from the segments center position $c = \frac{n}{2}$. The segments start position s and end position e is used in the calculation, and which is accessible from the tracks state reader (MLTrackState).

$$F_{\text{DistanceRelativeCenter}}(k,s,e) = \begin{cases} \dfrac{k-s}{e-s}, & k < \frac{e-s}{2}, e-s > 0 \\ \dfrac{e-k}{e-s}, & k > \frac{e-s}{2}, e-s > 0 \\ 1, & \text{otherwise} \end{cases}$$

Value features

The slope (`MLFeatureFunctionSlope`[13]) feature aims to capture the slope at any position k. It computes the slope of the subsequent point s and preceding point p, assuming that they are connected.

The slope value is computed using the mathematical slope property, by $\Delta y = p - s$ and $\Delta x = 3$. At the start- and end-position of the track, the s and p are not available. In such cases, the slope is computed by using the available start- or end-point and the current point k. If neither of s and p is available, the undefined value is returned. The current possition k is assumed to always be defined and present.

$$F_{\text{Slope}}(k,p,s) = \begin{cases} \dfrac{\Delta y}{\Delta x} = \dfrac{p-s}{3}, & \text{if both } p \text{ and } s \text{ is defined} \\ \dfrac{\Delta y}{\Delta x} = \dfrac{p-k}{2}, & \text{if only } p \text{ is undefined} \\ \dfrac{\Delta y}{\Delta x} = \dfrac{s-k}{2}, & \text{if only } s \text{ is undefined} \\ undefined, & \text{otherwise} \end{cases}$$

[11]See appendix A.2.2 for implementation details.
[12]See appendix A.2.2 for implementation details.
[13]See appendix A.2.2 for implementation details.

The strand (`MLFeatureFunctionStrand`[14]) feature aims to capture the positive or negative property of the value v assigned to the element e which occupies any position k. If the point is unvalued, the feature value returns the undefined value. Though the feature fits perfectly amongst the «condition» features, it assumes that all of a tracks positions are occupied, and does therefore better fit with the «function» features.

$$F_{\text{Strand}}(e,v) = \begin{cases} 1, & \text{if } v \geq 0 \text{ and } e \text{ is defined and } v \in \mathbb{R} \\ 0, & \text{if } v < 0 \text{ and } e \text{ is defined and } v \in \mathbb{R} \\ undefined, & \text{otherwise} \end{cases}$$

Condition features

The «condition» group features are, in contrast to «distance» and «value» features, known by outputting a small number of values. The conditions are based on the information offered by a track state (MLTrackState). Other features may also make use of conditions, but the group of «condition» features are the pure group where output is solely dependent on conditions. A conditions is great for detecting specific properties, e.g. detecting the start- and end-position of a segment.

The segment start (`MLFeatureSegmentStartPosition`[15]) feature aims to capture the start position of a (segment) element e. If a given position k is the start of e, the condition is met, and the corresponding specified value 1 is returned. Otherwise, when the condition is not met, the value 0 is returned.

$$F_{\text{SegmentStart}}(k,e) = \begin{cases} 1, & \text{if } k \text{ is start position of segment } e \\ 0, & \text{otherwise} \end{cases}$$

The segment end (`MLFeatureSegmentEndPosition`[16]) feature aims to capture the end position of a (segment) element e. If a given position k is the end of e, the condition is met, and the corresponding specified value 1 is returned. Otherwise, when the condition is not met, the value 0 is returned.

$$F_{\text{SegmentEnd}}(k,e) = \begin{cases} 1, & \text{if } k \text{ is end position of segment } e \\ 0, & \text{otherwise} \end{cases}$$

3.1.8 Transformations

The implemented application transformations[17] are generic transformations based on section 3.1.4 which alltogether aims to enrich the flexibility of the features of section 3.1.7.

[14]See appendix A.2.2 for implementation details.
[15]See appendix A.2.2 for implementation details.
[16]See appendix A.2.2 for implementation details.
[17]See appendix A.2.3 for implementation details.

The favouring transformation family (`MLTransformationFavour*`[18]) implements the favoring idea of section 2.2.5 on page 28.

The transformation family members favour the left, right, center or both-sides (left and right) values of a range. They all rely on that a range of the genomic context is set as a dynamic variable. A range may either represent an element at position k, or a gap outside the surrounding elements if k is not occupied. The favouring may be calculated for a range by using the start position s and the end position e in addition to k.

$$T_{\text{FavourLeft}}(x|k,s,e) = \begin{cases} x + x \times \dfrac{e-k}{e-s}, & \text{if } e-s \geq 1 \\ x, & \text{otherwise} \end{cases}$$

$$T_{\text{FavourRight}}(x|k,s,e) = \begin{cases} x + x \times \dfrac{k-s}{e-s}, & \text{if } e-s \geq 1 \\ x, & \text{otherwise} \end{cases}$$

$$T_{\text{FavourCenter}}(x|k,s,e) = \begin{cases} x + x \times \dfrac{k-s}{e-s}, & k < \frac{e-s}{2}, e-s \geq 1 \\ x + x \times \dfrac{e-k}{e-s}, & k > \frac{e-s}{2}, e-s \geq 1 \\ x+1, & \text{otherwise} \end{cases}$$

$$T_{\text{FavourSides}}(x|k,s,e) = \begin{cases} x + x \times \left(1 - \dfrac{k-s}{e-s}\right), & k < \frac{e-s}{2}, e-s \geq 1 \\ x + x \times \left(1 - \dfrac{e-k}{e-s}\right), & k > \frac{e-s}{2}, e-s \geq 1 \\ x, & \text{otherwise} \end{cases}$$

The relativity transformation (`MLTransformationRelative` implements the relativity idea of section 2.2.5. It may be used to facilitate situations similar to mathematical similitude, where the lengths of ranges or elements are relative to each other with a magnitude m at any position k.

$$T_{\text{Relative}}(x|k,m) = \begin{cases} \dfrac{x \times k}{m}, & m > 0 \\ undefined, & \text{otherwise} \end{cases}$$

The logarithmic transformation (`MLTransformationLogarithmic`) implements the distinguishing idea of section 2.2.5. For instance, if two relatively close points lie close to a segment, the distinction between the two points will be smaller after applying such a transformation. However, points of relatively larger distance would not be as much affected. The logarithm could use a base n of 10 or any other positive integer.

$$T_{\text{Logarithmic}}(x|n) = \begin{cases} \log_n(1+x), & x \geq 0,\ n \geq 2 \\ undefined, & \text{otherwise} \end{cases}$$

[18]The favouring family consists of * = Left, Right, Center, Sides.

The condition transformation familiy (`MLTransformationCondition*`[19]) implements idea of discretization from section 2.2.5 on page 28. For all of the family transformations, a condition is evaluated using a threshold z. The input value v is evaluated to be exactly equal to, or less, or greater than z.

$$T_{\text{ConditionEqualsValue}}(x|v,z) = \begin{cases} 1, & \text{if } v = z \\ 0, & \text{otherwise} \end{cases}$$

$$T_{\text{ConditionLessThanValue}}(x|v,z) = \begin{cases} 1, & \text{if } v < z \\ 0, & \text{otherwise} \end{cases}$$

$$T_{\text{ConditionGreaterThanValue}}(x|v,z) = \begin{cases} 1, & \text{if } v > z \\ 0, & \text{otherwise} \end{cases}$$

The roundoff transformation (`MLTransformationRoundOff`) aims to «clamp» together values that lie close to each other. In situations where multiple values tend to align relatively close to each other, and may be treated as almost equal, a round-off[20] could, at a decimal position d, lead for instance two samples to share the equal (rounded) values. The decimal position d to round off at, is set as a dynamic variable, but may «fall back» on a default static value (e.g. 5) if not set.

$$T_{\text{RoundOff}}(x|p) = \begin{cases} round(x,p), & \text{if } x \geq 0,\ p \geq 0 \\ undefined, & \text{otherwise} \end{cases}$$

The polynomial transformation (`MLTransformationPolynomial`) raises a given value x to a given polynomial p. The polynomial p may be set as a dynamic variable, buy could default to a static variable (e.g. 1). A square transformation could be achieved by $p = 2$, and equally the cube transformation with $p = 3$.

$$T_{\text{Polynomial}}(x|p) = x^p$$

The angle transformation (`MLTransformationAngle`) computes the mathemathical angle of a given (slope) value x by the mathematical property of arcus tangent[21].

$$T_{\text{Angle}}(x) = \arctan(x)$$

The addition transformation (`MLTransformationAddition`) adds a given number a, to the value x and returns the sum of $a+x$. For $a < 0$ the transformation works equally good as a «subtraction» transformation. Note, that a must be a static variable not to break the equivalence relation.

$$T_{\text{Addition}}(x|a) = x + a$$

[19]The condition family * = EqualValue, LessThanValue and GreaterThanValue.

[20]The implmentation uses the `round` method of the `math` package in `Python`.

[21]The implmentation uses the `arctan` method of the `math` package in `Python`.

The product transformation (`MLTransformationProduct`) multiplies a given number a, to the value x and returns the product $(a \times x)$. For $a < 1$ the transformation could work equally good as a «division» transformation. Note, that a (again) must be a static variable not to break the equivalence relation.

$$T_{\text{Product}}(x|a) = x \times a$$

The exponential transformation (`MLTransformationExponential`) returns the exponential value of $x + 1$. This, because $x = 0$ is common for many features, and would lead to errors because the exponent is undefined for 0. For $x < 0$, the undefined value is returned.

$$T_{\text{Exponential}}(x) = \begin{cases} \exp(x+1), & \text{if } x \geq 0 \\ undefined, & \text{otherwise} \end{cases}$$

The square root transformation (`MLTransformationSquareRoot`) returns the mathematical square root of a given value x. For $x < 0$, the undefined value is returned. The base n is set to 2 (because of «square» root), but may be changed by replacing it with another positive integer.

$$T_{\text{SquareRoot}}(x|n) = \begin{cases} \sqrt[n]{x}, & n \geq 2,\ x \geq 0 \\ undefined, & \text{otherwise} \end{cases}$$

3.1.9 Response measures

The response measures «translates» a response track T into class representation. There are two levels of measurements, as illustrated in figure 3.8 on the following page.

The first level (L1) corresponds to a binary classification problem by only outputting a boolean value (True or False) based on whether a condition is met or not. Usually, such a condition checks the existance of an element at any given position of T.

The second level (L2) goes a step further than just checking the existence of an element, it also takes the elements assigned value into account. Recall, that an unvalued element is assigned the *null* value. Iff all elements in T only have assigned one single value (in addition to the *null* value), then the problem is (still) a binary classification problem. Otherwise, if there are n (two ore more) assigned values, it is a n-class classification problem.

There are 5 implemented response measures, namely the «point exists», «point value», «segment exists», «segment value» and «function value».

The point exists (`MLResponsePointExists`[22]) response measure is a L1 measure which returns *True* if a MLTrackState S is positioned at an occupied position k in a response track T. Otherwise, it will return *False* since the position is open.

$$R_{\text{PointExists}}(k,T) = \begin{cases} True, & \text{if position } k \text{ in } T \text{ is occupied} \\ False, & \text{otherwise} \end{cases}$$

[22]See appendix A.2.2 for implementation details.

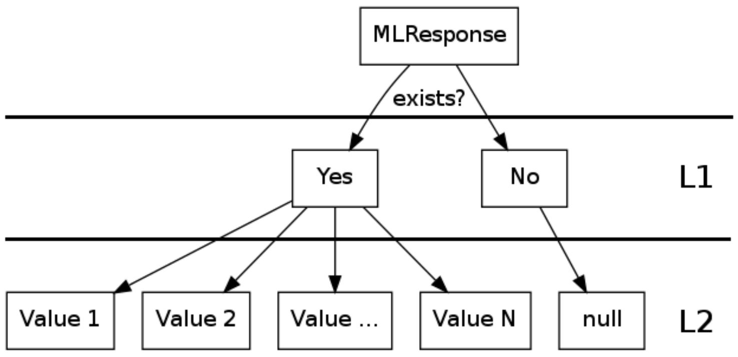


Figure 3.8: The two levels (L1 and L2) to measure response.

The point value (`MLResponsePointValue`[23]) response measure is a L2 measure and an extension of «The point exists» response, which aims to return the point value v of an element e at position k in a response track T, iff such an element exists. Otherwise, the undefined value is returned.

$$R_{\text{PointValue}}(k,T,e,v) = \begin{cases} v, & \text{if } e \text{ exists at position } k \text{ in } T \\ undefined, & \text{otherwise} \end{cases}$$

The segment exists (`MLResponseSegmentExists`[24]) response measure is a L1 measure which returns *True* if a MLTrackState S is positioned at an occupied position k inside a segment of a response track T. Otherwise, it will return *False* since the position is open.

$$R_{\text{SegmentExists}}(k,T) = \begin{cases} True, & \text{if position } k \text{ in } T \text{ is occupied by a segment} \\ False, & \text{otherwise} \end{cases}$$

The segment value (`MLResponseSegmentValue`[25]) response measure is a L2 measure and an extension of «The segment exists» response, which aims to return the value v of the segment e which occupies position k in a response track T, iff such an element exists. Otherwise, the undefined value is returned.

$$R_{\text{SegmentValue}}(k,T,e,v) = \begin{cases} v, & \text{if segment } e \text{ occupies position } k \text{ in } T \\ undefined, & \text{otherwise} \end{cases}$$

The function value (`MLResponseFunctionValue`[26]) response measure is a L2 measure which returns the value v at any position k in a response track T, by assuming that it is occupied. Otherwise, the undefined value is returned.

$$R_{\text{FunctionValue}}(k,T,e,v) = \begin{cases} v, & \text{if } e \text{ exists at position } k \text{ in } T \\ undefined, & \text{otherwise} \end{cases}$$

[23]See appendix A.2.2 for implementation details.
[24]See appendix A.2.2 for implementation details.
[25]See appendix A.2.2 for implementation details.
[26]See appendix A.2.2 for implementation details.

Selecting response measures

Selecting a track response measure is constrained by the track type. If a response track is unvalued, then only L1 measures are available. A track is unvalued if none of its elements are valued. The reason behind this constraint, lies in the fact that predicting values for an unvalued track would predict the *null* value at all positions, (which makes no sense). This, because both existence of an unvalued element and non-existence of an element would (at L2) returns the *null* value.

By contrast, valued response tracks can choose amongst both L1 and L2 measures. An valued element occupies a position (by definition), which corresponds to its existance in the notion of response measures. Therefore, it enables both (L1) existance classification an (L2) value classification. How to select the measures, and what interpretation to put into it, is up the user on a case-to-case basis. Notice, that it may not make any sense to detect existence of a «function» track, since all positions are occupied by its definition.

3.2 Genomic machine learning data representation

The «translation» technique for turning genomic data content into a format which is understandable for machine learning algorithms is called *matrix generation*. It is a translation techique because it tries to retain the information of the existing format within the new (matrix) format. The translation generates a dataset (matrix) $\mathbb{S} = \{s_1, s_2, \ldots, s_m\}$ from an explanation track E, and possibly a response track R, where each sample has features from E and possibly response values (classes) from R. By this, a sample set $\mathbb{S}$ of m sample vectors have length m ($m = \mid \mathbb{S} \mid$). The samples of $\mathbb{S}$ is sorted based on their genomic coordinate positions, being the start position by default. A single sample $s_i \in \mathbb{S}$ $(1 \leq i \leq m)$ have a length of the n measures assigned to E and R. The application translation process uses the strategy described in section 2.1.2 on page 24, storing the samples in a matrix of shape ${}_m\mathbb{S}_n$.

3.2.1 Example translation

To illustrate a translation process, two data examples from 3.1.1 on page 32 are used. Example 3 is selected as the explanation track E, while Example 1 is selected as the response track R. Both tracks have start position $s = 0$, and end position $e = 20$, resulting in a total of $n = 20$ positions in both tracks. The undefined value is selected to be π, because its value $(3,14)$ is distinct and may easily be detected by the human eye in the sample set, as well as it is positive and is both similar to the range of other values at the same time as it is noticeably higher. The regularization parameter (for counter-acting possible overfitting) is set to the small positive number $\frac{3}{10}$ to use some (but not too much) regularization.

Since E is unvalued, only L1 response measures for binary classification are available. The `MLMeasurePointExists` is then chosen, in addition to 10 explanation features:

F_1 MLFeaturePositionRelative

F_2 MLFeaturePositionRelativeInverted

F_3 MLFeaturePositionRelativeCenter

F_4 MLFeaturePositionRelativeSides

F_5 MLFeaturePointDistanceLastRelative

F_6 MLFeaturePointDistanceFutureRelative

F_7 MLFeaturePointDistanceInnerRelative

F_8 MLFeaturePointDistanceOuterRelative

F_9 MLFeatureSegmentStartPosition

F_{10} MLFeatureSegmentEndPosition

The learning problem is a supervised learning problem since R is given as "answers" to E. The abstract challenge is to detect a relationship between the points of R and the segments of E. By looking at figure 3.9, it seem that points tend to occupy positions inside or close to the segments. This would then be the (abstract) targeted relationship. An interpretation is purposely not given to any of the tracks. Thus, analyzing the challenge could thus possibly benefit other abstractly alike challenges. The data is read from a MLTrack representation which is only an annotation track, which in turn is an abstraction of a genomic data sequencing process.

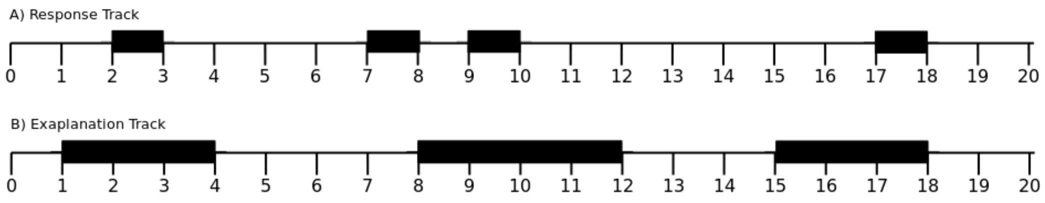


Figure 3.9: Illustration of a response track (1a) and an explanation track (1b), aligned on top of each other.

The first point of R (at position $k = 2$) lie in a position which is the center of the corresponding segment in E. The preceeding point ($k = 7$) lies next to the starting position of a segment in E, while the third point ($k = 9$) lie inside a segment in E. Finally, the last point ($k = 17$) of R also lie inside (at the end position) of a segment in E.

To give some insight into how the translation process works, a «snapshot» of position $k = 9$ is displayed in figure 3.10. The dashed arrows is directed at position k, which is colored gray in both tracks (E and R).

The state reader (MLTrackState) attached to E contains the list of available properties and is displayed in table 3.1. Concretely, the position $k = 9$ is clearly offered. Furthermore, since it is inside a segment, it has marked the position as occupied. Since the track format is unvalued, it holds the assigned value *null*. The distance markers are, in figure 3.10, marked with circles for start- and end-positions, a triangle for the (last) subsequent point (end-point of the segment) and a diamond for the (future) preceding point (start-point of the segment).

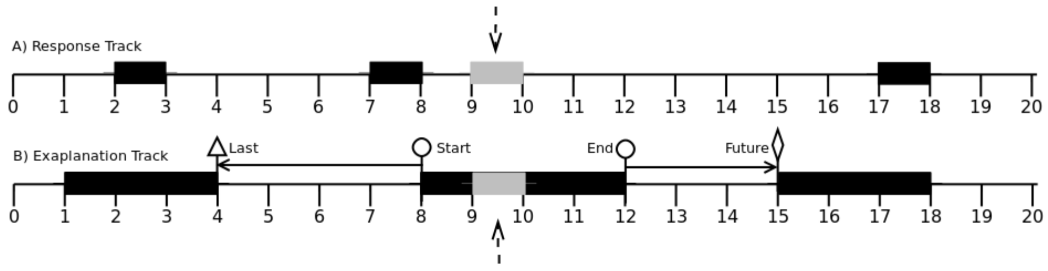


Figure 3.10: A «snapshot» of the application translation process at position $k = 9$.

Variable	**Value**
Position	9
Position condition (Open or Occupied)	Occupied
Position condition (Point or Segment)	Segment
Position condition (Valued or Unvalued)	Unvalued
Assigned value	*null*
Length	4
Distance to preceding point or segment	3
Distance to the subsequent point or segment	4

Table 3.1: The full list of the available properties, offered by the explanation track state at $k = 9$.

Variable	**Value**
Position	9
Position condition (Open or Occupied)	Occupied
Position condition (Point or Segment)	Point
Position condition (Valued or Unvalued)	Unvalued
Assigned value	*null*
Length	1
Distance to preceding point or segment	7
Distance to the subsequent point or segment	1

Table 3.2: The full list of the available properties, offered by the response track state at $k = 9$.

The properties of the response track state of R is shown in 3.2. It is a point, and is therefore occupying the position at $k = 9$.

The resulting matrix of the translation process is shown in table 3.3 on the next page and visualized in figure 3.11.

3.2.2 Post processing translation data

When translating genomic data, the dataset size may quickly reach millions. By using genome-wide annotation tracks, the data samples are intrinsic, since it is assumed that all available «content» is present. The ENCODE projects estimation of that 80% of the human genome is regarded as noncoding[11] (junk) DNA, is alone an indication that concept imbalance is lurking in the background.

1.00	0.00	1.00	1.00	0.00	3.14	0.05	3.14	0.05	0.00	0.00	0.00
1.00	0.05	0.95	0.90	0.10	3.14	0.20	0.00	0.20	1.00	0.00	0.00
1.00	0.11	0.89	0.80	0.20	3.14	0.20	0.05	0.20	0.00	0.00	1.00
1.00	0.16	0.84	0.70	0.30	3.14	0.20	0.00	0.20	0.00	1.00	0.00
1.00	0.21	0.79	0.60	0.40	0.00	0.15	3.14	0.00	0.00	0.00	0.00
1.00	0.26	0.74	0.50	0.50	0.05	0.10	3.14	0.05	0.00	0.00	0.00
1.00	0.32	0.68	0.40	0.60	0.10	0.05	3.14	0.05	0.00	0.00	0.00
1.00	0.37	0.63	0.30	0.70	0.15	0.00	3.14	0.00	0.00	0.00	1.00
1.00	0.42	0.58	0.20	0.80	0.20	0.15	0.00	0.15	1.00	0.00	0.00
1.00	0.47	0.53	0.10	0.90	0.20	0.15	0.05	0.15	0.00	0.00	1.00
1.00	0.53	0.47	0.00	1.00	0.20	0.15	0.05	0.15	0.00	0.00	0.00
1.00	0.58	0.42	0.10	0.90	0.20	0.15	0.00	0.15	0.00	1.00	0.00
1.00	0.63	0.37	0.20	0.80	0.00	0.10	3.14	0.00	0.00	0.00	0.00
1.00	0.68	0.32	0.30	0.70	0.05	0.05	3.14	0.05	0.00	0.00	0.00
1.00	0.74	0.26	0.40	0.60	0.10	0.00	3.14	0.00	0.00	0.00	0.00
1.00	0.79	0.21	0.50	0.50	0.15	3.14	0.00	0.15	1.00	0.00	0.00
1.00	0.84	0.16	0.60	0.40	0.15	3.14	0.05	0.15	0.00	0.00	0.00
1.00	0.89	0.11	0.70	0.30	0.15	3.14	0.00	0.15	0.00	1.00	1.00
1.00	0.95	0.05	0.80	0.20	0.00	3.14	3.14	0.00	0.00	0.00	0.00
1.00	1.00	0.00	0.90	0.10	0.05	3.14	3.14	0.05	0.00	0.00	0.00

Table 3.3: Overview of the translated dataset. The dataset is slightly imbalanced, with 4 samples of class=1 and 15 samples of class=0, giving an imbalance ratio of $4:15$ $(1:3,75)$.

In general, there may be many reasons for ending up with class imbalance of one or more majority and minority classes. A few strategies for counteracting this effect has been implemented. The idea behind the strategies are to group samples which are «equal enough» with others, so that they may be represented as a single combined sample.

There are special cases, where it might be better to select a narrower track range to learn from than using a post processing step for removing samples later on. Such cases could arise if a certain track range was of more interest than another, e.g. if there was a final number of positive instances of a given range which would make the data set imbalanced due to an absolute imbalance ratio. Of course, it is important to be aware of that by reducing the track range might exclude useful samples which may exist outside the given range.

Intervals

By the fact that samples are ordered the same way as they were translated, it is possible to group samples by extracting one combined sample for each interval of size k throughout the dataset $\mathbb{S}$ (where all samples has the same concept). The combined sample could be extracted in a multiple of ways, e.g. the first, the last, the average, a random sample of the range. The application implementation extracts an averaged sample.

The strategy requires the user to select a k value in order to know how many samples which it should be extracted. The k value is the amount of intervals, meaning the interval size is computed by $\frac{n}{k}$, where $n = |\mathbb{S}|$.

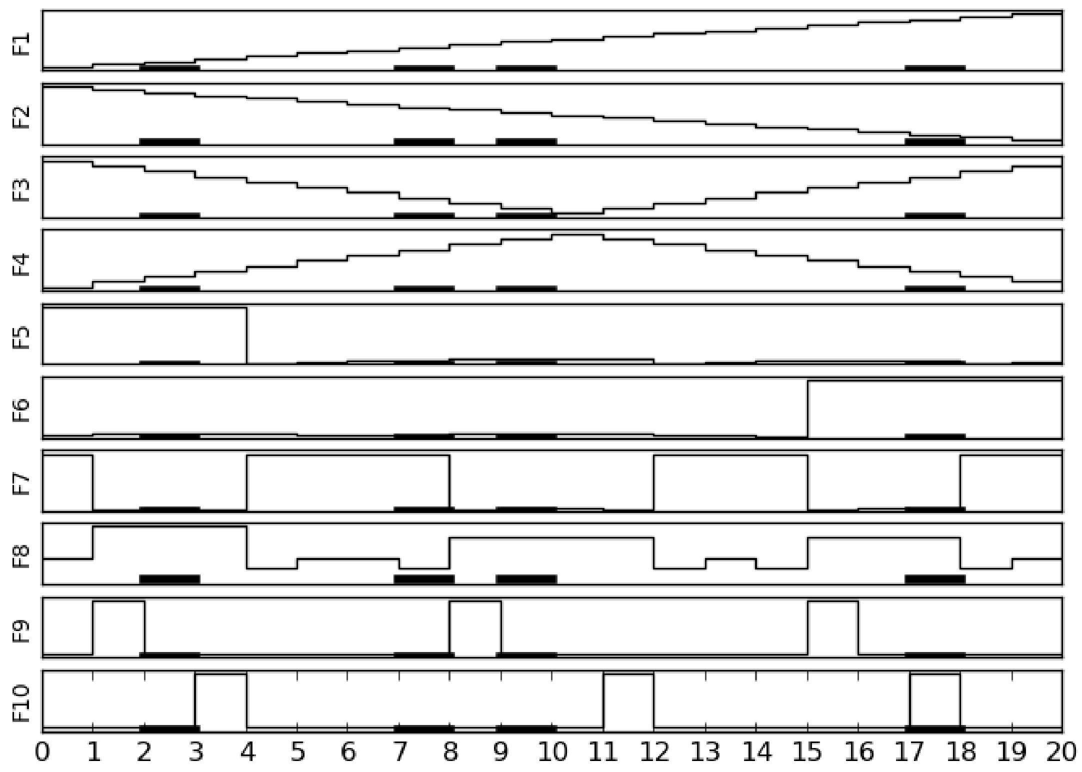


Figure 3.11: An visualization of the feature values of table 3.3. The illustration aims to highlight feature values in respect to the positions of the positive samples (class=1). The solid black (lower) line, at each feature, represents the presence of positive samples at position 2, 7, 9 and 17. The feature values are drawn as a (stepped) line for each position. The value scales are normalized to show the relative differences of the values.

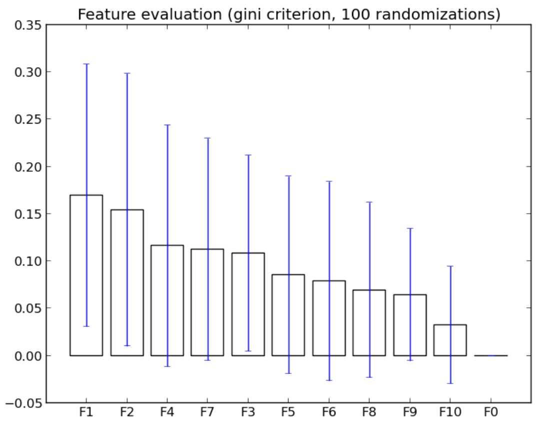
K-means

The unsupervised machine learning algorithm k-nearest neighbors uses *k* «centroids» in the same $\mathbb{R}^n$ dimensional space as for the sample data set which it operates in. By this, it is possible to group all samples to any of the *k* random placed centroids, and use the centroids position as the combined sample to represent the others in the matrix. Every sample is connected to the centroid which is closest, measured by the smallest euclidean distance.

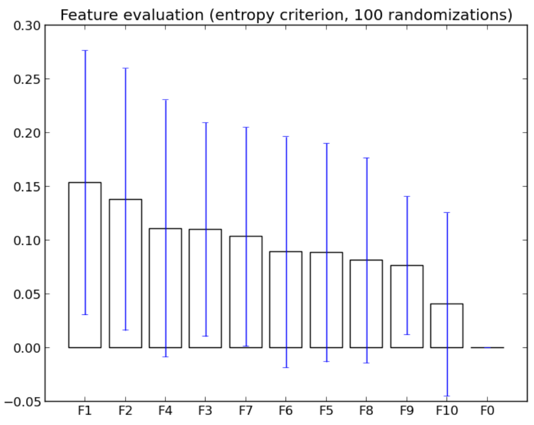
The strategy requires the user to select a *k* value in order to know how many samples which it should be returned. The *k* value is the amount of centroids. While the initialization of the centroid are randomized, there are performed a multiple of randomizations, were the best on who has smallest total distance of all centroids are chosen.

Roundoff

The strategy makes use of the fact that many of the samples are relatively similar due to the focus on the elements distance to each other. If an annotation track is relatively large (millions of positions), then two elements which are far



(a) The (gini) impurity.



(b) The information gain (entropy).

Figure 3.12: The feature similarity measurements by (a) gini and (b) entropy of the feature data in table 3.3.

from another element may be «quite equal» to each other. For instance, two points which has a gap of 1000 and 1001 to another element are pretty similar, at least in that particular distance feature. If the distance feature also makes use of the relativity, meaning it divides the value by the track length, the two points could share a number of common decimals. The common decimals, is what this grouping technique makes use of. It compares every sample with the rest of the samples and only keeps the ones which equals on the first d decimals. The strategy is implemented within the application to support the variants of both comparing 0, 1, 2, 3 and 4 «roundoff» decimals.

3.2.3 Feature similarity

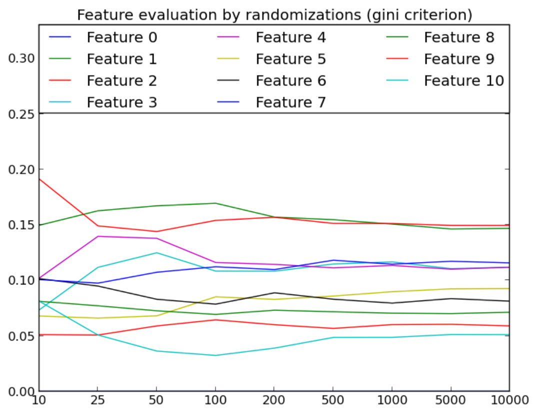
Evaluating *goodness* of features may be done in many ways[18]. Evaluating feature *similarity* inside the application are done by using two built in feature evaluation tools of SciKit, the impurity measure (gini) and the information gain (entropy)[27]. The standard deviation is also measured and shown as the blue line in the figure.

The gini coefficient[13] measures *impurity* or in-equality within a data set, for each feature individually. Concetely, when measuring impurity of a feature F, only the feature values of F are selected as the dataset for measuring the impurity of F. The measurement range is $[0,1]$, where 1 denotes the maximum and 0 means the minimum in-equality when all feature values are positive[28]. A small impurity value implies that all feature samples share similar values. A higher number implies that there are a (relatively) larger variety of values within the samples and possibly an in-class imbalance.

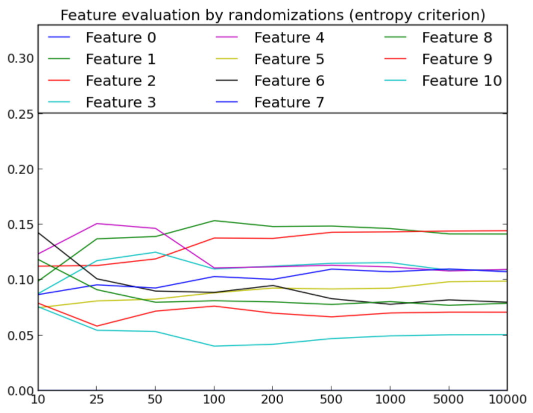
The Shannon entropy[36] measures *information gain* or the unpredictability of a random variable. It detects the unique amount of available information in a dataset. In the context of machine learning, it computes the average

[27]The feature evaluation is based on SciKit: `http://scikit-learn.org/0.12/auto_examples/ensemble/plot_forest_importances.html`

[28]If some of the values are negative, the range may theorethically excceed 1.



(a) The (gini) impurity.



(b) The information gain (entropy).

Figure 3.13: The feature similarity measurements by (a) gini and (b) entropy of the feature data in table 3.3 based on increasingly many randomizations.

information of each sample for the features, individually. A high value implies much information gain, while a smaller value implies less gain. A gain of 0 implies no information gain.

Detection feature goodness

The similarity measures shown in figure 3.12 is conducted based on 100 sample randomizations. However, more precise results may be achieved by doing increasingly more evaluations. To select a standard randomization number r for use with the various machine learning algorithms, a computation of increasingly may evaluations is shown in figure 3.13. The graph shows that there are some changes in the feature performances. However, it may appear that the features generally does not change much. Feature 9 improves a lot in impurity measure, while feature 3 drops in the entropy measure.

At around 200 evaluations, the plot more or less «flattens» out. The value r is therefore set as 200 within hte application when computing feedback for feature similarity. Thus, it aimsz to give a reasonable measure of feature goodness without beeing too computational expensive.

3.2.4 Track combinations

The combination of multiple tracks into a single track is proposed as a solution to situations that require learning from or prediction of multiple tracks. This can be achieved in a multitude of ways as long as it does not break any of the requirements of GTrack, MLTrack or any of their components.

A combination process of three ($n = 3$) single (UP) tracks (A, B and C) into a combined (VP) track D is illustrated in figure 3.14. The single tracks are numbered from 0 and upwards, resulting in $A = 0$, $B = 1$ and $C = 2$. The points in D is then computed using the existence property of the points in the single tracks. If none of the single tracks contains a point, then no point is created in D. Otherwise, a point is created in D as the binary sum of the existing points in the single tracks, by assigning the sum of the power of the track number for each track that contains a point at a position i. For instance, at position 0 ($i = 0$), C is the only track that has an existing point. C is numbered as track 2, which then assignes a point to D of value 4 (2^2). More formally, any position i in D (D_i) may be occupied by a new point of the assigned value v_i using

$$v_i = \sum_{j=0}^{n-1} \begin{cases} 2^j, & \text{if position } i \text{ in track } j \text{ is occupied} \\ 0, & \text{otherwise} \end{cases}$$

$$D_i = \begin{cases} \text{point of value } v_i, & v_i > 0 \\ \text{open position}, & \text{otherwise} \end{cases}$$

where n is the length of the set $\{A, B, C\}$ of single tracks.

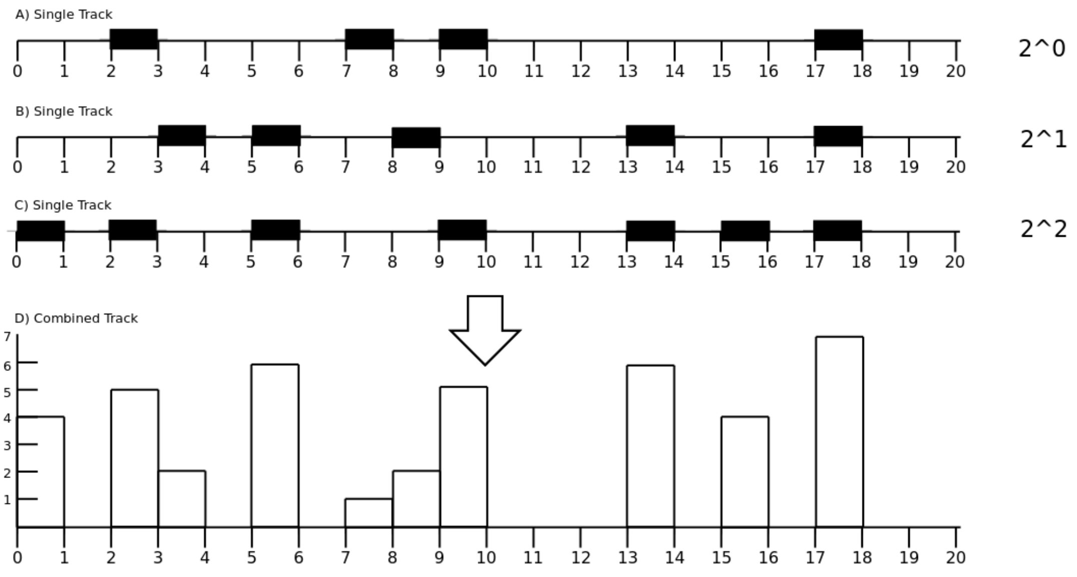


Figure 3.14: Combining three MLTracks into one.

3.3 Adapting machine learning algorithms

There are adopted 7 machine learning algorithms within the application, namely «Decision Tree», «Logistic regression», «Artificial Neural Network», «Support Vector Machine», «Anomaly detection», «K-nearest Neighbors» and «Multiple Linear Regression». All the algorithms extends the `MLAlgorithm`[29] interface[30], which is used by the application to execute the different algorithms in a generic manner. In addition, it offers visualizations (e.g. a learning curve) and various functionality performance measures (e.g. accuracy, precision and recall).

The application makes substantial use of the algorithm implementations and various other functionality from SciKit. Two of the algorithms («Artificial neural network» and «Anomaly Detection») are implemented using libraries from

[29]See appendix A.1.6 for implementation details.

[30]Concretely, the interface is implemented as an abstract class.

`scipy` and `numpy`, since the algorithms were not provided in the current version of SciKit. In general, the algorithms are more complex and holds a variety of implementation details which is not covered in this presentation. Some algorithms may both have a supervised and an unsupervised alternative, even though only one variant is used by the application.

The application has adopted the selected machine learning algorithms by focusing on learning genomic (and abstract) relationships. Consequently, only supervised machine learning algorithms or variants thereof is adopted. To learn a relationship, one genomic data track may thus be provided as «answers» to another genomic data track.

Each learning algorithm is presented in the context of reading translated genomic data based on the dataset (training set) of table 3.3. The presentation focuses on how genomic data is read and interpreted by the algorithm on a use-case level and not on any deep mathematical level. Thus, implementation details and mathematical proofs are not presented, even though some equations are provided.

Concretely, each of the learning algorithms is presented using the dataset (training set) of table 3.3, generated by using the applications translation strategy. To describe the prediction step, an *unseen* track is used as an example. The track is *unseen* because it is not used in the learning step, and its response track is unknown. The unseen track is of same type (US) as the explanatory track used to generate the training set. A visualization of the unseen track is shown in figure 3.15. The resulting dataset (prediction set) of the translated unseen track (using the same features as the ones used in the learning step) is shown in table 3.4.

The undefined value parameter is selected to be π. The data partitioning pipeline step is chosen to be skipped in the illustration, to maintain a simple example. The learning set is thus used for both learning and cross-validation whenever applicable. The focus is to explain the learning and prediction step of each algorithm in the context of working with the (translated) genomic data.

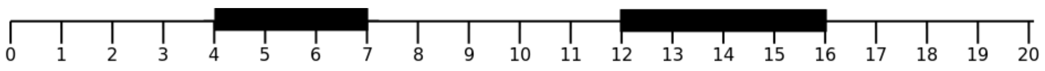


Figure 3.15: Visualization of the prediction track.

Prediction track as GTrack

```
##gtrack version: 1.0
##track type: segment
###seqid          start    end
####genome=hg18;start=0;end=20
chr1     4          7
chr1     12         16
```

Prediction track as MLTrack

```
####genome=hg18;start=0;end=20
chr1     4          7
chr1     12         16
```

1.00	0.00	1.00	1.00	0.00	3.14	0.20	3.14	0.20	0.00	0.00
1.00	0.05	0.95	0.90	0.10	3.14	0.15	3.14	0.15	0.00	0.00
1.00	0.11	0.89	0.80	0.20	3.14	0.10	3.14	0.10	0.00	0.00
1.00	0.16	0.84	0.70	0.30	3.14	0.05	3.14	0.05	0.00	0.00
1.00	0.21	0.79	0.60	0.40	3.14	0.25	0.00	0.25	1.00	0.00
1.00	0.26	0.74	0.50	0.50	3.14	0.25	0.05	0.25	0.00	0.00
1.00	0.32	0.68	0.40	0.60	3.14	0.25	0.00	0.25	0.00	1.00
1.00	0.37	0.63	0.30	0.70	0.00	0.20	3.14	0.00	0.00	0.00
1.00	0.42	0.58	0.20	0.80	0.05	0.15	3.14	0.05	0.00	0.00
1.00	0.47	0.53	0.10	0.90	0.10	0.10	3.14	0.10	0.00	0.00
1.00	0.53	0.47	0.00	1.00	0.15	0.05	3.14	0.05	0.00	0.00
1.00	0.58	0.42	0.10	0.90	0.20	0.00	3.14	0.00	0.00	0.00
1.00	0.63	0.37	0.20	0.80	0.25	3.14	0.00	0.25	1.00	0.00
1.00	0.68	0.32	0.30	0.70	0.25	3.14	0.05	0.25	0.00	0.00
1.00	0.74	0.26	0.40	0.60	0.25	3.14	0.05	0.25	0.00	0.00
1.00	0.79	0.21	0.50	0.50	0.25	3.14	0.00	0.25	0.00	1.00
1.00	0.84	0.16	0.60	0.40	0.00	3.14	3.14	0.00	0.00	0.00
1.00	0.89	0.11	0.70	0.30	0.05	3.14	3.14	0.05	0.00	0.00
1.00	0.95	0.05	0.80	0.20	0.10	3.14	3.14	0.10	0.00	0.00
1.00	1.00	0.00	0.90	0.10	0.15	3.14	3.14	0.15	0.00	0.00

Table 3.4: The feature values of the translated prediction track, using the features of section 3.2.1 on page 45.

3.3.1 Decision Tree

The «Decision Tree» algorithm[42] (`MLDecisionTree`[31]) uses a tree structure for representing features as inner nodes and classes as leaf nodes.

In the learning step, a decision tree T i built up by feature nodes, ordered by their level of entropy (information gain). The best feature node (with the highest entropy of all the features) is selected as the *root* node. Then, the best of the remaining feature nodes are added as the current (root) nodes first child. Each feature node has one or more child nodes (feature nodes or class nodes). During learning, the inner nodes learns to select which of their assigned children to return, based on one or more (inner) conditions. The class nodes are *leaf nodes* (ends of the trees branches) and have (only) children which corresponds to one or more aswer classes. A tree is built based on some given parameters, such as how many feature nodes to use, how many times a feature node may be used, and so on. There are many available options. Thus, a feature node may be used once, a multiple of times or not at all. It solely depends on the algorithm implementation and its (optional) parameters. The SciKit implementation of the decision tree is configured (parameterized) to use entropy[32] for building the tree and it has no upper limit of feature reuse.

In the prediction step, the inner feature nodes decide what child (branch) to follow or return, based on individually learned conditions. Any condition is

[31]See appendix A.2.1 for implementation details.

[32]The entropy is selected to be the default configuration used by the algorihtm in the implemented application.

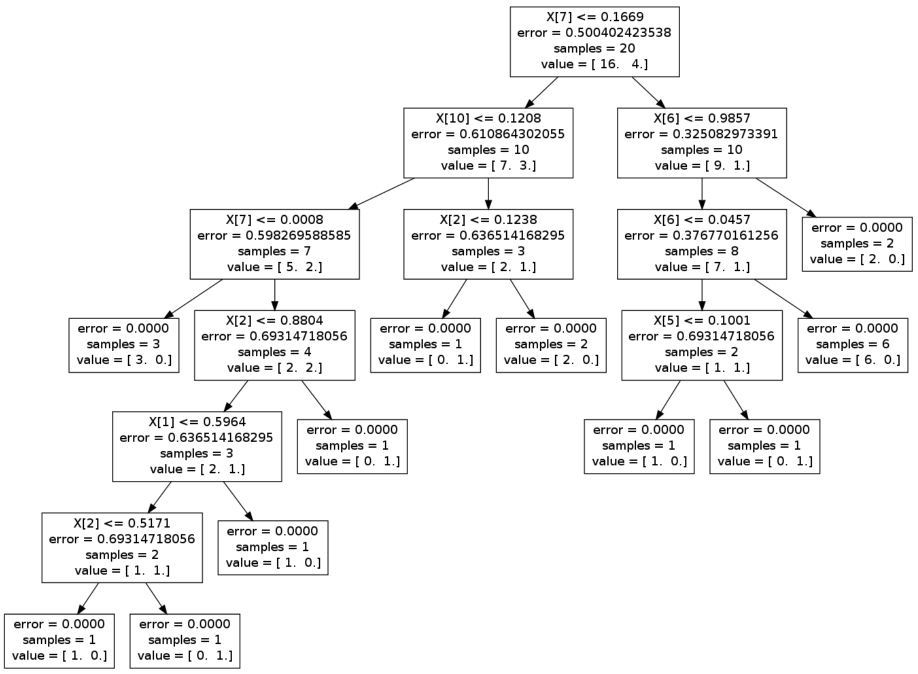


Figure 3.16: A decision tree built from the training set with 100% accuracy.

evaluated based on a given feature value from the prediction sample. Based on the condition, an inner feature- or class-node is returned. If a class node is returned, then the sample is predicted to be the class represented by the class node. Otherwise, if an inner feature node is returned, more classification consideration is needed. Thus, the process recursively evaluates the returned child node in the same way as the current node. Figure 3.16 shows a learned decision tree (with an accuracy of 100%) for training set. The root node is the single node on top of the figure, representing the $7th$ feature (MLFeaturePointDistanceInnerRelative). It has two children which is both (other) feature nodes. It evaluates its (learned) condition based on a given feature value v from an unseen sample. If $v \leq 0.1669$, the left branch to the $10th$ feature node is selected. Otherwise, the right branch to the $6th$ feature node is selected.

By following the right path from the root node, only one of the leaf nodes along the continuing path has a positive class (class=1), while the remaining three positive sample classes are left out. It is a «dramatic» decision to exclude $\frac{3}{4}$ of the positive classes in the very first decision, but this is precisely the aim when using entropy. It tries to pinpoint information gain as fast as possible. Thus, the reduction of positive classes confirms that the selected root feature is a good selection, since the model have perfect accuracy.

The complexity of T is rather large compared to the size of the used training set. The number of nodes in the tree is about the same as the number of samples. It raises the question of generalization, since having a very specific model might not suite other (possibly) different situations as good, if overfitted.

To illustrate the use of improving the generalization, another more simplistic decision tree T' is learned, as shown in figure 3.17. The simplicity is due to a given «max-depth» parameter of 2. It parameter selection results in a much

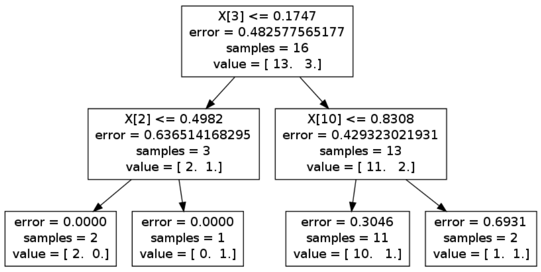
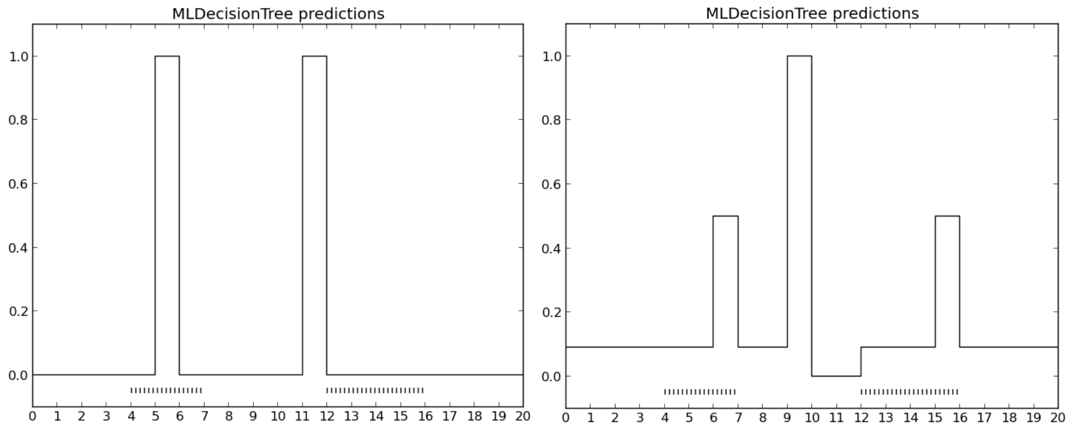


Figure 3.17: Decision Tree applied to the sample data set with restrictions a max depth of 2, achieving an accuracy of 85%.

simpler model with only 7 nodes (a reduction by 66,7% from 21 nodes). Even though the accuracy drops down to 85,0% for the same training set (a reduction of 15,0%), it is still a reasonably good performance. Furthermore, the prediction estimates of the unseen track as shown in figure 3.18b, is less accurate than ones in figure 3.18a.



(a) Decision tree probabilities of T. **(b)** Decision tree probabilities of T'.

Figure 3.18: Decision tree probabilities for predicting positive classes (class=1) on an unseen track of the two hypothesises T and T'. The dots on the lower part of the figure is the presence of segments in the prediction track.

3.3.2 Multiple linear regression

Multiple linear regression [42] (`MLMultipleLinearRegression`[33]) is a regression algorithm. It tries to fit a line to a dataset $\mathbb{S}$ by assigning a weight to each features of the dataset. The line is a continous function, which outputs a computed value v for any given input sample s ($s \in \mathbb{R}^k, s \in \mathbb{S}$).

$$h(x, \beta) = \beta_0 + x_1 \times \beta_1 + x_2 \times \beta_2 + \ldots + x_{k-1} \times \beta_{k-1} + x_k \times \beta_k$$

In the learning step, the samples from the training set is represented as points in a $\mathbb{R}^k$ dimensional space, based on the k features of the dataset. Then, each parameter (β) for each of the features is adjusted to minimize the distance

[33]See appendix A.2.1 for implementation details.

from any computed value to the samples. Usually the minimization function is the least squares equation. It computes the squared sum of distances (cost) between the "correct" (training) class values and the computed values for all the samples, as in:

$$cost(\mathbb{S}\,|\,\beta) = \sum_{i=1}^{i=|\mathbb{S}|} \Big(s_{i,k} - h([s_{i,0}, s_{i,1}, \ldots, s_{i,k-2}, s_{i,k-1}], \beta)\Big)^2$$

If the model h has trained the parameters (β) perfectly, the cost would equal 0. The β_0 parameter (the bias) is the intercept which is usually adjusted to a value which would minimize the cost.

In the prediction step, a unseen data sample may be passed to h using the learned parameters (β), to compute its value. If the model is properly trained, the unknown value would be at the line.

The algorithm is not used to estimate the unseen track, since it is a regression model and not a classification model.

3.3.3 Logistic regression

Logistic regression [22, 42] (`MLLogisticRegression`[34]) is a (binary) supervised classification algorithm. It classifies samples by a learned classification line called a *decision boundary*. It is closely related to linear regression, where such a decision boundary is a learned mathematical function for estimating (calculating) values. The logistic function adds an extra (sigmoid) evaluation on top of this, to be able use these values to make decisions. The sigmoid function, shown if equation 3.1, returns a numeric value within $[0,1]$ based on a given value v. Thus, a threshold ϵ may be learned and used to decide whether v is under or above it, so it could be classified as a positive or negative instance (class).

The decision boundary is separating at most two classes, and does by this work naturally as a binary classifier. It is also possible to use it for multiclass classification, using the «one-vs-all» method from section 1.1.2 on page 8.

$$S(v) = \frac{1}{1 + e^{-v}} \tag{3.1}$$

$$decide(v,\epsilon) = \begin{cases} True, & \text{if } S(v) \geq \epsilon \\ False, & \text{otherwise} \end{cases} \tag{3.2}$$

The learning step initializes a parameter vector (hypothesis h) of the length of features in addition to a bias feature. The parameter values of h are adjusted (trained) to separate class instances in a manner which minimizes the amount of mis-classifications. The cost function used for training is a maximum-likelihood estimation.

The prediction step classifies a sample s by using h to compute its value v which in turn is compared to ϵ (the threshold). The value is used as the estimated probability for the positive class.

The learned hypothesis h for the example dataset in table 3.3 achieves an accuracy of 80%. The prediction estimates, shown in figure 3.19, are slightly higher in the segment regions than in the areas outside them.

[34]See appendix A.2.1 for implementation details.

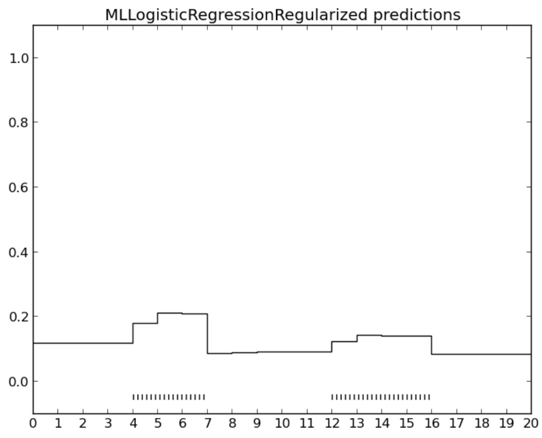


Figure 3.19: Logistic regression probabilities for predicting positive classes (class=1) of an unseen track. The dots on the lower part of the figure is the presence of segments in the prediction track.

3.3.4 Artificial Neural Network

Artificial neural network[42] (`MLArtificialNeuralNetwork`[35]) is a supervised algorithm, inspired by the way the human neural network operates. Concretely, it uses the notion of combining the many dendrites of an axon to multiple other axons while learning the signal «routes» amongst these (the «network»), individually. The simulation of such a networked relationship between axons and dendrites are implemented by representing the axons a matricies *layers* and their dendrite conncetions though matrix multiplications.

There are involved usually a minimum of three layers (matrices) when learning such network routes, namely the input layer, the output layer and minimum one hidden layer. The input layer size equals the number of features, while the output layer equals the amount of response classes. The hidden layers are usually chosen and tuned specifically (and individually) on a case-to-case basis. Furthermore, there may be a multiple of (trained) hidden layers, which together forms the learned hypothesis h.

The learning step uses a two step process called *feed-forward* and *back-propagation* to train the hidden layer(s). The feed-forward step calculates the current performance and cost of feeding the training samples through the input layer of h. The back-propagation starts with the answers of the output layer and moves backwards, towards the input matrix while calculating error costs for every uncertainty in the matrices of h. These two training steps runs back and forth until it performs either «well enough» or reach a given maximum number of iterations. In figure 3.20, the three layers are visualised as a heat-map, where a darker color means a higher value. The three model matrices are (in the example case) the input layer $\mathbb{I}$, the hidden layer $\mathbb{H}$ and the output layer $\mathbb{O}$. The model achieves an accuracy of 85%.

In the prediction step, a (input) dataset $\mathbb{I}$ is calculated based on the learned hidden layer $\mathbb{H}$ so that ${}_{20}\mathbb{I}_{10} \times_{10} \mathbb{H}_{29} \times_{29} \mathbb{O}_2 =_{20} \mathbb{R}_2$. The two columns of ${}_{20}\mathbb{R}_2$ contains the estimates for the classes. The i-th sample is classified at class=j when the j-th column has the highest estimation value.

[35]See appendix A.2.1 for implementation details.

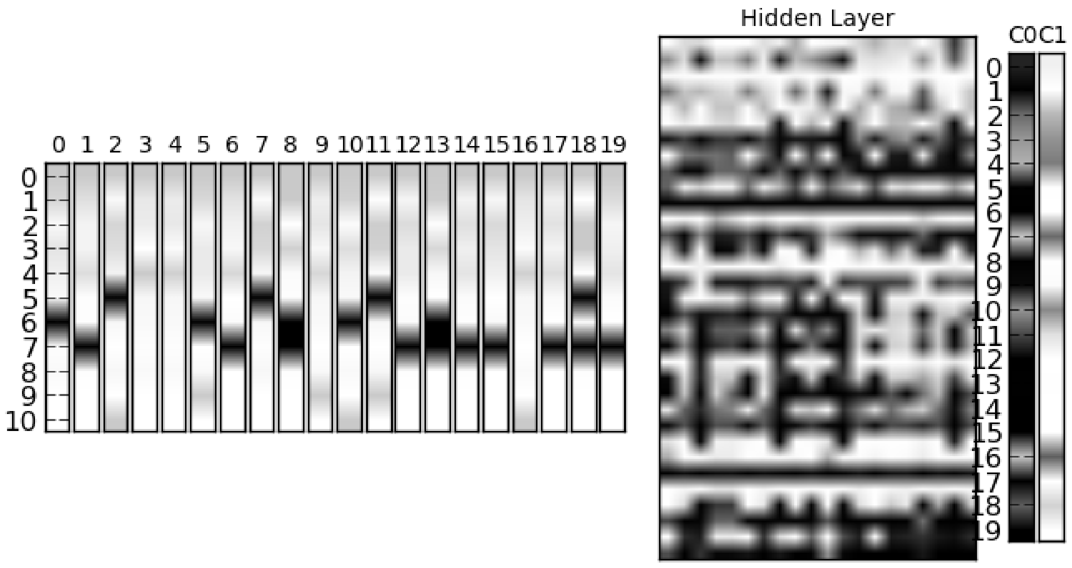


Figure 3.20: A visualisation of an artificial neural network, leared from the example training set. The black areas correspond to a high value while the white areas corresponds to a low value within the range $[0,1]$. An additional layer factor parameter is set to 2.9, and the regularization parameter is set to 0.

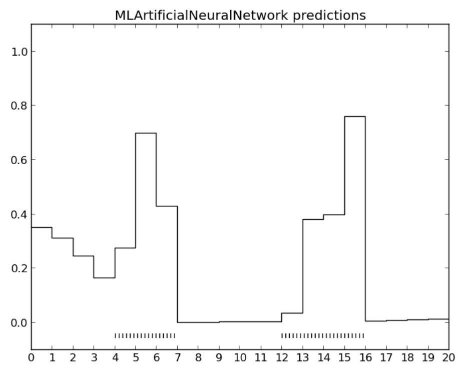


Figure 3.21: Artificial neural network probabilities for predicting positive classes (class=1) of an unseen track. The dots on the lower part of the figure is the presence of segments in the prediction track

3.3.5 K-nearest Neighbors

The k-Nearest neighbours algorithm[42] (`MLKNearestNeighbor`[36]) is an unsupervised learning algorithm. It groups samples into k groups in k-dimensional space $\mathbb{R}^k$, where all samples have an euclidean distance to each other, and may therefore be «close» or «distant» from each other through the k dimensions. The parameter k is usually chosen as the amount of classes to classify, and denotes the amount of segments to be grouped.

The learning algorithm randomly initializes «centroids» which take «ownership» of the samples closest to its position in $\mathbb{R}^k$. The k (different) centroids reposition themselves increasingly closer to its nearest (and largest) group for every iteration, until the iterations reaches its maximum or when all centroids have found a *sweet-spot*. A sweet-spot is where a centroid are not befitting from any additional repositioning. Since centroid positions are randomly initialized, the

[36]See appendix A.2.1 for implementation details.

algorithm introduces a notion of chance, where one solution may be better than the other solutions and possibly an optimal[37]solution. Therefore, the algorithm is executed a set number of times which hopefully is enough to select a proper solution with the possibly lowest total distance from all centroids to their owned samples of the assigned classes.

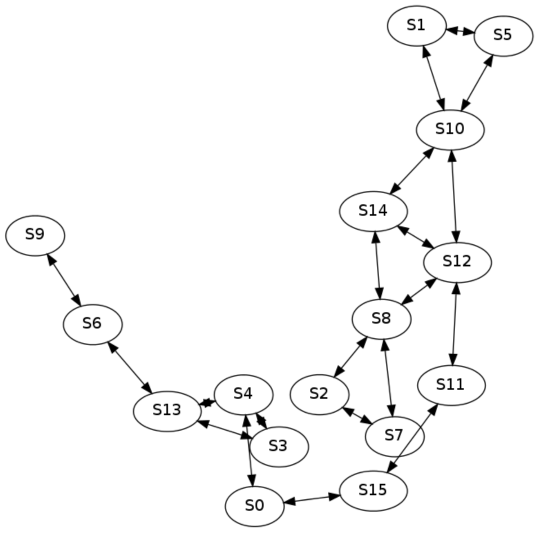


Figure 3.22: A visualization of the distances in-between the learning samples.

The prediction step maps an unseen sample s ($|s| = k$) into $\mathbb{R}^k$, and detects its closest centroid, which represents the class (answer). The most important (learned) distances are the in-between distances of negative samples (class=0) and is shown in figure 3.22. The distance relationship between the samples are shown in $\mathbb{R}^2$, since a true visualization in $\mathbb{R}^{10}$ is difficult[38]. The length of the arrow represents the distance length. In this, specific case, the learned hypothesis h achieves an accuracy of 85%.

The prediction estimates looks promising within the segments of the prediction track, as shown in figure 3.23. But, the gap between the two segments also yield high estimates.

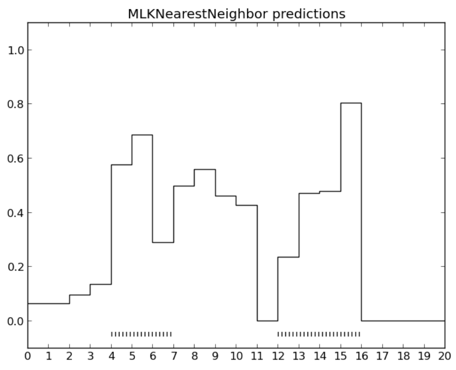


Figure 3.23: K-nearest neighbour probabilities for predicting positive classes (class=1) of an unseen track. The dots on the lower part of the figure is the presence of segments in the prediction track.

[37]An optimal solution is a solution where all pure segments are covered by the centroids in a way that perfectly match any future prediction. Such a solution is difficult, if possible, to find.

[38]The visualization of points is usually limited to $\mathbb{R}^3$). Techniques for displaying higher dimensions is difficult, and beyond the scope of presenting the algorithm.

3.3.6 Anomaly Detection

Anomaly detection (`MLAnomalyDetection`[39]) is a semi-superviced (binary) classification algorithm, which classifies samples as a *normal* sample or an *anomaly*. It is used to discover samples which are specially different (anomal) to the other (normal) samples. It is a semi-supervised algorithm, since it first builds an hypothesis h of all samples which is considered to be normal, and then sets a boundary for anomalies based on samples considered to be anomalies.

In the learning step, the algorithm «fits» a normal (gaussian) distribution (equation 3.3) for each feature f in the training set ($f \in \mathbb{F}$). Each feature computes its average value μ_f and its standard deviation σ_f, individually. The feature values are usually normalized (individually) to make each feature have similar range of values (influence) as the rest of the features. The algorithm has been created based on the idea that a normal samples should have feature values within the range r of $\mu_f \pm \sigma_f$. By this, an anomaly might easily be detected if the sample falls outside r in at least one of the features.

$$P_f(x_f|\sigma_f, \mu_f) = \frac{1}{\sigma_f \sqrt{2\pi}} e^{-(x_f - \mu_f)^2 / 2\sigma_f^2} \tag{3.3}$$

To predict whether a sample s is an anomaly or not, the product of all feature values is computed using the learned μ and σ for all features f ($f \in \mathbb{F}$). The estimated value of s is the product of all the individual feature probabilities, as represented in equation 3.4.

$$decide(s) = \begin{cases} Normal, & \text{if } \left(\prod_{f=0}^{|\mathbb{F}|} P_f(s_f) \right) \geq \epsilon \\ Anomaly, & \text{otherwise} \end{cases} \tag{3.4}$$

By looking at the example dataset (table 3.3 on page 48), there are 4 positive (class=1) and 16 negative samples (class=0). Thus, the positive classes are the minority and may be treated as the "anomalies". The learned normal distributions show the distribution of the samples in figure 3.24, with positive samples marked with crosses ($\times$) and negative samples marked with circles (o). An anomaly feature should conceptually gather as many negative examples at the center top of the graph, while leaving positive values on the lower sides. The result of applying each of the features in the left column, is shown in the right column, with the separation boundary marked with the pipe symbol ($|$). A perfect separation would put all negative samples on the right, and all the positive samples on the left of the boundary. The prediction results of h is shown in figure 3.25a, and achieves an accuracy of 20%.

By looking at the visualization in figure 3.24, some of the features looks to be performing better than others. To improve the accuracy of h, the features that performances poorly may be removed.

Therefore, a new simplified hypothesis h' is learned solely based on the features F_5 and F_7 from the original list of features in section 3.2.1 on page 45. The predictions of h' achieve an accuracy of 60% and is shown in figure 3.25b. The prediction results of h' on the unseen data is thus remarkably better than the ones of h.

[39]See appendix A.2.1 for implementation details.

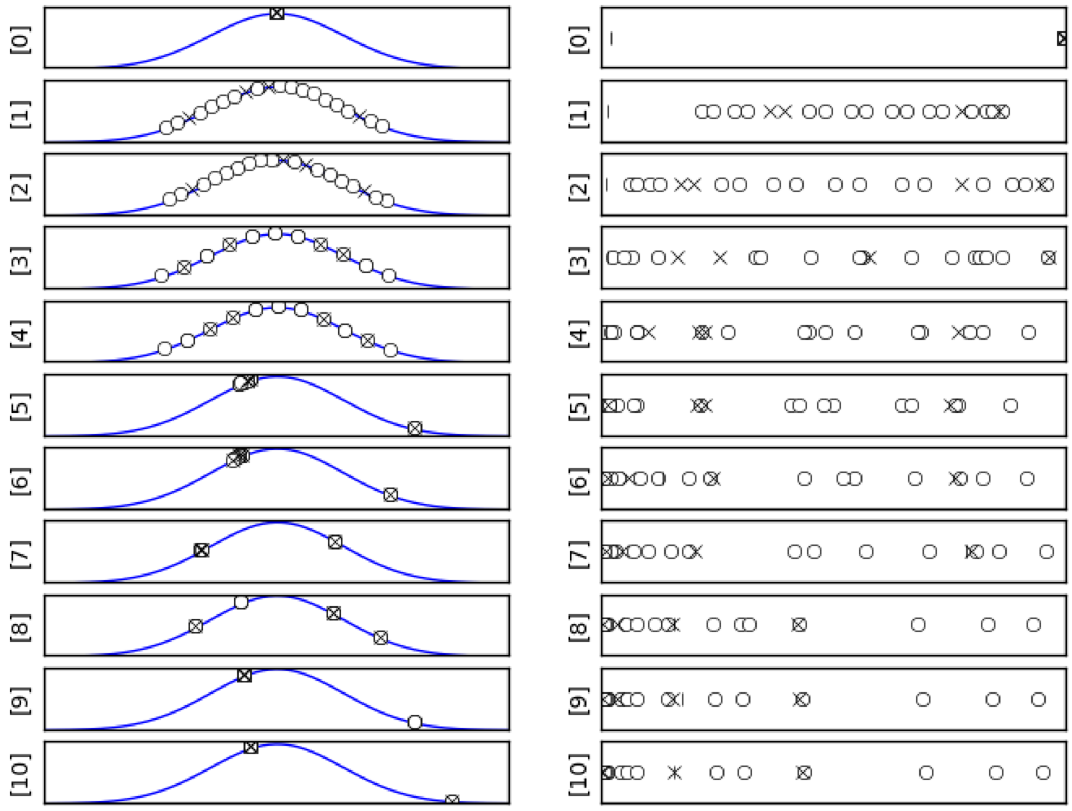
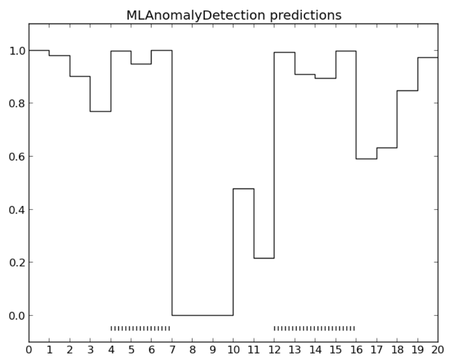
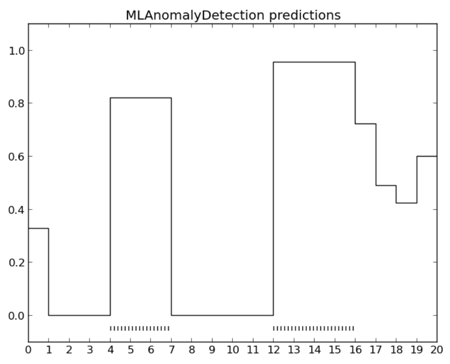


Figure 3.24: The left column is the normal distributions for each feature. The circles are the normal samples, while the crosses are the anomalies to be detected. The right column displays the result of applying the features from the left column. The vertical line (the «pipe» '|') displays the threshold. All samples to the left for this pipe is regarded as an anomaly.



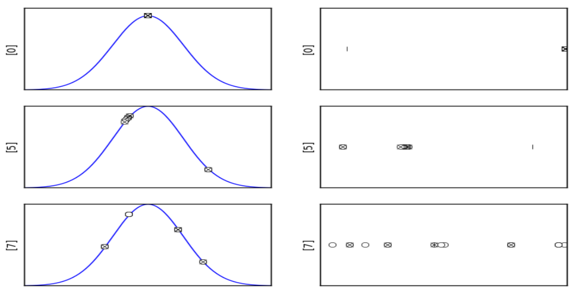
(a) The first model.



(b) The simplified model.

Figure 3.25: Anomaly detection probabilities for (a) the first model and (b) the simplified model, used for predicting positive classes (class=1) of an unseen track. The dots on the lower part of the figure is the presence of segments in the prediction track.

Figure 3.26: A simplified anomaly detection model, only using features F_5 and F_7.



3.3.7 Support Vector Machine

Support vector machine (`MLSupportVectorMachine`[40]) is a supervised learning algorithm, which separates n classes of points by the largest margins in $\mathbb{R}^n$, known as *support vectors*, using «kernels». There are many types of kernels (ways of separating classes). Any given kernel must be a «valid» kernel by satisfying Mercer's Theorem[29]. Figure 3.27 displays three common kernels, supported by SciKit. As shown in the figure, the points are differently separated based on the kernel selection.

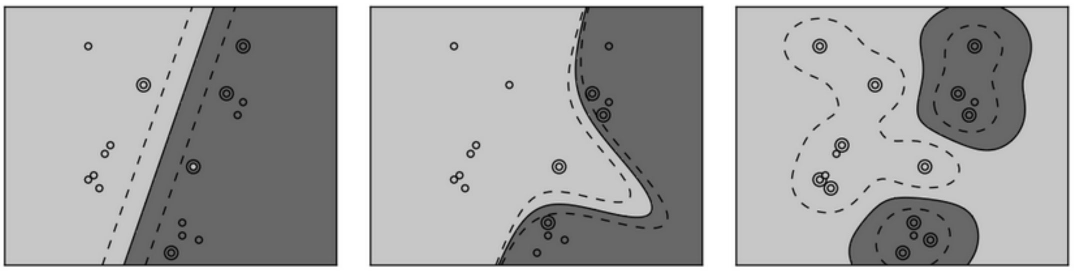


Figure 3.27: Illustration (from the left) of the linear-, polynomial- and rbf-kernel. Image is taken from SciKit.

The learning step maps each sample (vector) into a new «feature vector» based on the selected kernel. The new feature vectors are then used to find the largest margin separation boundaries by calculating the largest euclidean distances between the feature vectors. By this, it is the sum of all the samples kernel boundaries which makes the overall decision boundary, which therefore may become very complex.

The prediction step maps each sample in the same way as in the learning step, and returns the class based on its position in regards to the decision boundary. The simplest case of bounday, is in the case of binary classification. Then, a sample (represented as a feature vacor) may be inside or outside the boundary. In the case of multi-class or n-class classification, the one-vs-all method is used to learn n classifiers.

[40]See appendix A.2.1 for implementation details.

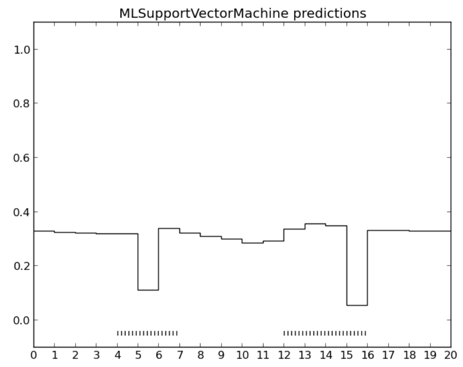


Figure 3.28: Support vector machine probabilities, for predicting positive classes (class=1) of an unseen track. The dots on the lower part of the figure is the presence of segments in the prediction track.

3.3.8 Comparing probabilities

Throughout the presentation of the algorithms, an estimation of the (unseen) prediction track has been provided. Figure 3.29 show a comparison of the probabilities as a combined track, which is the mean of all the tracks. Thus, using the results of a combination of two or more machine learning algorithms may be beneficial for either supporting or reducing the influence of each of the individual algorithm finds.

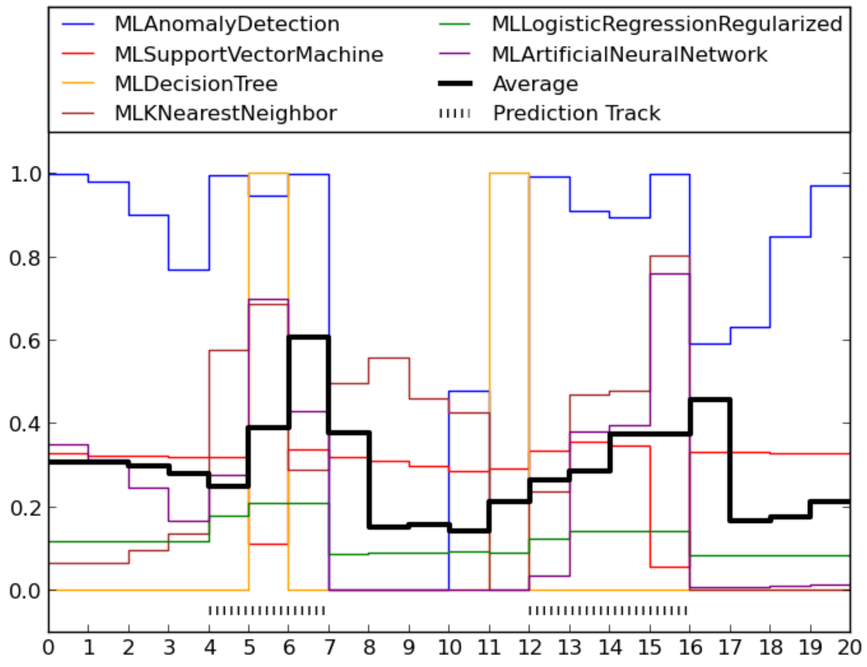


Figure 3.29: Probabilities for the algorithms of the (unseen) prediction track in figure 3.15 on page 53 in addition to a combined probability track of their mean values.

3.4 Application design

This section makes use of the components of section 3.1 and 3.1.7 for creating a flexible machine learning application for analyzing (generic) genomic challenges.

3.4.1 Design goals

The main goal of the application is to enable biological (abstractly equivavalent) problem formulations to be analyzed in a generic way. It aims to generalize currently known problem formulations in hope of establishing a applicable framework to future problem forumlations as well.

Another goal is to keeping «all options on the table» for as long as possible. Thus, the application user may consider and explicitly choose what actions, configurations and tradeoffs to make.

The application design guidelines are to:

- Support the representation of the genomic elements.
- Favor generalization over specialization.
- Offer a list of possible machine learning algorithms.
- Support a broad range of data input.
- Automatically perform a good parameter selection by default.

3.4.2 Framework

In order for all of the elements to work together, a framework has been created. It is illustrated in figure 3.30.

3.4.3 Tools

The application exists of four tools representing the «natural» isolated steps of an abstract machine learning pipeline.

Creating a tool for each of the steps aims to keep the application flexible by allowing tools to be reused. For instance, any hypothesis could thus be used to predict multiple genomic tracks or ranges.

Each tool makes use of both mandatory (required) or optional parameters. However, all optional parameters is defaulted to a reasonable value in regards to the guidelines.

The first step is to create measures, and is supported by the «Measurement Creator Tool».

The second step is to learn an hypothesis from the created measures and selected genomic tracks[41] using the «Learning Tool».

The third step, uses the learned hypothesis to make predictions and estimations for another genomic track with the «Prediction Tool».

Finally, the fourth step exports estimations, predictions and graphical visualizations (plots) using the «Exportation Tool».

[41]Genomic tracks are provided by the HyperBrowser framework.

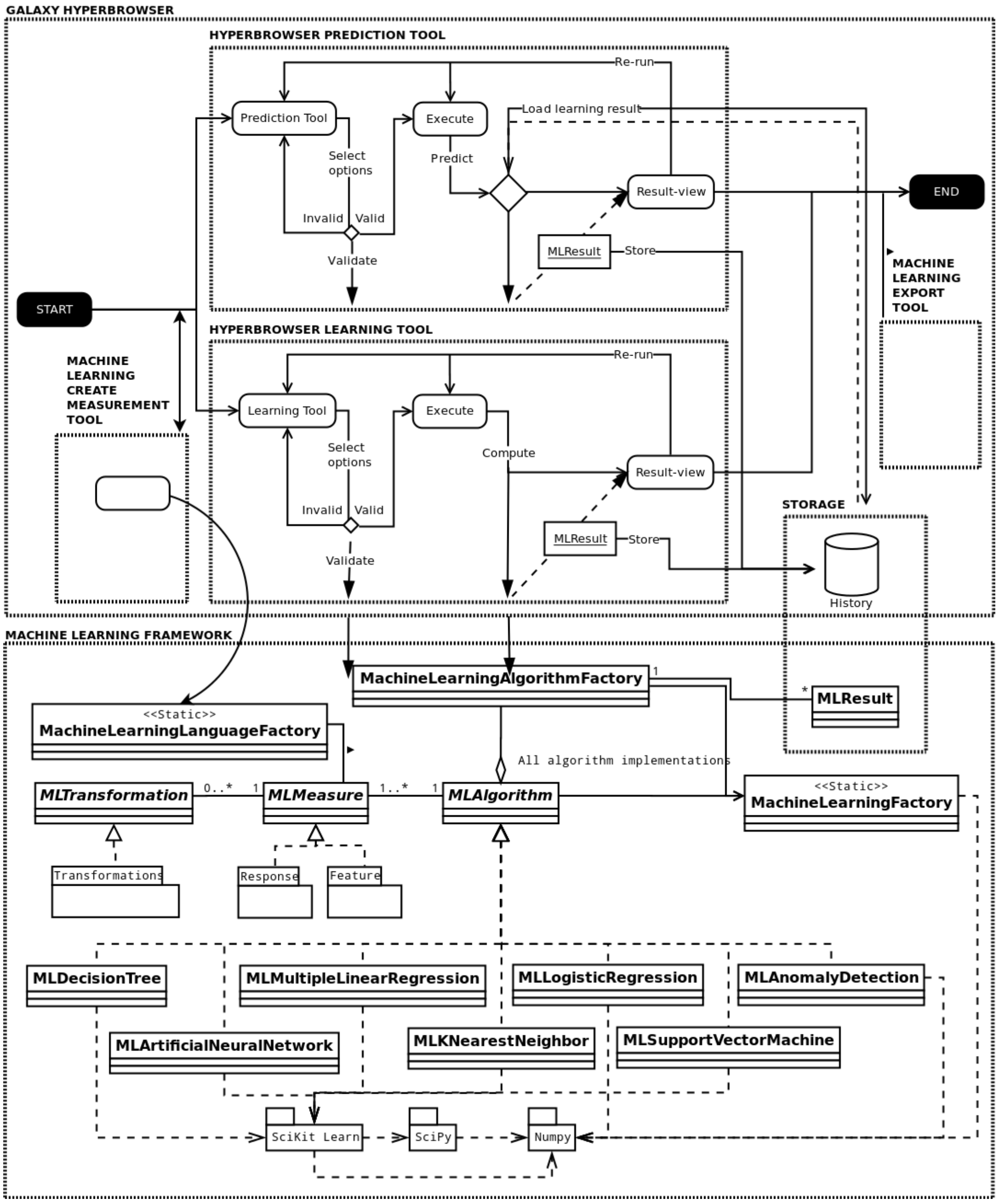


Figure 3.30: A combined UML class- and state-diagram of the machine learning framework and its relation to the HyperBrowser. The four tools are all using the same framework API. The learning and prediction tools are mainly wrappers around invoking the *compute* and *predict* methods.

Measurement Creator Tool

The tool produces a textfile of measures and transformations in MLL syntax to be used by the «Learning Tool». Thus, it makes use of the applications implementation of MLL by using the MLMachineLearningLanguageFactory for handling the transition of measures, transformations and strings. Storing the measures as textfiles enables sharing the measurements with others, as well as it hopefully ease the recievers effort of interpreting the purpose of the measures through the high-level MLL syntax.

However, it is possible to directly upload a MLL file within the HyperBrowser[42]. Any uploaded MLL file in valid syntax could be used directly by the «Learning Tool». Thus, the «Measurement Creator Tool» is by itself optional.

The tool offers to either produce a MLL file by either making parameter selections or edit an existing MLL file[43]. The tool parameters are:

Create or change existing measures
: Mandatory. Select between «creating new measures» or «edit existing measures».

Measurement usage
: Mandatory, iff[44] «creating new measures» is selected. Choose the type of measures that are to be created. The two available types are «explanation» or «reponse» measures.

Track format
: Mandatory, iff «creating new measures» is selected. Select between the 5 supported track formats, namely «segment», «valued segment», «points», «valued points» and «function».

Transformations
: Optional, iff «creating new measures» is selected. Add[45] one or more generic transformations to each generated feature. Defaults selection is «No transformations», where no transformations are added.

Undefined value
: Optional, iff «creating new measures» is selected. Select a value v to be used when a measurement is undefined (cannot determine v). Default selection is the systems maximum (positive) integer[46].

Existing measurements
: Mandatory, iff «edit existing measures» is selected. Select[47] an existing textfile of MLL syntax to edit.

[42]Uploading a datafile in the HyperBrowser is done at the left menu on the frontpage under «Galaxy Tools» -> «Get Data» -> «Upload File».

[43]Due to integration challenges in the framework, a textfile cannot both be created and edited at the same time. It has to be done i two steps. An additional «work mode» parameter is also needed to load the content of the selected MLL file at the correct time.

[44]«iff» is short for «if and only if».

[45]Adding transformations is done by an cutsom implemented selection «wizard», which enables to select transformations in an iterative selection process. The «wizard» was created because of integration challenges with the framework. Advanced use, as selecting a transformations multiple times, is not supported directly. The advanced use selections are possible through manually editing the MLL content.

[46]In python, the systems maximum integer is the `maxint` value in the `sys` package.

[47]Selection of files are done by looking at already existing tool results, stored in the «history» pane on the right menu at the HyperBrowser.

Learning Tool

The tool enables learning an hypothesis h from MLL measures (as textfiles) and genomic tracks provided by the HyperBrowser framework. The only additional (and mandatory) parameters are thus selecting the genomic context and range to learn from. In addition, it is strongly recommended that (at least) the machine learning algorithm to be used is selected. However, it is automatically is guessed[48] by the application if not explicitly selected.

Even though the application is built for analyzing the human genome, it is generalized to work on any genome provided by the HyperBrowser framework.

Only a single explanation or response track may be selected. To represent more tracks, a generalization, as shown in section 3.2.4, may be used. A track may be uploaded using the existing functionallity of the HyperBrowser.

The tool parameters are:

Genome
: Mandatory. Any genome provided by the HyperBrowser framework, including the human genome.

Chromosome
: Mandatory. Select either a single chromosome or all chromosomes from the selected genome.

Explanation Track
: Mandatory. A single GTrack of type S, US, P, VP, or F.

Response Track
: Mandatory. A single GTrack of type S, US, P, VP, or F.

Explanatory measures
: Mandatory. A textfile containing MLL measures with possibly associated transformations, to be used as explanation measures (features) on the explanation track.

Response measures
: Mandatory. A textfile containing MLL measures with possibly associated transformations, to be used as response measures (concepts) on the response track.

Learning algorithm
: Optional. The machine learning algorithm for learning an hypothesis h. It may be either of the 7 algorithms from section 3.3. The default selection is «Autodetect», which (naturally) aims to automatically select a proper algorithm to learn h.

Lowest data-imbalance ratio
: Optional. An integer r, to make all minority classes of ratio less than $1:r$ request imbalance «actions». The appropriate action is set by the «Data

[48] The application automatically selects a proper algorithm based on the explanation track format and feature- and class-count.

imbalance enforcement factor» parameter. Default value of r is «None» which skips the imbalance-step.

Data imbalance enforcement factor

Optional, iff a lowest data-imbalance ratio r is not «None». An integer factor f ($f \geq 2$) creates $f \times |c|$ synthethic samples using SMOTE, where $|c|$ is the number of samples in a subclass c. After creating the synthethic samples, the tomek-links are removed, resulting in a new sample set c' where $|c| \leq |c'| \leq |c| + f \times |c|$.

Regularization value

Optional. An floating value v ($v \geq 0$) used for counteracting overfitting. For $v > 0$, increasingly stronger attempts of counteracting overfitting is performed. If v is too large, then underfitting may be occur. If $v = 0$, no counteraction is performed.

Maximum iterations

Optional. If the selected algorithm demands long executions when learning h, then setting a lower iteration number will in some cases stop the learning process at an earlier stage. Thus, it may influence the performance of h.

Learning curve

Optional. Select an inspection level for computing the learning curve. The available levels are «none», «low», «medium» or «high». The «low» level only evaluates the learning progress 10 times, while «medium» evaluates 100 times and «high» a 1000 times. If the «none» level is selected, then no learning curve is generated.

Cross validation percent

Optional. A percent p ($0 \leq p < 1$) which determines the cross validation size of the translated dataset. The training dataset size is then the remaining $(1 - p)$ percent. The default value is 0,2 (20%).

Grouping strategy

Optional. A reduction reduction strategy, for coping with replicated samples in the translated dataset.

Grouping strategy parameters

Optional. Additional parameters to specify or configure the selected grouping strategy.

Start position

Optional. An integer s denotes the starting position (chromosome coordinate) of where to learn from. It is an offset of the selected genomic context. The default value of s is 0, which is no offset.

End position

Optional. An integer e denotes the ending position (chromosome coordinate) of the range to learn from, within the genomic context. The default value of e is the full length of the genomic context.

In addition to the list, some other (implementation specific) parameters may extend the tool. Two such extra parameters are the «explanation track conversion» and the «response track conversion». They are added for optional type conversion of the explanation track and the response track.

Prediction Tool

The tool produces estimates and predictions of the concepts (classes) learned by an hypothesis h of the «Learning Tool». To follow the design guildelines, it does not limit the prediction range to be within the range used for learning h. Thus, it requires the user to interpret any results and justifying them. In general, the biggest concern is determining if a prediction suffer from extrapolation. It may occur if the training set data is differing from the prediction set data. However, a good choice is usually to make predictions on similar types of data as h was learned from. The tool parameters are:

Hypothesis
Mandatory. A learned hypothesis h, the result of the «Learning Tool».

Genome
Mandatory. Any genome provided by the HyperBrowser framework, including the human genome.

Chromosome
Mandatory. Select either a single chromosome or all chromosomes from the selected genome.

Prediction Track
Mandatory. An annotation track for predicting its response values according to h. The annotation track may either use the same track as used to learn from, or another track[49]. Defaults to selecting the same track as was used to learn from.

Answer measures
Mandatory. A selection of wheteher the «real» answers of the prediction track should be computed, in order to do evaluations on it. Default value is «No», which does not perform any evaluations.

Start position
Optional. An integer s denotes the starting position (chromosome coordinate) of where to learn from. It is an offset of the selected genomic context. The default value of s is 0, which is no offset.

End position
Optional. An integer e denotes the ending position (chromosome coordinate) of the range to learn from, within the genomic context. The default value of e is the full length of the genomic context.

[49]Selecting another track may be done by selecting «No» in the additional (and optional) «Prediction same track?» parameter.

Export Tool

The tool both exports the results of the «Prediction Tool» and enables visualization of predictions and estimates for each concept (class). The predictions results are represented in valid GTrack format to be used with other tools or diverse functionallity on the HyperBrowser.

In addition, the tool has some parameters for «zooming» into a specific range of the results. The default value is to visualize the full result range. However, if a smaller range is of more interest, the zoom may be used to get a higher resolution and a better look at the data. The parameters are:

Prediction
: Mandatory. A prediction generated by the «Prediction Tool».

Export option
: Mandatory. The prediction content to be exported or visualized. One of the following:

  Probability Track
  : Export the class probabilities for c at each position in the prediction track range.

  Class Prediction Track
  : Export the predicted classes for each position in the prediction track range.

  Region Track
  : Export the region, if which the prediction was done.

  Answer Track (with correct elements)
  : Export the (correct) classes from the response track.

  Probability Track (histogram distribution)
  : Visualize the probabilities for a class c as a histogram.

  Probability Track (line graph)
  : Visualize the probabilities for a class c as a function (line) graph.

Class identifier
: Mandatory. If any of the export options rely on exporting a special identified class c, then its identifier must be specified. There may be n classes, and it must therefore be specified which class to export.

Start zoom position
: Optional. The position s to start exporting data from, where $0 \leq s \leq n$ and n is the length of the prediction track. Furthermore, s must be smaller than the «End zoom position» parameter. Default value of s is 0.

End zoom position
: Optional. The position e is the end position for exporting data. If n is the length of the prediction track, then $s \leq e \leq n$ where s is the «Start zoom position» parameter. Default value of e is n.

Decimal

Optional. Enables reducing the detail level when possibly visualizing millions of position values. If a decimal value d is set, any subsequent points agreeing on the first d decimals (at all coordinates) are not shown. The simplification is made due to the definition of a straight line. The points on a straight line does not need to be drawn if the starting- and end-point is drawn.

3.4.4 Usage and interaction

The application has been assembled with the focus of giving its users the key functionality to solve their genomic data problems using machine learning, with as little pre-knowledge about it as possible. In order to make it possible, the application tries to automatically choose as many of the optional parameters as possible, by defaulting to "standard" or likely selections. These selections aim to be useful, but does not at any means claim to be optimal choices. By this, the basic use, aims to fit the users of no or little machine learning knowledge, while the advanced use targets users of more expertise.

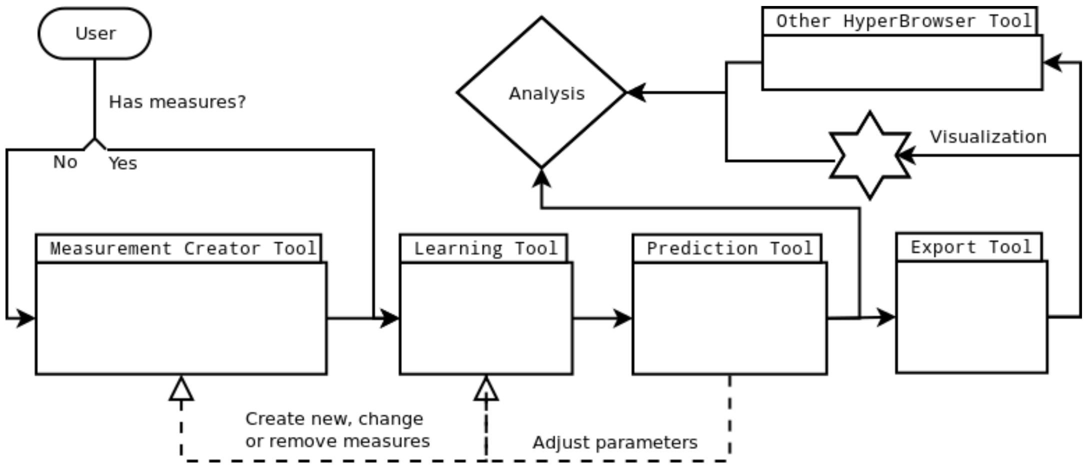


Figure 3.31: The workflow of the machine learning application in regards to the HyperBrowser environment. The solid lines show the basic use. The dashed lines display more advanced use.

Basic use

The basic use, is the applications most simplistic use, with as many of the optional parameters chosen by the application. The workflow of learning, predicting and exporting application content is shown as the solid line in figure 3.31. The smallest set of parameters which need to be selected is the mandatory ones (in each tool). This is the minimum selections which needs to be done, in order to gain some results.

Advanced use

An important part of machine learning, when learning a hypothesis, is the error analysis. It is through error analysis, that most of the optimization and fine-tuning of any hypothesis are done. Reasons for misclassifications are here being spotted by trained experts.

Providing general purpose features may not always be enough to learn a decent hypothesis. For instance, features may fail to catch certain events, which is vital for classifying a sample correctly. This calls for another alternative to simple point-and-click user interface. Therefore, another advanced learning parameter is added, for users demanding more flexibility, namely the option to manually create features and transformations. The parameter expands the basic workflow with an alternative route for adding specialized features. It is optional, so it has no effect on the basic use and would appear almost invisible to the unexperienced user. For working with this, the machine learning language has been created.

The advanced use is shown as the dotted line in figure 3.31. At any time, the user may jump back and forth between the tools, re-starting any of the processes, changing parameters, prediction another dataset and so forth.

3.4.5 Learning tool pipeline

Each of the four tools is implemented to use a pipeline of steps to compute their results. Amongst the many various processing steps provided by the application, some steps are optional and only processed if certain parameters are set. The user interface of the implementation offers this cind of parameter selection.

The «Learning tool» is the largest and thereby the most complex pipeline of all the tools. It is shown in figure 3.32. The pipeline is relativly complex by having many optional steps. The many optional steps serve the flexibility goal. Some pipeline steps are automatically selected based on other selections, but could have been optional. For instance, selecting the «balance matrix» step automatically selects the «Tomek link removal» step after the «Addition of synthethic samples SMOTE» step. This is a design choice, soly based on the assumption that it would be beneficial to clean up noise and border samples after adding synthethic ones.

It is vital that all routes of the pipeline work together. Concretely, the ouput of any pipeline must be an exeptable input of the optional next steps in the pipeline. The design showed in figure 3.32 allowes this.

3.5 The machine learning language

The machine learning language (MLL) is implemented as a «proof-of-concept» programming language for creating features and transformations for use within the machine learning application, presented in the thesis. It is created as a sublanguage of Python[50], and is therefore an interpreted language. It is written using uppercase letters only. In addition, to run like python code, it has some

[50]Python version 2.7.3.

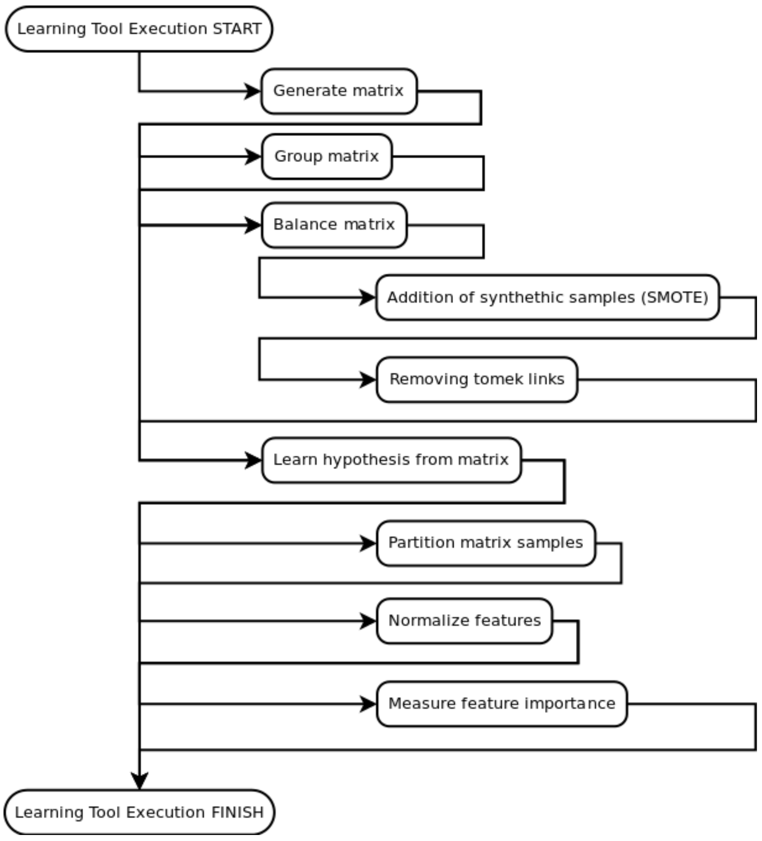

Learning Tool Execution START
Generate matrix
Group matrix
Balance matrix
Addition of synthethic samples (SMOTE)
Removing tomek links
Learn hypothesis from matrix
Partition matrix samples
Normalize features
Measure feature importance
Learning Tool Execution FINISH

Figure 3.32: The pipeline of the machine learning application.

restrictions and some added macros. Normal `Python` operators may be used, but thay should be in uppercase letters. The macros are used to enable the user to make use of the variables of the MLTrackState. A `Python` interpreter class is built as a «proof-of-concept» in order to show its usage.

The language has added some macros to be used to access the properties of the MLTrackState API. However, any `Python` code should work «out of the box» if no macros or reserved symbols are used in a way which is not supported. While the language is a «proof-of-concept», it may raise some yet undiscovered challenges.

The smallest possible script for creating a feature is shown in figure 3.33. A fully compliant feature script with a logarithmic transformation is shown in figure 3.34.

MLL Example 1

```
FEATURE BEGIN
    BODY BEGIN
        OUTPUT 1
    BODY END
FEATURE END
```

Figure 3.33: MLL sample-code for building a feature outputting 1 at all positions in the track view.

MLL Example 2

```
FEATURE BEGIN
    CONFIG BEGIN
        UNDEFINED = 105
        TRANSFORMATIONS BEGIN
            TRANSFORMATION BEGIN
                CONFIG BEGIN
                    SITUATION = 'ALL'
                    UNDEFINED = 210
                CONFIG END
                BODY BEGIN
                    OUTPUT LOG( VALUE )
                BODY END
            TRANSFORMATION END
        TRANSFORMATIONS END
    CONFIG END
    BODY BEGIN
        IF STATE IS NONE:
            OUTPUT UNDEFINED
        ELSE:
            IF STATE.IS_INSIDE_SEGMENT:
                OUTPUT 1
            ELSE:
                OUTPUT 0
    BODY END
FEATURE END
```

Figure 3.34: MLL sample-code for building a fuller feature, with the logarithmic transformation added in the config block. The feature outputs 1 if the current position is inside a segment, or otherwise 0.

3.5.1 Structure and syntax

The `FEATURE` and `TRANSFORMATION` are both structures which is used to map to the application API. The `FEATURE` maps to the `MLMeasure` API, while the `TRANSFORMATION` maps to the `MLTransformation` API. Both structures encapsulates its code by `BEGIN` and `END` clauses, as shown for the `FEATURE` structure (on line 1 and 5) in figure 3.33. Concretely, a transformation uses `TRANSFORMATION BEGIN` and `TRANSFORMATION END`, while a feature uses `FEATURE BEGIN` and `FEATURE END`. Each structure has two inner elements, namely the optional configuration clause `CONFIG`, and the mandatory body clause `BODY`. Both of the elements encapsluates its content in the same way as the structures.

The configuration clause `CONFIG` specifies an undefined value, or adds transformations and meta-data to it.

The `BODY` clause contains the MLL code which evaluates and outputs a value for each position of the track view. There are multiple values available through the built-in `STATE` macro.

3.5.2 Macros

All macros are read-only. Variables may be created as in the same way as in `Python`. However, any variable must be in upper case letters and may never contain any of the macros as its substring.

These are the global `FEATURE` macros, which are available «as-is» and may be used directly within the `BODY` clause.

STATE MLTrackState instance.

SIZE Integer. The range length of the genomic context.

POSITION Integer. A relative position inside the interval, from 0 to SIZE.

These are the `STATE` specific macros. Any inner macro, e.g. `MACRONAME`, should be addressed as `STATE.MACRONAME`.

SIZE Integer. The length of the `STATE` structure.

START_POSITION Integer. The position where the `STATE` starts inside interval.

END_POSITION Integer. The position where the `STATE` ends inside interval.

OFFSET Integer. The difference between `START_POSITION` and `POSITION`.

FUTURE_POINT 2-Tuple : (position,value). The future point is represented as a tuple, not saying if it is a point or segment. If no future point, then None.

FUTURE_POSITION Integer. The position inside the tuple of `FUTURE_POINT`.

FUTURE_VALUE Float. The value inside the tuple of `FUTURE_POINT`.

LAST_POINT 2-Tuple : (position,value). The last observed point or segment. If no such point, then None.

LAST_POSITION Integer. The position inside the tuple of `LAST_POINT`.

LAST_VALUE Integer. The position inside the tuple of `LAST_VALUE`.

IS_POINT Boolean. True if the element is a point, else False.

IS_SEGMENT Boolean. True if the element is a segment, else False.

IS_INSIDE_SEGMENT Boolean. True if the element is inside a segment, else False.

IS_OUTSIDE_SEGMENT Boolean. True if the element is outside a segment, else False.

Chapter 4

Results

4.1 Application

The application is implemented inside The Genomic HyprBrowser and is publicly available at `http://hyperbrowser.uio.no/ml/`[1]. It offers researchers, or anyone interested, the opportunity to use the application on their own biological challenges.

To illustrate the application usage, the rest of the chapter introduces an use case for analyzing a concrete biological challenge.

4.2 Use case

The use case idea is inspired by the discovery in [33] of a possible relationship between vitamine D receptor (VDR) binding sites and regions associated with diceases such as diabetes, cancer, multiple sclerosis (MS) or others.

Here, the aim is to learn a relationship between vitamine D receptor binding sites (points) and regions (segments) which could be associated with multiple sclerosis (MS). It is a binary classification problem since the goal is to predict the existence of a vitamine D receptor binding site for each individual base-pair position.

In biological terms, the use case aims to learn a model which could predict if a vitamine D receptor site is present or not at any base-pair position of a genomic track containing regions which could be associated with MS.

Abstractly, the problem formulation is to detect the relationship of points (P) and segments (S).

The focus of the use case is to present a walk-through of the methodology to show the basic use of the application. The focus is not to analyze the biological results or optimize any models thereof.

4.2.1 Data access

The actual results of the use case, with all details, are available at the HyperBrowser «case-page»: `http://hyperbrowser.uio.no/ml/u/fredrik-haaland/h/use-case` .

[1]Referenced to as the «homepage» in the use case.

Abstract question

Is machine learning able to learn a relationship between points and segments, and thus predict the existence of the points in a way which is better than by chance?

Biological question

Could a relationship between VDR binding sites and regions associated with a disease such as MS be learned so that the binding sites achieve higher probabilities than average?

4.2.2 Walk-through

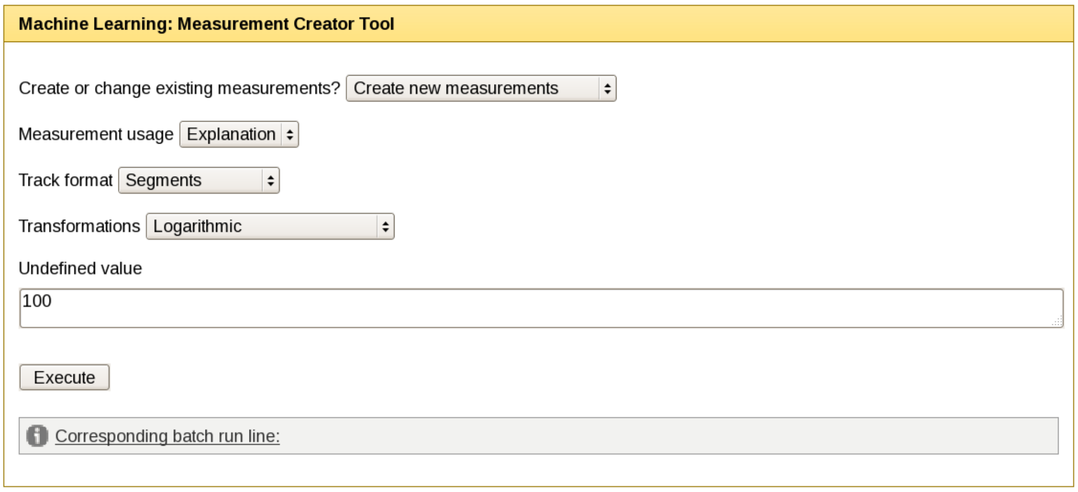
Selecting parameters and genomic tracks

The selected biological (genomic) data was provided by The Genomic Hyper-Browser framework, where the «hg18» (NCBI36) human genome is selected as the base genome parameter in all tools throughout the use case.

The chromosome 6 (chr_6) with length of roughly 170,000,000 base-pairs was selected as the range to be learned. The explanation track was selected as a gene track of chr_6[2], while the response track was selected as vitamine D receptor binding sites[3]. The track files are accessible at the case-page, found in section 4.2.1 on the previous page.

Creating measures

Measurements was created using the «Measurement Creator Tool»[4].



Machine Learning: Measurement Creator Tool

Create or change existing measurements? Create new measurements

Measurement usage Explanation

Track format Segments

Transformations Logarithmic

Undefined value

100

Execute

Corresponding batch run line:

Figure 4.1: Creating use case measures with the «Measure Creator Tool».

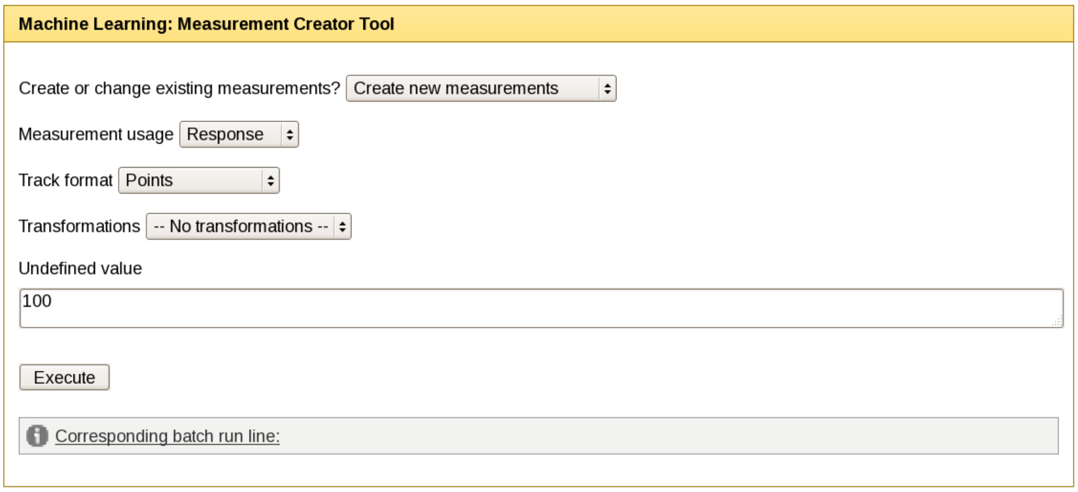
[2]To select the track in the HyperBrowser, select: «Genes and gene subsets» -> «Genes» -> «Nscan».

[3]The track is a privatly added. HyperBrowser selection: «Private» -> «GkMs» -> «VdrRegs». The track is originally in a segment format, so the midpoint of the segments were selected as the points to be learned.

[4]The «Measurement Creator Tool» is found at the left panel of the homepage under the «HYPERBROWSER TOOLS» -> «Restricted and experimental tools» -> «Generic Tool 1».

Feature measures was created by selecting the «measurement usage» as «explanation», the «segment» track format, and the «logarithmic» transformation was added to all the measures. The undefined value was selected to be 100 for both the features and the transformations. The «Point exists» response meaure was selected, because of the aim to learn and predict the existence of points. Figure 4.1 on the facing page show the selections.

Response measures was selected very similar to the feature measures. The «measurement usage» was selected as «response» and «points» was selected as track format in stead of «segments». No transformations were added. The undefined value is left unchanged at 100. Figure 4.2 shows the selections.


Machine Learning: Measurement Creator Tool

Create or change existing measurements? Create new measurements

Measurement usage Response

Track format Points

Transformations -- No transformations --

Undefined value

100

Figure 4.2: Creating use case measures with the «Measurement Creator Tool».

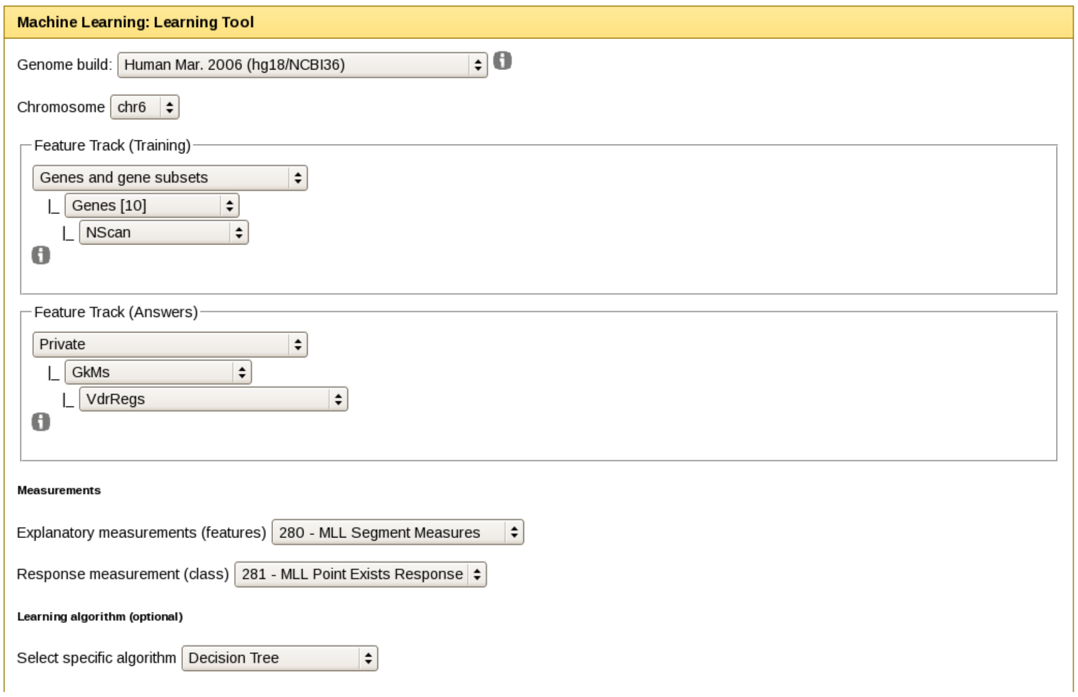
Learning the relationship

A minimal amount of parameters are selected to show the basic use of the «Learning Tool». Thus, all mandatory parameters are selected in addition to a few others. Concretely, the «medium» learning curve is selected in order to display a more detailed learning curve (for illustrative reasons) than the default parameter. Also, the «Decision Tree» learning algorithm is selected and the translated matrix is set to «prune»[5] off samples as a post processing step. The mandatory selections in addtion to the learning algorithm are shown in figure 4.3 on the following page.

The execution result of the «Learning Tool» contains the learning curve, the feature similarity measures and more. Allthough chr$_6$ has a range of 170,000,000 , the post processing step has «pruned» the sample dataset down to 5,439 samples. The samples are partitioned into a training set of 20% and a cross validation set of 80%, shown in table 4.1 on the next page.

The learning curve is shown in figure 4.4 on page 81. The «training» error (in accuracy) increases while learning increasingly more samples. The x-axis show the intervals of added samples, while the y-axis show error rates (as decimals).

[5]Pruning is performed by selecting the decimal $d = 0$ to remove all samples which agree on all feature values without looking at any decimal differences.


Machine Learning: Learning Tool

Genome build: Human Mar. 2006 (hg18/NCBI36)

Chromosome chr6

Feature Track (Training)
Genes and gene subsets
|_ Genes [10]
|_ NScan

Feature Track (Answers)
Private
|_ GkMs
|_ VdrRegs

Measurements

Explanatory measurements (features) 280 - MLL Segment Measures

Response measurement (class) 281 - MLL Point Exists Response

Learning algorithm (optional)

Select specific algorithm Decision Tree

Figure 4.3: Selecting mandatory parameters for use with the «Learning Tool». In addition, the optional «learning algorithm» parameter is selected to be «Decision Tree».

Dataset / Class	Negative samples	Positive samples
Training set	4153	1049
Cross validation set	198	39
Sum	4351	1088

Table 4.1: The partition of the use case dataset.

A possible explanation of the «training» error could be generalization difficulties. For the model to comprehend more samples, the model may not be able to perfectly separate the amount of closely connected samples. The model may thus select to trade some current classification error against the possibility for better predictions of other unseen samples. The training error reaches a rate of ~3% when having learned all the 100 intervals (4153 samples).

The cross validation (accuracy) error is rather stable at ~6% while increasingly more samples are added. This may be a second indication of that the model is favouring generalization, since the cross validation set is not used to train the model.

Overall, the two error rates seem to move towards each other. It may be an indication of that the model is learning the relationship well. However, accuracy may not be the best way of measuring the learning performance, since it does not distinguish between false positives and false negatives.

The features are the same and has the same numbering as the ones from section 3.2.1 on page 45. Feature F_0 and F_1 are the ones who have both most

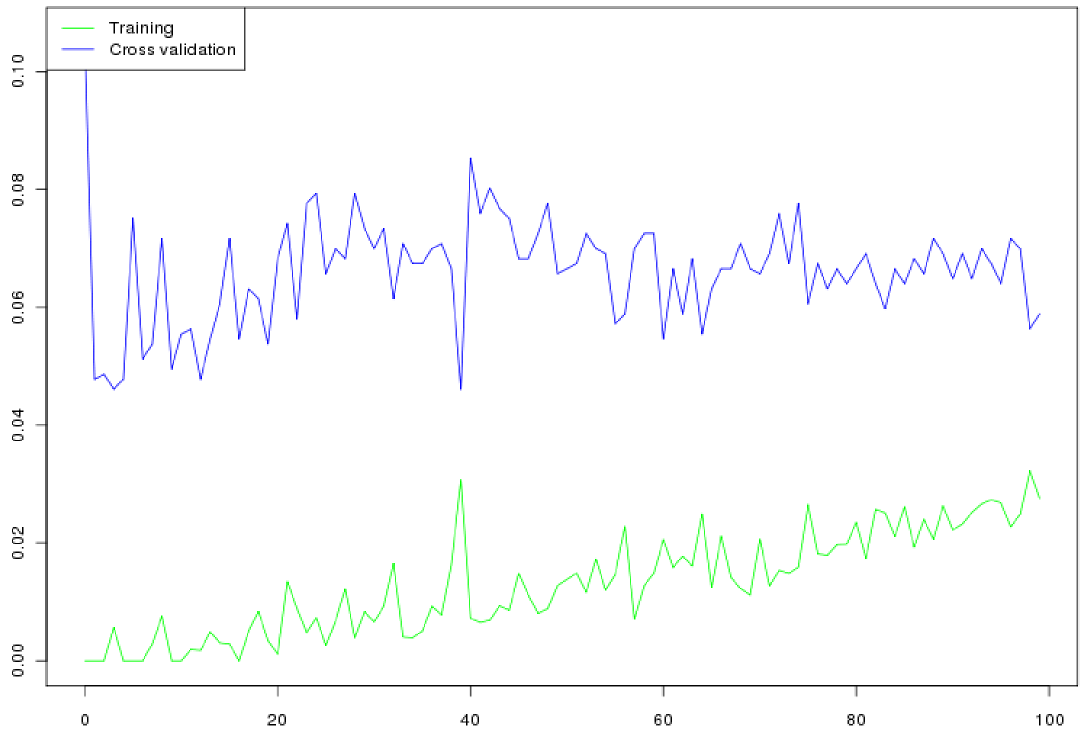


Figure 4.4: The use case learning curve. The «training» line display the error in accuracy as increasingly more samples are learned. The «cross validation» line display the accuracy error of the cross validation set.

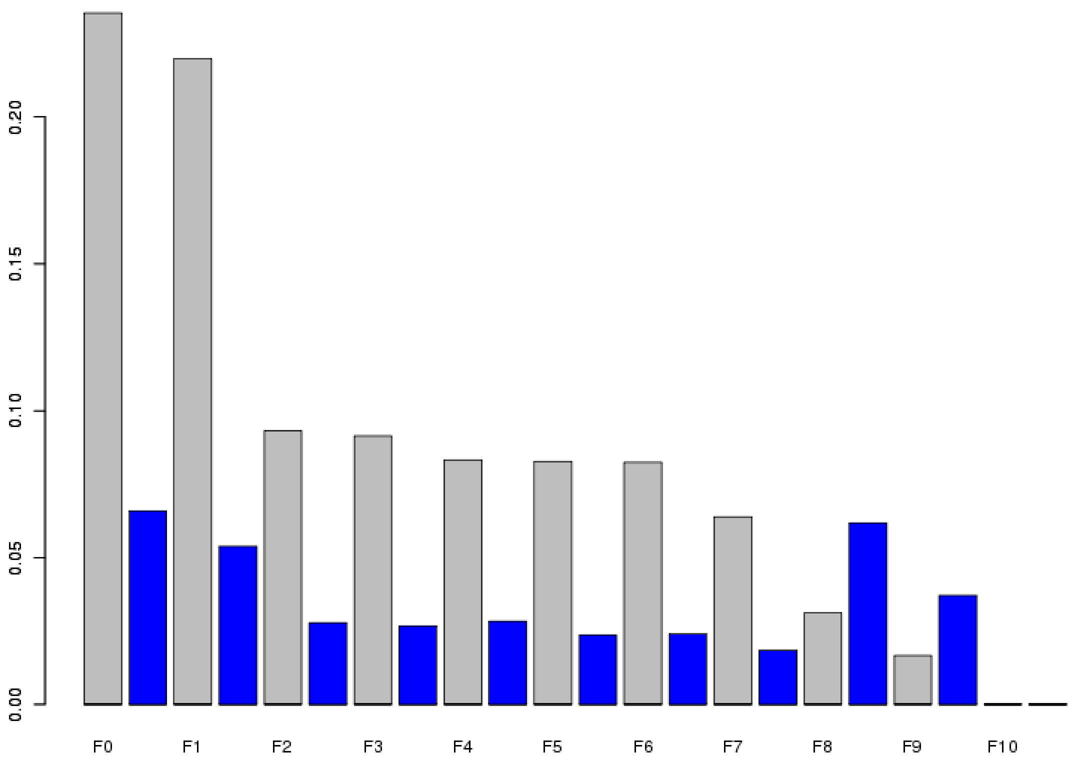
information gain and impurity. These features are based on the mathematical similitude property which, in turn, may be harmful for generalization if used to predict other ranges at a later stage. However, in this case the objective is to predict probabilities within the learned track. Thus, the similitude property could hardly be justified. Feature F_8 and F_9 (and possibly F_{10}) are the features which has a much higher information gain than impurity. Thus, it looks like these are the best measures. Feature F_2 and F_3 seem to perform reasonably well, while F_4, F_5, F_6 and F_7 are questionable. The feature similarities are shown in figure 4.5 on the following page.

Predicting point existence

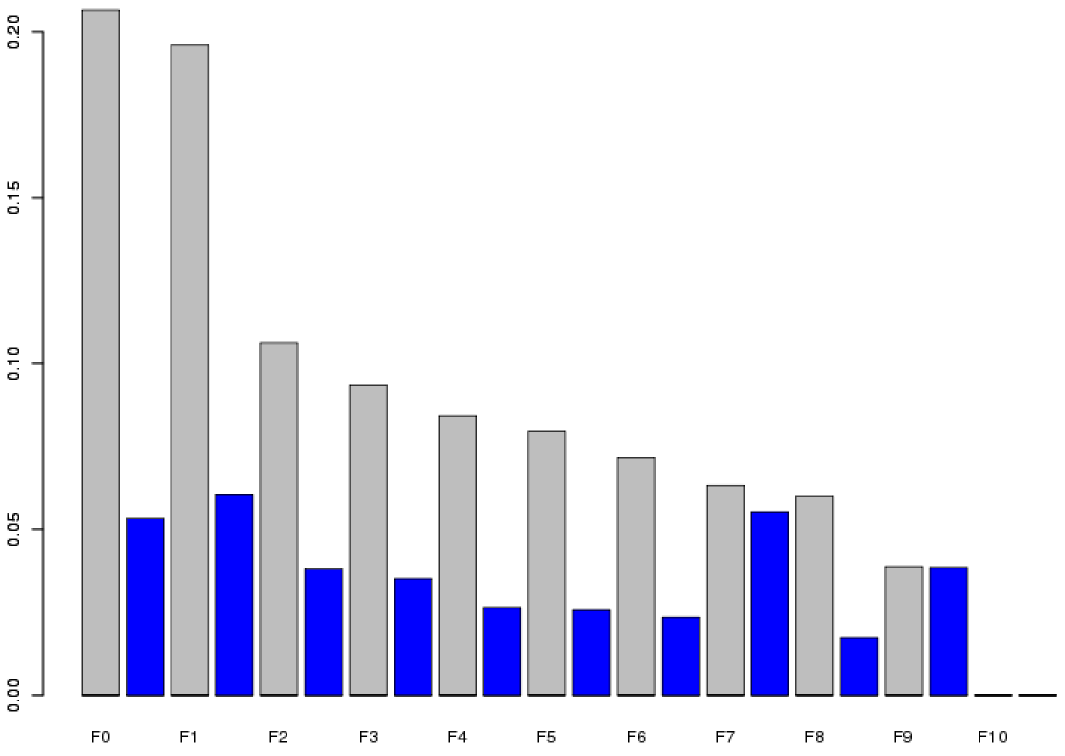
The «learned hypothesis» from the «Learning Tool» was selected in the «Prediction Tool» for making predictions. All of the parameter selections are shown in figure 4.6. The prediction track and range were selected as the ones that were used for learning. Since the (full) range is selected, no «zooming» is performed. Consequently, the «start position» is 0[6] and the «end position» is omitted[7].

A list of some of the prediction results is shown in table 4.2 on page 83. Allthough the accuracy is high (~90%), the precision and recall is rather poor. The average prediction for the positions which predicts point existence (class=1)

[6]A «start position» of 0 equals omitting it.
[7]By omitting the «end position», the full range length of chr_6 is used by default.



(a) Information impurity (gini).



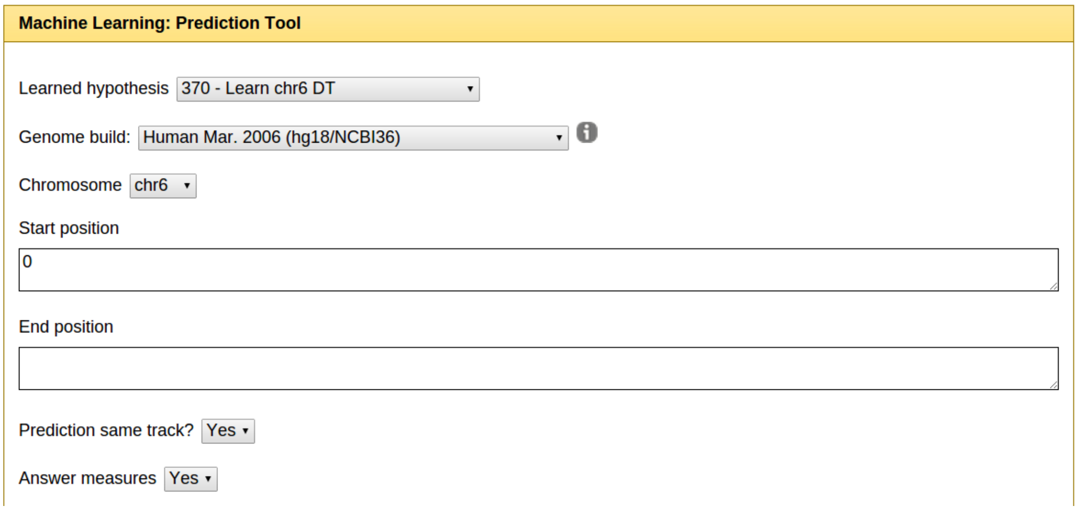
(b) Information gain (entropy).

Figure 4.5: Feature similarity measurements of the use case features.

is ~59%. By contrast, the average prediction for the positions which predicts no point existence (class=0) is ~80%. Thus, the model is ~59% sure of point existence and ~80% of no point existence whenever it makes the prediction, on average.

Accuracy	0.90071025
Precision	0.00000518
Recall	0.37130801
F-score	0.00001037
Prediction average : Class 0	0.80413736
Prediction average : Class 1	0.59311413

Table 4.2: Selected use case prediction results.


Machine Learning: Prediction Tool

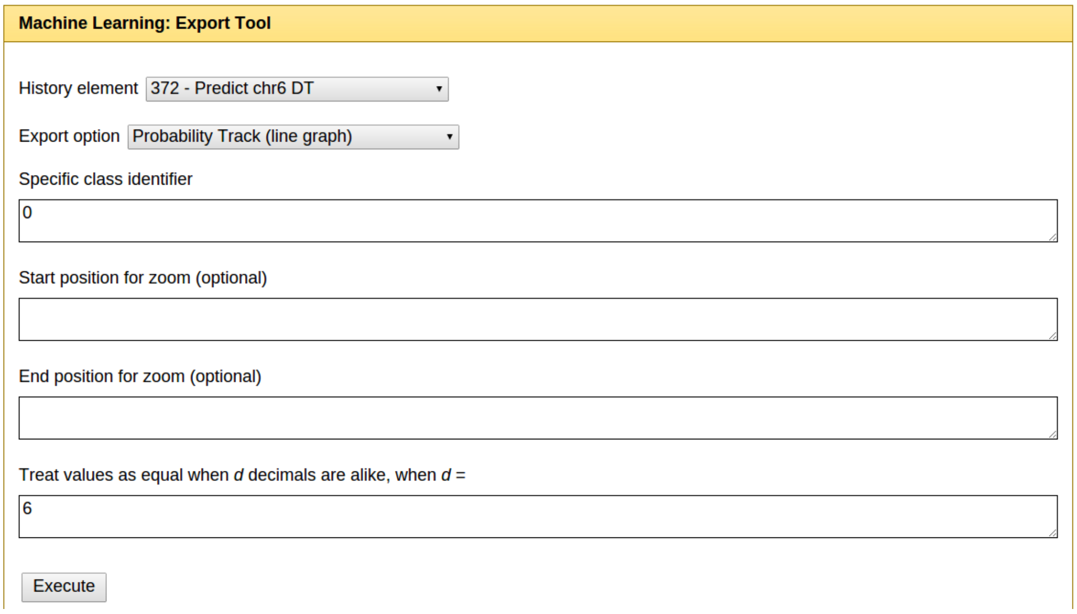
Learned hypothesis	370 - Learn chr6 DT
Genome build:	Human Mar. 2006 (hg18/NCBI36)
Chromosome	chr6
Start position	0
End position	
Prediction same track?	Yes
Answer measures	Yes

Figure 4.6: The selected use case parameters for the «Prediction Tool».

Prediction extraction

To gain a better understanding of the predicted results, the «Export Tool» provides options to export or visualize the data.

To get a view of the base-pair probabilities for all positions, a probability graph of point existence for each position might be a good selection. The resulting prediction from the «Prediction Tool» is selected as the «history element». Furthermore, the «probability track (line graph)» is selected as «export option». The parameter selection is shown in figure 4.7. The probabilities are shown in figure 4.8 on the following page.


Machine Learning: Export Tool

History element: 372 - Predict chr6 DT

Export option: Probability Track (line graph)

Specific class identifier: 0

Start position for zoom (optional):

End position for zoom (optional):

Treat values as equal when *d* decimals are alike, when *d* = 6

Figure 4.7: The selecting use case parameters for the «Export Tool».

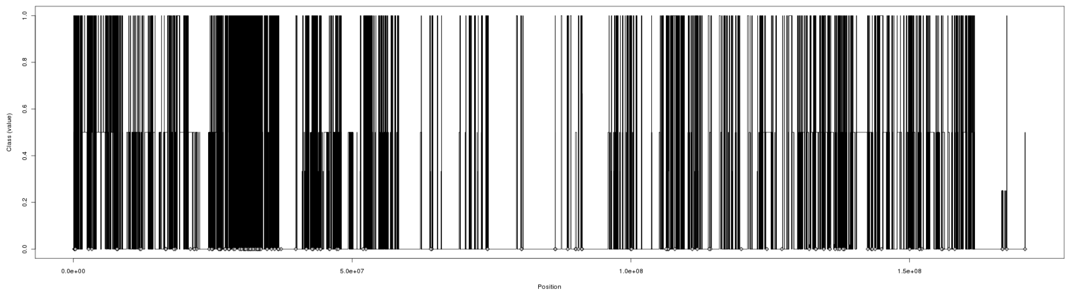


Figure 4.8: Use case probabilities for point existence at all positions of chr_6 by the «Decision Tree» algorithm. The x-axis represents the track positions while the y-axis show the probabilities, from 0 - 100%. The lower diamonds along the x-axis display the learned points. The use-page contans a higher image resolution.

Evaluating predicion results

A simple way to predict point existence could be done by counting the points which overlaps[8] with the segments and predict the point existence by a constant probability. For instance, chr_6 has 237 points and 1020 segments. The segments occupies 78,311,521 positions which means that the rest of the 92,588,471 positions are left open. The overlap of points inside segments is 130 points for chr_6. Then, based on how many points which overlaps, the probabilities for all positions i could then be computed as:

$$P(i) = \begin{cases} \frac{\text{points inside segment}}{\text{occupied positions}} = \frac{130}{78311521} = 0,00000166, & \text{if } i \text{ is inside} \\ \frac{\text{points outside segment}}{\text{open positions}} = \frac{107}{92588471} = 0,000001156, & \text{if } i \text{ is outside} \end{cases}$$

[8]An overlap occur whenever a position i is occupied by both a point and a segment in their separate tracks.

The probabilities for a point existence would, by using the simple model, be 0,00000166% for all positions inside a segment and 0,000001156% for all positions outside a segment. However, the machine learned model for chr_6 is able to predict this probability individually for each position. As an example, the 458th segment of chr_6 starts at position 42,824,362 and ends at position 42,944,268 , which gives the segment a length of 119,906 positions. It overlaps with two points at postion 42,859,126 and 42,859,531. The probability for a point existence on the single segment is thus $\frac{2}{119906}$ = 0.00001668% for all the segments positions.

However, the learned model of chr_6 could then be used to predict the probabilities for the 458th segment of chr_6. The figure 4.9 shows the prediction. The dotted line is the simple constant probability, while the solid line is the (normalized) model probabilities. The probabilities are thus higher than the constant probability at some areas and lower at others. Allthough it gives the highest probability for the two overlapping points, it also gives high probabilities at other positions without any learned points.

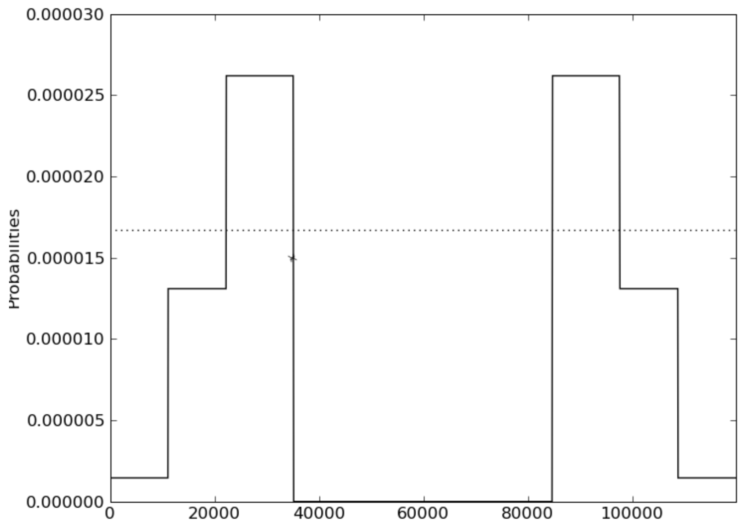


Figure 4.9: Comparison of probabilities of a simple model (dotted line) and the learned model (solid line). The markers are two points at the (relative) position 34,764 and 35,169. The y-axis show the probabilities (as decimal numbers). The x-axis is the relative positioning starting from position 42,824,362 which is the start of the 458th segment of chr_6.

Figure 4.10 on the following page shows a histogram of the model probabilities for point existance of figure 4.9. It shows that most positions have 0% probabilies (for point existence) which is consistent with the existence of only 237 positions. The second largest bar is of 50% probability which is ok since it is not pointing either way. The third largest bar of 100% probability is a bit high by considering the relative few existing points. Allthough there is a small chance that the other positions with high probabilities are existing points which are yet to be discovered, the chance is much higher for an inaccuracy in the model.

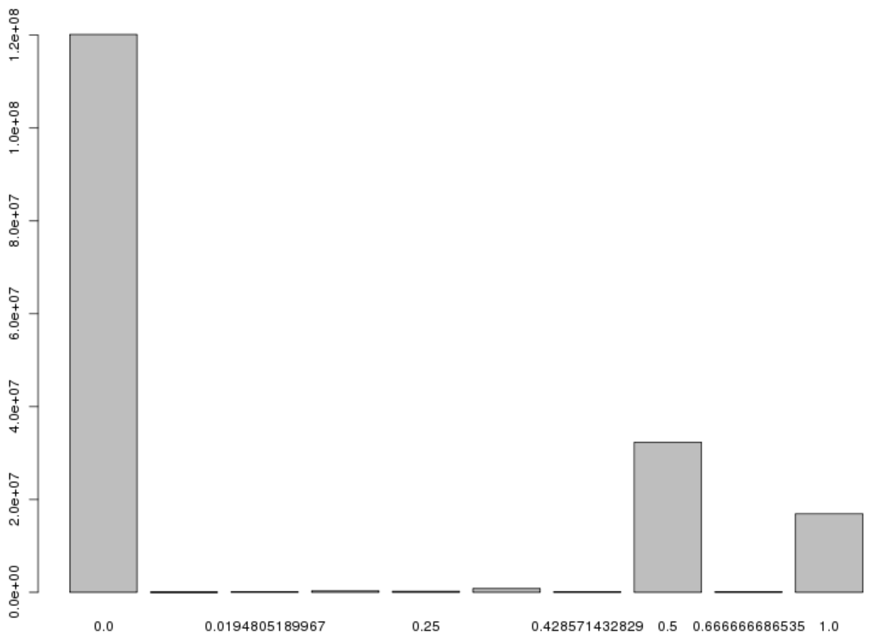


Figure 4.10: A histogram of the models probabilities for all positions of chr_6. The zero probability column is highest, as expected since there is only 237 points that exists.

The average (point existence) prediction for the positions of the known 237 points is ~59,5%. The average prediction for all the positions is ~19,5%. Thus, it might seem that the model is capable of learning the relationship between the points and segments, and thus predict the existence of the points in a way which is better than by chance. However, the histogram show that the two closest prediction values are 50% and 66,7%. It might not be easy to determine which of these values to set as the threshold for predicting point existence. A threshold of 50% would predict 49,273,162 points which is tremendously more than 273. However, 50% is less than the average of ~59,5% and may thus be a rather poor choice. A threshold of 66,7% would predict 16,968,556 points which is still an enormous amount of possibly false positives.

4.2.3 Improving the model

Allthough the model has been evaluated and some results has been provided, there are a few steps that possibly could improve the models performance. To start off, the reduction of the dataset down to 5,439 samples is a rather dramatic choice. By selecting an higher decimal d for pruning «equal» samples, then both more samples and higher level of detail per sample is aquired. Also, a second review of the used feature measures could benefit the learning. New and better features may be applied to keep more information when translating the dataset.

Furthermore, a more balanced dataset could possibly enhance the learning process. The current balance ratio is $\sim 1:0.05$ $(4153:198)$. The positive samples could probably be doubled or trippeled to see if it has a constructive effect.

Finally, the predictions could possibly be more nuanced in combination with additional (machine learned) model probabilities as illustrated in section 3.3.8 on page 64. Also, a prediction could possibly be used as an addition to other statistics, e.g. for removing noise.

Part III

Discussion

Chapter 5

Challenges

5.1 Translation and representation

The «translation» of genomic data into a representation for use with machine learning algorithms has been challenging. Representing base-pairs as the smallest building blocks is done in regards of keeping the application generic. The choice of this representation could be viewed as a «safe» solution, since it opens up for the possibilty of combining the smaller building blocks into larger building blocks at a later stage. Concretely, it stores information on a base-pair level, which for instance could later be used to merge multiple base-pairs and its information into a larger region of base-pairs. In addition, the representation of base-pair level information may possibly be overwhelmingly large in terms of computer resources and could cause slow performance in both learning and prediction.

The choice of representation selected in the thesis is only one of many. For instance, a whole track could be used as a single instance instead of a base-pair. Such situations may seek to find patterns amongst a multiple of tracks to predict a «common» track. This type of representation would then be more alike a data-mining approach since aiming to detect patterns within the data. In any case, representing a multiple of tracks is still possible using the baise-pair representation, though the number of instances would result in an huge dataset. Concretely, each base-pair of every track could be translated into a matrix representation.

The capture of track properties in section 2.2.1 on page 26 does not claim to capture «all» available properties. It aims to capture the fundamental properties, such as length, positioning, distance and element specifics. However, the discovery of more properties would strengthen the foundation for creating more advanced and creative measurements.

5.2 Implementation

The implemented application is a result of following the set design goals and guidelines for making it flexible and generic. The process of creating the application has been increasingly challenging by trying to comprehend the many aspects of machine learning which has become visible along the way.

Concretely, the aspects of determining what data to be used and how it should be represented, stored and dealt with has presented many possibilities and have thus required many decisions to be made.

5.2.1 Overdesign

The wide range of possibilities lead to creating an early design which aimed to enhance «all possibilies». The purpouse of creating such an application was to aim as widely as possible in order to be as generic as possible. The design was inspired by learning from «all known sources». Concretely, the design tried to create interfaces for working with the basic entities, such as the «list» and «dict» structures as well as the «int» and «float» structure. In addition, the design tried to automatically load any `Python` class which implemented a known interface and were put in a special folder. The design was a clear «overdesign», which were actually focusing more on the idea of *being* generic than acting in a generic manner to solve diverse challenges. The design was modelled and is shown in figure 5.1 on the facing page.

Due to the overdesign, the guidelines of section 3.4.1 on page 65 was established. Amongst many changes in the design, the focus were changed to learning from only a single type of data (GTrack). Components that were built only for the sake of being generic was removed[1]. The simplified design is shown in figure 3.30 on page 66. One of the main reasons for the overdesign was probably both due to the "optimistic" approach of solving «everything» at once and the lack of experience in creating applications.

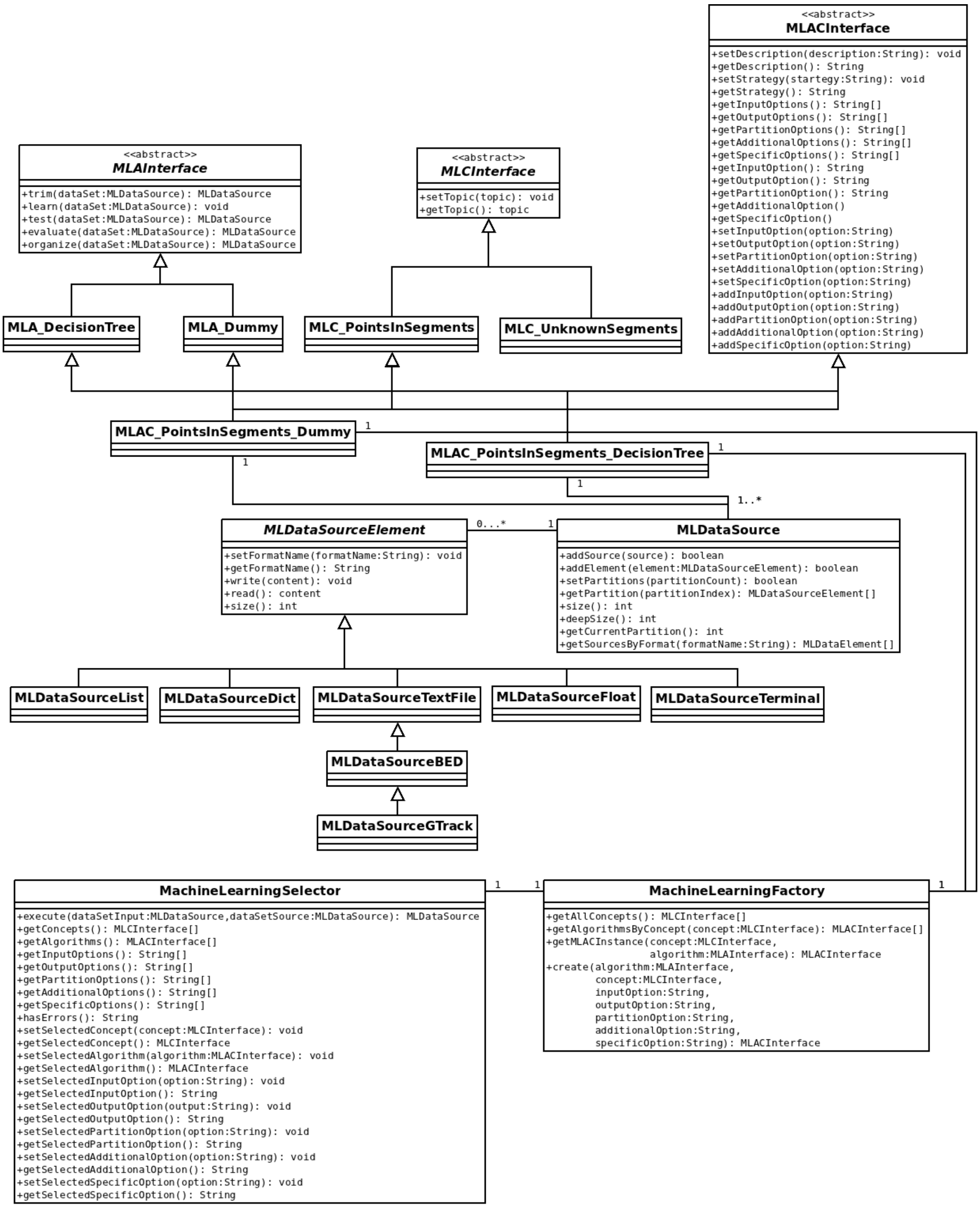
A weakness of the application is that it is currently only supporting to learn from a single explanatory track. It requires the user to represent a multitude of tracks through an combination track. Thus, there is no support for assigning only some measurements to a some of the tracks. However, this is result of trying to reduce the complexity of the early design, and has caused the application to be less flexible.

5.3 Selection of programming language

As the HyperBrowser platform has been implemented in `Python`, the choice of application programming language more or less selected from the beginning. In order to make use of the HyperBrowser interfaces and the resources thereof, the application required a `Python` compatible implementation. Implementing the application in `Python` was then an natural choice.

The nature of the machine learning is somewhat related to «number crunching». Thus, extensive use of vectorized (lower-level) libraries play a key role for the application to run efficiently. A challenge in regards to performance arises since `Python` is an interpreted language and thus not compiled. In general, the higher level languages such as `Python` runs much slower than lower level languages such as `C` and `C++`. Vectorized library bindings such as `numpy` may provide efficient solutions to number crunching. However, the SciKit library

[1]The component implementations which were removed are not published, but may be provided on request. It would only require a rollback operation in the revision control system.



<<abstract>>
MLAInterface
+trim(dataSet:MLDataSource): MLDataSource
+learn(dataSet:MLDataSource): void
+test(dataSet:MLDataSource): MLDataSource
+evaluate(dataSet:MLDataSource): MLDataSource
+organize(dataSet:MLDataSource): MLDataSource
<<abstract>>
MLCInterface
+setTopic(topic): void
+getTopic(): topic
<<abstract>>
MLACInterface
+setDescription(description:String): void
+getDescription(): String
+setStrategy(startegy:String): void
+getStrategy(): String
+getInputOptions(): String[]
+getOutputOptions(): String[]
+getPartitionOptions(): String[]
+getAdditionalOptions(): String[]
+getSpecificOptions(): String[]
+getInputOption(): String
+getOutputOption(): String
+getPartitionOption(): String
+getAdditionalOption()
+getSpecificOption()
+setInputOption(option:String)
+setOutputOption(option:String)
+setPartitionOption(option:String)
+setAdditionalOption(option:String)
+setSpecificOption(option:String)
+addInputOption(option:String)
+addOutputOption(option:String)
+addPartitionOption(option:String)
+addAdditionalOption(option:String)
+addSpecificOption(option:String)
MLA_DecisionTree
MLA_Dummy
MLC_PointsInSegments
MLC_UnknownSegments
MLAC_PointsInSegments_Dummy
1
1
MLAC_PointsInSegments_DecisionTree
1
1
1..*
MLDataSourceElement
+setFormatName(formatName:String): void
+getFormatName(): String
+write(content): void
+read(): content
+size(): int
0...*
1
MLDataSource
+addSource(source): boolean
+addElement(element:MLDataSourceElement): boolean
+setPartitions(partitionCount): boolean
+getPartition(partitionIndex): MLDataSourceElement[]
+size(): int
+deepSize(): int
+getCurrentPartition(): int
+getSourcesByFormat(formatName:String): MLDataElement[]
MLDataSourceList
MLDataSourceDict
MLDataSourceTextFile
MLDataSourceFloat
MLDataSourceTerminal
MLDataSourceBED
MLDataSourceGTrack
MachineLearningSelector
+execute(dataSetInput:MLDataSource,dataSetSource:MLDataSource): MLDataSource
+getConcepts(): MLCInterface[]
+getAlgorithms(): MLACInterface[]
+getInputOptions(): String[]
+getOutputOptions(): String[]
+getPartitionOptions(): String[]
+getAdditionalOptions(): String[]
+getSpecificOptions(): String[]
+hasErrors(): String
+setSelectedConcept(concept:MLCInterface): void
+getSelectedConcept(): MLCInterface
+setSelectedAlgorithm(algorithm:MLACInterface): void
+getSelectedAlgorithm(): MLACInterface
+setSelectedInputOption(option:String): void
+getSelectedInputOption(): String
+setSelectedOutputOption(output:String): void
+getSelectedOutputOption(): String
+setSelectedPartitionOption(option:String): void
+getSelectedPartitionOption(): String
+setSelectedAdditionalOption(option:String): void
+getSelectedAdditionalOption(): String
+setSelectedSpecificOption(option:String): void
+getSelectedSpecificOption(): String
1
1
MachineLearningFactory
1
+getAllConcepts(): MLCInterface[]
+getAlgorithmsByConcept(concept:MLCInterface): MLACInterface[]
+getMLACInstance(concept:MLCInterface, algorithm:MLAInterface): MLACInterface
+create(algorithm:MLAInterface, concept:MLCInterface, inputOption:String, outputOption:String, partitionOption:String, additionalOption:String, specificOption:String): MLACInterface

Figure 5.1: The initial design of the machine learning application.

does often require parameters to passed as the *list* structure. Thus, much type concersions is required to get the various components to work together.

In general, an vectorized `Python` implementation combines the best of the two worlds of high-level and low-level programming. A possible way of combining the «two worlds» is to use the `Cython`[2] language. It is a language which compiles `Python code` to equivalent `C` code. Even though it would be perfectly possible to implement the application in `Cython`, it would make the code less flexible and more complex to understand. It would be less flexible since parameter changes would require methods to be compiled before execution, and it would be more complex since each `Cython` module exists of three files rather than one. In addition, the most time consuming parts of the application is already either vectorized with `numpy` or using the SciKit library.

5.3.1 Working with large datasets

The application design goal of being both flexible and generic has resulted in "loosely coupled" components. A (natural) disadvantage of this design is that it makes it difficult to build components which work closely together. One negative effect of this is that sharing resources becomes difficult, if not impossible. Concretely, the memory resource, which is one of the most vital resources in machine learning, may cause a great number of challenges. As `Python` is an interpreted language, the memory lock may possibly cause two loosely coupled components to duplicate a memory area to both work on the same data, if parallelized. This is true for most variables since they usually are passed *by reference*. If a method were to change a referenced object in memory, the object might require to be duplicated elsewhere[3]. Thus, it may obtain more resources, such as more memory or disk space. The relatively simple operation of classifying a matrix might serve as a good illustration, when an algorithm and the output format is loosely coupled. Instead of iterating through each matrix row and directly writing the resultsing classes to disk (as a GTrack file), the matrix results is first stored in memory (requiring additional memory space) before it is returned to the «controller». In the next pipeline step (if no post-processing steps are required), the controller outputs the result from memory to disk. Clearly, it would be possible to optimize this operation, but it is probably not trivial if one would like to keep the flexibility of optional post-processing steps.

Many of the steps in the learning tool pipeline, shown in figure 3.32 are likely to work with large amounts of data, because the human genome itself is huge. In the case of segments and points in section 4.2 on page 77, the 6th chromosome (chr_6) has 170,899,992 baise-pairs where only 45 baise-pairs positions are occupied. Representing all occupied positions of chr_6 could result in a large matrix if having such an imbalance of open and occupied positions in addition to large gaps between the occpied positions. To represent a single measurement for each base-pair as a `32bit float` would for all positions of chr_6 require nearly 0.7 GB[4] of memory. Thus, a relatively small selection of

[2]Cython website: `http://cython.org/` (2013-04-29)

[3]In linux, a child process creation by *fork* would copy a memory area whenever it is first written to. This is known as «copy-on-write». The HyperBrower runs on a linux operating system.

[4]170,899,992 base-pairs $\times$ 32 bit/base-pair = 5,468,799,744 bits = 683,599,968 Bytes $\approx$ 0.7 GB

10 features and a single response measure would then require around 7,5 GB of memory storage alone. Furthermore, performing the calculations of learning and predicting it would require additional memory space. Finally, the whole human genome of 3,200,000,000 base-pairs would require around 12,8 GB of memory for each measurement. Thus, the amounts of data show to be rather large.

The combination of generic problem solving and large amounts of data has created a "pressure" for the individual (isolated) pipeline steps of the tools to be as efficient as possible. The total processing time of a tool is the sum of the time spent by each of the pipeline step. Due to the flexibility design goal, there has been created a number of available processing steps. Thus, as many steps as possible are left optional by default. However, the output of one pipeline step may affect the requested effort of the next. Usually, the «grouping» of similar samples may reduce the processing time if many "less informative" (redundant) samples could be removed. However, simply removing such samples could result in loss in learnability. A combination of removing samples and assigning a weight equaling the amount of removed samples using cost-sensitive learning could probably prevent such an effect. However, the SciKit implementation in the current version did not offer such functionallity. Implementing the machine learning algorithms for only adding this functionallity is a too great job and above the scope of the work of this thesis.

Some optimizations have been performed to make the isolated parts run more efficiently. Even though many futher improvements are possible, the «focus» has been on making the application work in a «reasonable» amount of time and not on the optimizations themselves.

Storing executables

Storage of large dataset results to disk has been done to improve matrix generation time. Concretely, a generated matrix is stored to disk with the filename of an *hash*[5] of the parameters used for its generation. Thus, the storage space resource is used to enable faster matrix generations when working a multiple times with the same parameters. However, the method is rather simple and does not perform any additional checks than comparing hash-values. It may therefore be improved to detect smaller changes in parameters which may be used to re-use computed matrices in other ways. For instance, changes in the "undefined value" would cause the matrix to be re-generated. A log of undefined values, could probably be used to load the matrix and replace old undefined values with new ones, rather than generating and storing a new matrix for every undefined value. Furthermore, techniques for copying subsets of a generated matrix could be done if an existing matrix contains the required range with the same parameters.

[5]An hash of numbers and integers is computed using the `sha224` of the `hashlib` in `Python`. The probability of creating an equal hash for two different sets of parameters isclose to 0 and therfore not a practical ppproblem.

The translation process

The translation process (matrix generation step), shown in figure 3.32 is the first processing step in the pipeline of the «Learning Tool». To make use of the full potential of any explanation track, it has to be accessible at any given time. Being accessible does not necessarily equal being loaded into memory, if other mechanisms are able to load the data into memory on request.

The sampling implementation of the application makes use of this by loading elements one by one from the explanatory and possibly the response track on request. They comply with the sampling strategy by providing information about the current, preceding and subsequent element at any given time, by doing file pointer jumps according to what is being requested. Currenly, only jumps to the subsequent and preceeding elements on the track is supported. Thus, it implements and supports the strategy of provinding information of the preceeding and subsequent track elements. However, more advanced jumps inside the track could be made possible if the strategy were to change. The generated matrix is stored as an vectorized matrix, using `numpy`. In addition, there are a few implementation details worth commenting upon.

First, it is difficult, if not impossible, to determine how well the average quality (e.g. entropy) of the generated samples are during the matrix generation process itself. By this, it becomes difficult to do feature under or over-sampling during the process, if no pre-knowledge about the data is known. Therefore, the matrix is generated as a single executable, isolated operation. Thus, the process could have been implemented with a parallel processing algorithm, but the smarter file pointer operation implementation is used instead. Concretely, the track files are read on demand. The current element and possibly the next element is simultaneously read at any given time.

Secondly, the matrix allocation is done in a single operation. This, because the size of the matrix (the track length) is known before the translation begins. The number of matrix rows equals the number of base-pairs in the selected track range, while the number of matrix columns equals the amount of measures.

Finally, an explanatory track state is used by all feature measures. The track state information does therefore only need to be computed once for every change of position along the explanatory track. Performing a profiling[6] of the MLTrackState implementation revealed that many conditions where calculated multiple of times by dynamically computing the same method calls by all feature measures. By computing (caching) these variables once for every track position, the matrix generation process accieved an imProvement in time performance of 55%.

The prediction

In the «Prediction Tool», each sample of the prediction dataset is predicted independently from the other samples and only based on its feature values. Concretely, a matrix row may be predicted individually, independently of the rest of the matrix rows. By this, the matrix can be broken down into smaller

[6]Improving python performance: `http://wiki.python.org/moin/PythonSpeed/PerformanceTips` (2013-04-29)

pieces (slices) to enable multiple processes to work with them indicidually and in parallel. The technique for performing this task is based on the idea of MapReduce[8]. Since the application is built to run on a large computer cluster, a multi-core approach is implemented. Thus, each worker is given a slice of the shared matrix. Additional required memory is then limited to sum of the paralell matrix sizes and the associated prediction cost of the calculations.

The «mapping» is done dividing specific parts of the sample set into parts of a given size, and delegating it to *k workers*. The workers are implemented as threads, for working on the same (shared) memory. They could have been implemented as (forked) processes, but is not done so due to heavy memory-use with the `multiprocessing`[7] library. Each thread returns or «emits» its predictions back to the prediction matrix of the main process (the reducer). The performance improvement of doing this in parallel increases by the amount of time each prediction takes. This, because the bottleneck is «emitting» the results back to the reducer.

User defined measurements

The machine learning language was created to enable users with little or no programming experience to be able to create measurements on based on their requests. While it is an high level (interpreted) language, the flexibility has risen some concerns in regards to efficiency. To meet this challenge, every MLL measurment is therefore «compiled» before it is used in the translation process. This is done by replacing the macros with implementation specific arguments, and using the `compile` function of `Python`. Then, the track state information is executed (in-memory) using the `exec` method. In addition, the `math` package is only loaded once by the measure object.

Enabling the user to define specific measurements on a case-to-case basis makes the application flexible. Even though some basic feature- and response-measures are provided, the quality of any execution results relies completely on the users understanding of both the application and machine learning in general. The default parameters provided by the application is no guarantee for any meaningful result whatsoever. The applications main goal is to enable a user to explore and experiment with machine learning in a generic way. Thus, a flexible application is naturally not a specificly created application, and does therefore not guaratee any results whatsoever.

The interpretation of user-created code is a potential security risk. The user could add «feature code» that could harm the server installation, delete files, and more. Therefore, the `import` statement is not allowed as MLL code and is removed whenever detected to prevent the import of system libraries.

While it is possible that collecting as many features as possible would result in better models, the memory required for each feature may sets a practical upper limit of how many that can be used. A practical way of detecting what feature that may be more important than others may be detected by analyzing the feature importance as described in 3.2.3 on page 50. However, improving features in this manner may interfear with the error analysis.

[7]The documentation of the `multiprocessing` package of `Python`: `http://docs.python.org/2/library/multiprocessing.html` (2013-04-29)

5.3.2 The struggle of legacy code

The implementation of the application and other various scripts have througout the whole thesis consumed much time and effort. The existing framework of the HyperBrowser is relatively large (thousands of lines of code) and consists of a lot of multi-purpouse functionallity.

One of the greatest implementation challenges has been to understand *how* the framework operates. The understanding of how it works is crucial for understanding how the application could be integrated. Even though it is a massive piece of software, it is poorly documented. Interpreting the code itself, has taken much time and effort because of the lack of documentation. Even though the HyperBrowser development team has been supportive, time-consuming email correspondances have been nessecary to make sense of the framework.

Another great challenge is that the software installation is configured customly so that installation on any other computer is difficult. The production code repository was therefore duplicated[8] in order to separate it from the production code. Access to the user interface was given through url[9]. This resulted a number of challenges, in addition to the time-consuming repository commit's for every code *change*.

A third challenge was the ability to test the created code. For every change, a commit (and a mouse-click action) was required. Thus, the lack of automated tests was a great drawback. In addition, on-server unit-testing was not possible to execute from the local repository when using framework components, since they was configured on the production server specifically.

Application integration

The framework has offered two main generic interfaces which have been used for integrating the machine learning application.

The first interface is `GeneralTool` for creating tools which interacts with the user through a GUI web-interface. It is built to provide options boxes for retrieveing strings, integers, tracks, history elements and more. It has focused on being so generic that some special features does not function in a way that was expected. For instance, it is not possible to set a default value to a track selection option. Also, simple things like changing a single option may influence and reset subsequent selections. Much of the code for creating a tool is focused on handeling events and do type and option checking, which is both time-consuming and feels like a waste of time when more *exciting* code awaits implementation.

The second interface is `Statistic` for creating larger executables that integrates and offers available framework resources. It is implemented for use with a «magic» Factory which instantiates it using a specifically created (and interpreted), string, syntax. To run a statisctic, a subclass of `GeneralTool` may generate such a string containing all parameters needed for the specific execution. It is therefore not possible to instantiate a subclass of `Statistic` if direcly, if one wants to have access to the framework resources. In addition, since

[8]The SubVersion (SVN) repository was branched into a separate folder.

[9]Development area: `http://hyperbrowser.uio.no/ml/` (2013-04-29)

the `Statistic` is a system component, it is not possible to instantiate it locally because of main repository configuration dependencies.

5.3.3 Testing

Unittesting key components (matrix generation, grouping) worked as aspected. Using functionallity on from the SciKit library has helped keeping the amount of application components to test to a minimum, since the library provides their own tests. However, there has been no focus on interaction testing (user testing) and integration- and system-testing (combination testing).

Interface testing could have been a great way of verifying that the aplication is equipped with the nessecary functionallity requested by the actual users. However, the design goal of affording as many (optional) parameters "as possible" is intended to help lowering the bar for the user to start using the application.

Integration- and system-testing of the application as a whole has not been any focus, to favour of unittesting. Integration testing is a trivial, but time-consuming. For instance, the combinations of testing the «Measurement Creator Tool» (including boundary-testing) is shown in figure 5.2. The combination count is:

$$1 \times (\ 1\ ) \times 1 \times (\ 2 \times 5 \times \binom{n}{k} \times 3\ )$$

where there are 3 boundary cases for the undefined value (zero,positive,negative) and k selected transformations out of n available.

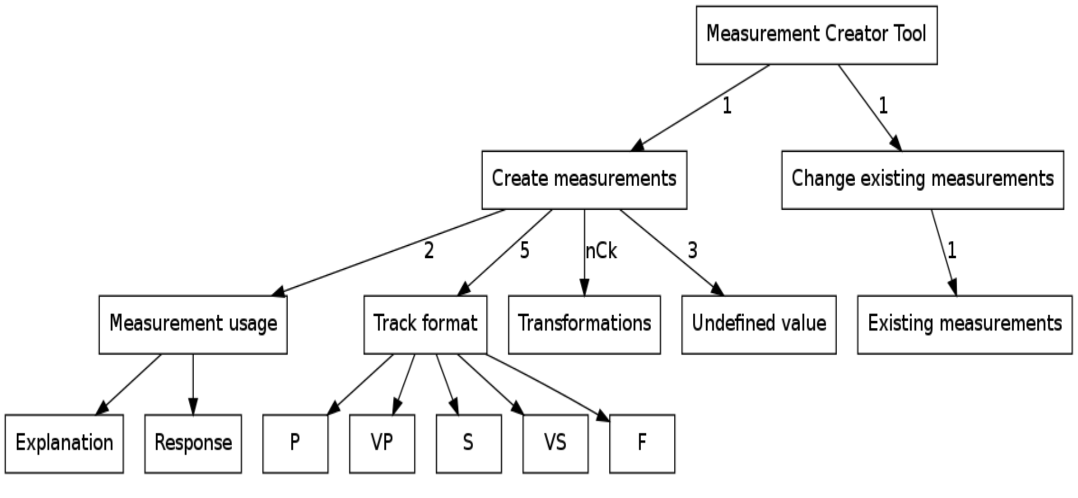


Figure 5.2: The test combinations of the «Measurement Creator Tool».

Chapter 6

Conclusion

A way of translating biological information into a format which is readable for generic machine learning algorithms has been shown. Generic measures and transformations for enhancing the translation process has been suggested.

A machine learning (programming) language (MLL) has been built to provide a flexible way to create and edit measures. Individual information gain and impurity for all measures are computed whenever a model is learned from a dataset. Thus, the measures could be managed in a way that could optimize the performance of the learned model on a case-to-case basis.

An application that incorporates the presented work has been implemented. The proposed strategy for translating and representing biological information is used to learn relationships between 5 selected GTrack formats. A total of 7 supervised machine learning algorithm variants are adopted. In addtion, a framework has been created so that the machine learning algorithms may be generically used for both learning and prediction.

The application is integrated within The Genomic Hyperbrowser, as four separate tools, and is publicly available. The tools aims to enhance the creation of measures, learning models, making model predictions and exporting prediction data. Each tool has a number of mandatory and optional parameter selections. The amount of mandatory parameters are kept to a minimum in order to accommodate users of little or no prior knowledge of machine learning. One of the application design goals has been to provide as many optional configurations options as possible, to make the tool flexible.

An use case, inspired by a biological challenge, is described and used to explore the application. It demonstrates how the basic use of the application may learn a model and then use it for making predictions. Some of the model results are discussed as well as possible further actions to make in order to improve the model. However, the full potensial of the application is yet to be proven by researches that seek to analyze their biological challenges using the computational power of machine learning.

Chapter 7

Future work

The current state of the implemented application has focused its effort on designing and implementing basic structures for making sense of the human genome by using genomic data. The design is one of many and does not claim to be the final work in any ways. The structures as well as the design goals could be changed and probably improved in future work.

The application has now reached a stage where user-testing might provide neccessary feedback to figure out what directions to follow when improving the application. First off, any feedback from concrete use-cases may detect the void of «missing» functionallity. Secondly, an interaction analysis would reveal how the user interface could be configured to improve the user experience.

If there were to be great interest in use of the application, a user-guide could be created to explain the application usage in a user-friendly way. Such a user-guide is a trivial, yet time-consuming, task and is has not been a part of the thesis focus. However, it might become useful in the future.

It could also be intresting to do some structured evaluations of the adapted machine learning algorithms on known generic challenges. Thus, it could be easier for an application user to get started by giving direction towards what algorithms which tend to perform «better» in common situations. This is also applicable to parameter selection. The application is already a flexible applications with many optional (pipeline) steps. An investigation into what parameters which tend to provide «better» results in generic situations may be helpful.

The thesis has focused on adapting supervised machine learning algorithms to learn relationships between genomic data. However, unsupervised learning, data mining and artificial intelligence are branches that are capable of providing new solutions that could be worth exploring.

Bibliography

[1] E. Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning series)*. The MIT Press, 2009. ISBN: 026201243X (cit. on pp. 7, 17, 18).

[2] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. 1st ed. Cambridge University Press, 2009. ISBN: 0521424267 (cit. on p. 24).

[3] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. “SMOTE: Synthetic Minority Over-sampling Technique”. In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357 (cit. on p. 18).

[4] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. Bowyer. “SMOTEBoost: improving prediction of the minority class in boosting”. In: *In Proceedings of the Principles of Knowledge Discovery in Databases, PKDD-2003*. 2003, pp. 107–119 (cit. on p. 18).

[5] F. S. Collins, M. Morgan, and A. Patrinos. “The Human Genome Project: Lessons from Large-Scale Biology”. In: *Science* 300.5617 (2003), pp. 286–290 (cit. on pp. 14, 15).

[6] The International HapMap Consortium. “The International HapMap Project”. In: *Nature* 426 (2003) (cit. on p. 14).

[7] S. A. Cook. “An overview of computational complexity”. In: *Commun. ACM* 26.6 (1983), pp. 400–408 (cit. on p. 24).

[8] J. Dean and S. Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI’04. USENIX Association, 2004, pp. 10–10 (cit. on p. 95).

[9] A. Estabrooks, T. Jo, and N. Japkowicz. “A Multiple Resampling Method for Learning from Imbalanced Data Sets”. In: *Computational Intelligence* 20.1 (2004), pp. 18–36 (cit. on pp. 17, 19).

[10] Nature Publishing Group: E. A. Feingold, P. J. Good, M. S. Guyer, S. Kamholz, L. Liefer, K. Wetterstrand, F. S. Collins, T. R. Gingeras, D. Kampa, E. A. Sekinger, J. Cheng, H. Hirsch, S. Ghosh, Z. Zhu, S. Patel, and A. Piccolboni. “Identification and analysis of functional elements in 1% of the human genome by the ENCODE pilot project”. In: *Nature* 447 (2007) (cit. on p. 14).

[11] The ENCODE Project Consortium: E. A. Feingold, P. J. Good, M. S. Guyer, S. Kamholz, L. Liefer, K. Wetterstrand, F. S. Collins, T. R. Gingeras, D. Kampa, E. A. Sekinger, J. Cheng, H. Hirsch, S. Ghosh, Z. Zhu, S. Patel, and A. Piccolboni. “The ENCODE (ENCyclopedia Of DNA Elements) Project”. In: *Science* 306.5696 (2004), pp. 636–640 (cit. on pp. 14, 47).

[12] R. J. Freund, W. J. Wilson, and P. Sa. *Regression Analysis: Statistical Modeling of a Response Variable*. Academic Press, 2006. ISBN: 0120885972 (cit. on p. 3).

[13] C. Gini. *Variabilità e mutabilità*. 1912 (cit. on p. 50).

[14] Nature Publishing Group. “An integrated encyclopedia of DNA elements in the human genome”. In: *Nature* 489 (2012) (cit. on p. 15).

[15] S. Gundersen, M. Kalaš, O. Abul, A. Frigessi, E. Hovig, and G. Sandve. “Identifying elemental genomic track types and representing them uniformly”. In: *BMC Bioinformatics* 12.1 (2011), p. 494 (cit. on p. 15).

[16] H. Guo and H. L. Viktor. “Learning from imbalanced data sets with boosting and data generation: the DataBoost-IM approach”. In: *SIGKDD Explor. Newsl.* 6.1 (2004), pp. 30–39 (cit. on p. 18).

[17] T. M. Ha and H. Bunke. “Off-line, handwritten numeral recognition by perturbation method”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 19.5 (1997), pp. 535–539 (cit. on p. 18).

[18] M. Hall. “Correlation-based Feature Selection for Machine Learning”. PhD thesis. University of Waikato, 1999 (cit. on pp. 26, 50).

[19] H. He, Y. Bai, E. A. Garcia, and S. Li. “ADASYN: Adaptive synthetic sampling approach for imbalanced learning”. In: *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. 2008, pp. 1322–1328 (cit. on p. 18).

[20] H. He and E. A. Garcia. “Learning from Imbalanced Data”. In: *Knowledge and Data Engineering, IEEE Transactions on* 21.9 (2009), pp. 1263–1284 (cit. on pp. 17, 19).

[21] M. Hilbert and P. López. “The World’s Technological Capacity to Store, Communicate, and Compute Information”. In: *Science* 332.6025 (2011), pp. 60–65 (cit. on p. 7).

[22] D. W. Hosmer and S. Lemeshow. *Applied Logistic Regression (Wiley Series in Probability and Statistics)*. Wiley-Interscience Publication, 2000. ISBN: 0471356328 (cit. on p. 57).

[23] X. Huang, J. Wang, S. Aluru, S. Yang, and L. Hillier. “PCAP: A Whole-Genome Assembly Program”. In: *Genome Research* 13.9 (2003), pp. 2164–2170 (cit. on p. 14).

[24] W. J. Kent, A. S. Zweig, G. Barber, A. S. Hinrichs, and D. Karolchik. “BigWig and BigBed: Enabling Browsing of Large Distributed Datasets”. In: *Bioinformatics* 26.17 (2010), pp. 2204–2207 (cit. on p. 15).

[25] M. Kircher and J. Kelso. “High-throughput DNA sequencing – concepts and limitations.” In: *Journal Article* (2010) (cit. on p. 13).

[26] E. S. Lander. “Initial impact of the sequencing of the human genome”. In: *Nature* (2011) (cit. on p. 13).

[27] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, R. Funke, D. Gage, K. Harris, A. Heaford, and J. Howland. “Initial sequencing and analysis of the human genome”. In: *Nature* (2001) (cit. on pp. 13–15).

[28] P. Larrañaga, B. Calvo, R. Santana, C. Bielza, J. Galdiano, I. Inza, J. A. Lozano, R. Armañanzas, G. Santafé, A. Pérez, and V. Robles. “Machine learning in bioinformatics”. In: *Briefings in Bioinformatics* 7.1 (2006), pp. 86–112 (cit. on pp. 4, 17).

[29] J. Mercer. “On the limits of real variants”. In: *Proc. London Math. Sac.* 2 (1907), pp. 206–224 (cit. on p. 63).

[30] R. J. Nowakowski. *Games of No Chance (Mathematical Sciences Research Institute Publications)*. Cambridge University Press, 1996. ISBN: 0521574110 (cit. on p. 3).

[31] D. Plewczynski, A. Tkacz, L. S. Wyrwicz, A. Godzik, A. Kloczkowski, and L. Rychlewski. "Support-vector-machine classification of linear functional motifs in proteins." In: *Journal of molecular modeling* 12.4 (2006), pp. 453–461 (cit. on pp. 4, 17).

[32] A. R. Quinlan and I. M. Hall. "BEDTools: A Flexible Suite of Utilities for Comparing Genomic Features". In: *Bioinformatics* 26.6 (2010), pp. 841–842 (cit. on p. 15).

[33] S. V. Ramagopalan, A. Heger, A. J. Berlanga, N. J. Maugeri, M. R. Lincoln, A. Burrell, L. Handunnetthi, A. E. Handel, G. Disanto, S. Orton, C. T. Watson, J. M. Morahan, G. Giovannoni, C. P. Ponting, G. C. Ebers, and J. C. Knight. "A ChIP-seq defined genome-wide map of vitamin D receptor binding: Associations with disease and evolution". In: *Genome Research* 20.10 (2010), pp. 1352–1360 (cit. on p. 77).

[34] G. Sandve, S. Gundersen, H. Rydbeck, I. Glad, L. Holden, M. Holden, K. Liestøl, T. Clancy, E. Ferkingstad, M. Johansen, V. Nygaard, E. Tostesen, A. Frigessi, and E. Hovig. "The Genomic HyperBrowser: inferential genomics at the sequence level". In: *Genome Biology* 11.12 (2010), R121 (cit. on pp. 6, 16).

[35] F. Sanger, S. Nicklen, and A. R. Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences of the United States of America* (1977) (cit. on p. 13).

[36] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1971. ISBN: 0252725484 (cit. on p. 50).

[37] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004. ISBN: 0521813972 (cit. on p. 9).

[38] R. S. Thakur, R. Bandopadhyay, B. Chaudhary, and S. Chatterjee. "Now And Next Generation Sequencing Techniques: Future of Sequence Analysis using Cloud Computing". In: *Frontiers in Genetics* 3.280 (2012) (cit. on p. 14).

[39] "Two Modifications of CNN". In: *Systems, Man and Cybernetics, IEEE Transactions on* SMC-6.11 (1976), pp. 769–772 (cit. on p. 19).

[40] L. G. Valiant. "A theory of the Learnable". In: *Communications of the ACM* (1984), pp. 1134–1142 (cit. on pp. 9, 11).

[41] J. L. Weber and E. W. Myers. "Human Whole-Genome Shotgun Sequencing". In: *Genome Research* 7.5 (1997), pp. 401–409 (cit. on p. 14).

[42] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 2011. ISBN: 0123748569 (cit. on pp. 3, 7, 9, 23, 54, 56–59).

Appendix A

Application implementation

This chapter includes references to some of the interfaces. The full source code is available at `http://hyperbrowser.uio.no/dev2/static/downloads/mlcode.zip`.

A.1 API

A.1.1 MLTrackState

MLTrackState.py

```
class MLTrackState(object):
    '''
        A state is a container of position data.

        Author: <fredrhaa> Fredrik Haaland
    '''
    def __init__(self, undefined=9999999):
        '''
            Initialize inner values.
        '''
        self.last = None # The last seen point
        self.data = None # The read from file
        self.xpos = None # The current position along the x'axis
        self.futr = None # The future point
        self.setUndefined( undefined )

        # Speedup for returning size
        # Size is 0, whenever not computed,
        # but contains the correct number otherwise
        self.size_ = 0

        # Speedup for positioning
        self.startPosition_ = None
        self.endPosition_ = None

    def setState(self, data, setPosition=True):
        '''
            Shuffles all positions by one, adding the
            new *point*, while dropping the previous.
            The future point must be set explicitly,
            since it does not always need to be change
        '''
        # Only set last element if a current exists
        if self.size_ > 0:
            self.setLastPoint( self.getCurrentEndPoint() )
        assert len(data) > 2, 'State data input is corrupt: ' % (data)
        self.data = data
        if setPosition:
            # Start from beginning of this state element
            self.setPosition( int(data[1]) )

        # Speed'up ''static'' variables,
        # which needs only to be calculated once.
        self.size_ = self.size()
        self._calculateStartAndEndPositions()

    def equals(self, otherTrackState):
        '''
            Compares current element with the <otherTrackState>.

            Returns True if both tracks have both same position and
                value,
            or False otherwise.

            Notice that True will be returned also when both values are
                 undefined.
        '''
        return self.getCurrentPoint() == otherTrackState.
            getCurrentPoint()

    def setValue(self, value):
        '''
            Assign a value to the current state element.
            Returns True on success, or otherwise False.
        '''
        if len(self) == 0:
            return False
        if len(self.data) >= 4:
            self.data[3] = value
        else:
            newdata = range(0,4)
            for idx,e in enumerate(self.data):
                newdata[idx] = e
            newdata[3] = value
            self.data = newdata
        return True


    def setFuturePoint(self, point):
        '''
            Assign, and override, future point.
            The <em>point</em> must be a 2-tuple.
        '''
        assert point.__class__.__name__ in ['tuple','NoneType'], \
            'The future point must be a 2-tuple (or None)'
        self.futr = point

    def setLastPoint(self, point):
        '''
            Assign, and override, last point.
            The <em>point</em> must be a 2-tuple.
        '''
        assert point.__class__.__name__ in ['tuple','NoneType'], \
            'The last point must be a 2-tuple (or None)'
        self.last = point

    def getLastPoint(self):
        '''
            Get the last point, meaning the previous point
            closest to the current.

            Returns the point as a 2-tuple, or None if not set.
        '''
        return self.last

    def getFuturePoint(self):
        '''
            Get the future point, meaning the next point
            closest to the current.

            Returns the point as a 2-tuple,
            or None if not set or not available.
        '''
        return self.futr

    def getCurrentStartPosition(self):
        '''
            The current (or last known) starting position.

            Returns None, if not set.
        '''
        return self.startPosition_

    def getCurrentEndPosition(self):
        '''
            Returns the position where the current state
            element ends. The position is not a part of the
            element size itself, it is where it ends.

            So, if an element is:

                chr1    10    15

            Then, the length is 5, and the 15th position is not used,
            meaning the end position will be 14, not 15.

            Returns None, if not set.

        '''
        return self.endPosition_

    def setPosition(self, xpos):
        '''
            Set the current position, which may or may not be
            inside the current loaded element.

            A call to this function is performed in setState(),
            to ensure all new states starts reading at the
            correct, starting, position of the element.
        '''
        if self.hasData() and xpos >= 0:
            self.xpos = xpos

    def getCurrentPosition(self):
        '''
            Returns the current position, which may
            or may not be inside the current state element.
            If inside the state element, the offset is used
            to determine the current position.

            Notice: Position is None by default.
        '''
        return self.xpos

    def getOffset(self):
        '''
            Returns the current offset, if a position is set,
            or None otherwise.
        '''
        if self.xpos is None:
            return None
        else:
            return self.getCurrentPosition() - self.
                getCurrentStartPosition()

    def increment(self):
        '''
            Increment internal variables, so more points
            within this element get's read.
        '''
        self.xpos = self.xpos + 1

    def getCurrentEndPoint(self):
        '''
            Returns the end point of the current state element.
        '''
        # Remember old position values
        oldPosition = self.getCurrentPosition()
        # Calculate end-point based on simulated position
        self.setPosition( self.getCurrentEndPosition() )
        endPoint = (self.getCurrentEndPosition(), self.getCurrentValue
            ())
        # Restore old position values
        self.setPosition( oldPosition )
        return endPoint

    def getCurrentPoint(self,offset=0):
        '''
            Returns the current point [2-tuple] (x,y),
            where <em>x</em> is the current position
            and <em>y</em> is the current value.

            If the element is unvalued, it is undefined, and
            the corresponding value set by setUndefined() is used.
        '''
        if self.isSegment() and self.hasValue():    # Valued segment
```

```
            value = self.getCurrentValue()
        elif self.isPoint() and self.hasValue():    # Valued point /
            Function
            value = self.getCurrentValue()
        else:                                       # Unvalued segment
            / point
            value = self.UNDEFINED

        # Return the adjusted index and its calculated value
        return (self.getCurrentPosition(), value)

    def getCurrentValue(self):
        '''
            Returns the current value if the element is valued,
            or otherwise the <em>undefined</em> value is used,
            as set by setUndefined().

            TODO-IMPROVEMENT:
             - Implement label recognition for 'value'
        '''
        if (self.isPoint() or self.isInsideSegment()) and self.hasValue
            ():
            return float(self.data[3])
        else:
            return self.UNDEFINED

    def hasValue(self):
        '''
            Returns True if the current element is valued
            and the current position is inside the element,
            otherwise False.

            TODO-IMPROVEMENT:
             - Implement label recognition for 'value'
        '''
        return (self.isPoint() or self.isInsideSegment()) and \
                 self.hasData() and len(self.data) > 3

    def hasNext(self):
        '''
            Returns True if the current loaded element has not
            yet output'ed _all_ of it's content, meaning there awaits
            another inner point inside the current state element.
        '''
        if self.isInsideSegment():
            return True
        else:
            return self.hasData() and len(self.data) > 0 \
                and self.getCurrentPosition() < self.
                    getCurrentEndPosition()

    def hasData(self):
        '''
            Simple check to verify that data has been set.
            Returns True on success and False otherwise.
        '''
        return self.data is not None

    def hasUndefinedValue(self):
        '''
            Returns True if the current value is undefined,
            else False.
        '''
        return self.getCurrentValue() == self.UNDEFINED

    #==================================================#
    # Helper functions                                 #
    #==================================================#

    def __len__(self):
        '''
            Returns the number computed by size()
            as a short speed-up notation.

            NOTICE: Should only be used when knowing
                    size() is computed.
        '''
        return self.size_

    def size(self):
        '''
            The size of the interval.
            Points will have length of 1,
            while segments will have values larger then 1.
        '''
        if not self.hasData() or len(self.data) < 3:
            return 0
        else:
            return int(self.data[2]) - int(self.data[1])

    def isOutsideSegment(self):
        '''
            Simply returns if the current position is outside a
            segment represented in the source.
        '''
        return self.getLastPoint() is not None \
            and self.getFuturePoint() is not None \
            and not self.isSegment() \
            and self.getLastPoint()[0] < self.getCurrentPoint()[0] \
            and self.getFuturePoint()[0] > self.getCurrentPoint()[0]

    def isInsideSegment(self):
        '''
            Returns True is the current index is
            inside the boundaries of the current element state.
        '''
        if not self.hasData():
            return False
        else:
            return self.isSegment() \
                and self.getCurrentPosition() >= self.
                    getCurrentStartPosition() \
                and self.getCurrentPosition() <= self.
                    getCurrentEndPosition()

    def isAtStartPosition(self):
        '''
            Returns True if state is a segment and the
            current position is at the very beginning of it,
            otherwise False.
        '''
        return self.isSegment() and \
            self.getCurrentPosition() == self.getCurrentStartPosition()

    def isAtEndPosition(self):
        '''
            Returns True if state is a segment and the
            current position is at the very end of it,
            otherwise False.
        '''
        return self.isSegment() and \
            self.getCurrentPosition() == self.getCurrentEndPosition()

    def isSegment(self):
        '''
            Returns True if the current position is part of multiple
            points, meaning it is interpreted as a file in the source.
        '''
        return self.size_ > 1

    def isPoint(self):
        '''
            Returns True if the current position is a point,
            meaning it is interpreted as a file in the source.
        '''
        return self.size_ == 1 \
            and self.getCurrentPosition() == self.
                getCurrentStartPosition()

    def setUndefined(self, value):
        '''
            Set the undefined value,
            used when a point has a undefined value.
        '''
        self.UNDEFINED = value

    def _calculateStartAndEndPositions(self):
        #===================#
        # Start position    #
        #===================#
        if self.size_ == 0:
            self.startPosition_ = None
        # If no last point has been set,
        # use the current start position
        if self.getLastPoint() is None:
            self.startPosition_ = int(self.data[1])
        elif self.getFuturePoint() is None:
            self.startPosition_ = int(self.data[1])
        else:
            ''' The last known value '''
            self.startPosition_ = int(self.data[1])
        #===================#
        # End position      #
        #===================#
        if self.size_ == 0:
            self.endPosition_ = None
        elif len(self.data) > 2:
            self.endPosition_ = int(self.data[2]) - 1

    #==================================================#
    # Other helpful functions, for debugging reasons   #
    #==================================================#

    def __str__(self, *args, **kwargs):
        return "\tlast: %s   \t cr: %s \tn: %s " % ((None,None) if self
            .getLastPoint() is None else self.getLastPoint(),self.
            getCurrentPoint(),self.getFuturePoint())
```

A.1.2 MLMeasure

MLMeasure.py

```
import new
import inspect

from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation
from StringIO import StringIO

class MLMeasure(object):
    '''
        A machine learning measure interface
        facilitated extracting information
        from a given track state (MLTrackState),
        possibly adding one ore more transformations
        (MLTransformation), when called.

        Author: <fredrhaa> Fredrik Haaland
    '''
    def __init__(self, undefined):
        '''
            Initialize undefined value,
            transformation and
            meta data structure.
        '''
        self.transformations = []
        self.transf_metadata = {}
        self.setUndefined( undefined )

    def __call__(self,trackState,n):
        '''
            Enable calling the feature directly,
            applying both feature computation
            and transformation.

            Returns the undefined value if trackState is None
            or n < 1, where MLTrack is not defined.
        '''
        if trackState is None or n < 1:
            return self.getUndefined()
        else:
            self.setMetaData('len', trackState.size())
            self.setMetaData('pos', trackState.getCurrentPosition())
            self.setMetaData('end', trackState.getCurrentEndPosition())
            self.setMetaData('start', trackState.
                getCurrentStartPosition())
            if trackState.isInsideElement():
                self.setMetaData('sit', 'inner')
            else:
                self.setMetaData('sit', 'outer')
            return self.transform( self.getValue( trackState, n ) )

    def getValue(self,trackState,n):
        '''
            Computes the feature value,
            but does not transform the value.

            May optionally set meta-data to be used to
            transform the value on return using
            setMetaData(key,value), where 'key' is the
            corresponding key as specified by
            the MLTransformation API.

            ** To be overridden **
        '''
        raise NotImplementedError()

    def prepare(self):
        '''
            Determine if any transformations are set,
            if not, the optimize the measure for speed.

            Subsequent calls would not do any harm.
        '''
        if len(self.getTransformations()) == 0:

            # The optimized transformation function
            def transform(self,value):
                return value

            # Replace the old, with the optimized
            self.transform = new.instancemethod(
                transform, self, self.__class__ )

    def transform(self, value):
        '''
            The given <em>value</em> is ran
            through a transformation pipeline
            consisting of all the added transformations.

            Returns the 'val' key from <em>value</em>.
            If any transformation layers are added,
            these are used to transform the value,
            otherwise the value is returned as-is.

            [ CONTRACT ]

            The value must be a dictionary valid
            in regards to the MLTransformation API.
        '''
        featureData = self.getMetaData()
        featureData['val'] = value
        for transform in self.getTransformations():
            # Only perform transformation if a
            # *valid* situation is set
            if featureData.has_key('sit') and \
                transform.isValidSituation( featureData.get('sit') ):
                featureData = transform( featureData )
        return featureData['val']

    def setMetaData(self, key, value):
        '''
            Adds meta data, to be used by transformations.
        '''
        self.transf_metadata[ key ] = value

    def getMetaData(self):
        '''
            Returns the meta-data supposedly used
            with transformations.
        '''
        return self.transf_metadata

    def addTransformation(self, transformation):
        '''
            Add a <em>transformation</em> layer,
            to transform the computed feature value.

            The <em>transformation</em> must be an
            instantiated sub-class of MLTransformation.

            The layers are stacked in a
            first-come-first-served manner.
        '''
        assert isinstance( transformation, MLTransformation ), \
            'The added transformation is not instantiated'
        self.transformations.append( transformation )

    def getTransformations(self):
        '''
            Returns the list of added transformations.
        '''
        return self.transformations

    def getClassName(self):
        '''
            Returns the class name.
        '''
        return self.__class__.__name__

    def getUndefined(self):
        '''
            Returns the undefined value.
        '''
        return self.undefined

    def setUndefined(self, value):
        '''
            Set the undefined value, used when a point
            has a undefined value.

            Also, make it available for the transformations.
        '''
        self.undefined = value
        self.setMetaData('undefined', value)

    def __repr__(self):
        '''
            Returns the MLL representation of the feature.
        '''
        handle = StringIO()
        handle.write("\nFEATURE BEGIN # {} #\n".format(self.__class__.
            __name__))
        # Detect configurations
        if len( self.getMetaData() ) > 0:
            handle.write("    CONFIG BEGIN\n")
            for key, value in self.getMetaData().items():
                key = key.upper()
                value = value
                handle.write("        {} = {}\n".format(key,value))
        # Possibly add transformations
        if len( self.getTransformations() ) > 0:
            handle.write("        TRANSFORMATIONS BEGIN")
            for transformation in self.getTransformations():
                for line in repr(transformation).split('\n'):
                    handle.write("            {}\n".format(line))
            handle.write("        TRANSFORMATIONS END\n")
        # Add attributes as configuration parameters
        if len( self.getMetaData() ) > 0:
            handle.write("    CONFIG END\n")
        # Add measurement code as MLL code
        handle.write("    BODY BEGIN\n")
        sourceCodeLines = inspect.getsourcelines( self.getValue )
            [0][1:]
        for line in sourceCodeLines:
            if line.strip().startswith('#'):
                continue
            # Swap macros and more
            line = line.replace('return', 'OUTPUT')
            line = line.replace('elif', 'else if')
            line = line.replace('self.getUndefined()', 'UNDEFINED')
            line = line.replace('trackState.getCurrentPosition()', '
                POSITION')
            line = line.replace('trackState.getCurrentValue()', 'VALUE'
                )
            line = line.replace('trackState', 'STATE')
            line = line.replace('.isAtStartPosition()', '.
                IS_AT_START_POSITION')
            line = line.replace('.isAtEndPosition()', '.
                IS_AT_END_POSITION')
            line = line.replace('.getCurrentStartPosition()', '.
                START_POSITION')
            line = line.replace('.getCurrentEndPosition()', '.
                END_POSITION')
            line = line.replace('.getFuturePoint()[0]', '.
                FUTURE_POSITION')
            line = line.replace('.getFuturePoint()[1]', '.FUTURE_VALUE'
                )
            line = line.replace('.getFuturePoint()', '.FUTURE_POINT')
            line = line.replace('.getLastPoint()[0]', '.LAST_POSITION')
            line = line.replace('.getLastPoint()[1]', '.LAST_VALUE')
            line = line.replace('.getLastPoint()', '.LAST_POINT')
```

```
                line = line.replace('.isInsideSegment()', '.
                    IS_INSIDE_SEGMENT')
                line = line.replace('.isOutsideSegment()', '.
                    IS_OUTSIDE_SEGMENT')
                line = line.replace('.isSegment()', '.IS_SEGMENT')
                line = line.replace('.isPoint()', '.IS_POINT')
                line = line.replace('.getOffset()', '.OFFSET')
                line = line.replace('.size()', '.SIZE')
                line = line.replace('float(n)', 'SIZE')
                line = line.upper()
                # Trim off possible post line comments
                indexOfComment = -1
                lineParts = line.split(' ')
                for i, part in enumerate(lineParts):
                    if part.lstrip().startswith('#'):
                        indexOfComment = i
                if indexOfComment >= 0:
                    line = "{}\n".format(' '.join( lineParts[0:
                        indexOfComment] ))
                # Write MLL code line
                handle.write("{}".format( line ))
            handle.write("    BODY END\n")
            handle.write("FEATURE END")
            handle.seek(0)
            content = handle.read()
            handle.close()
            return content
```

A.1.3 MLFeature

MLFeature.py

```
from sys import maxint
from quick.ml.api.feature.MLMeasure import MLMeasure

class MLFeature(MLMeasure):
    '''
        A machine learning feature
        extracts information from given track states.

        Is also capable of transforming the computed
        feature value using transformation layers
        implementing the MLTransformation API.

        Author: <fredrhaa> Fredrik Haaland
    '''
    def __init__(self, undefined=maxint):
        '''
            Initializes parent with maximum integer
            as the undefined value as default,
            but may be overridden.
        '''
        MLMeasure.__init__(self, undefined)
```

A.1.4 MLResponse

MLResponse.py

```
from sys import maxint
from quick.ml.api.feature.MLMeasure import MLMeasure

class MLResponse(MLMeasure):
    '''
        A machine learning response defines labels (classes)
        based on the information from given track states.

        Is also capable of transforming the labels
        using transformation layers implementing
        the MLTransformation API.

        Author: <fredrhaa> Fredrik Haaland
    '''
    def __init__(self, undefined=maxint):
        '''
            Initializes parent with maximum integer
            as the undefined value as default,
            but may be overridden.
        '''
        MLMeasure.__init__(self, undefined)
```

A.1.5 MLTransformation

MLTransformation.py

```
import inspect

from sys import maxint
from StringIO import StringIO

class MLTransformation(object):
    '''
        A machine learning feature transformation interface.

        Author: <fredrhaa> Fredrik Haaland
    '''
    def __init__(self, situation='all', undefined=maxint):
        '''
            Initialize the <em>situation</em>
            the transformation do apply.

            Allowed situations are:

                - all        (Default, meaning rest of the list)
                - inner
                - outer

            These reflect the ones returned in getSituations().
        '''
        self.setSituation( situation )
        self.setUndefined( undefined )

    def __call__(self, featureData):
        '''
            Enable calling the filter directly,
            as a synonym for getTransformedValue(),
            for readability.
        '''
        return self.getTransformedValue( featureData )

    def getTransformedValue(self, featureData):
        '''
            To be overridden by the inherited class.

            featureData is a dictionary, and has some reserved keys:

                - featureData['val'] : Contains the feature value
                - featureData['pos'] : Contains the current position
                - featureData['len'] : Contains the element's length (
                    size)
                - featureData['start'] : Contains the element's start
                    position
                - featureData['end'] : Contains the element's end
                    position
                - featureData['sit'] : Contains the element's situation

            Other keys may be used freely, if building own
                MLTransformation's.

            ** To be overridden **
        '''
        return featureData

    def isValidSituation(self,situation):
        '''
            Returns true if the situation is valid
            according to this configuration,
            otherwise False.
        '''
        return self.getSituation() in [ 'all', situation ]

    def getSituations(self):
        '''
            Returns the list of allowed situations.
        '''
        return ['all','inner','outer']

    def getSituation(self):
        '''
            Returns the currently set situation.
        '''
        return self.situation

    def setSituation(self, situation):
        '''
            Set the <em>situation</em> where
            *this* (self) transformation is applied.
        '''
        self.situation = situation


    def setUndefined(self, undefined):
        self.undefined = undefined

    def getUndefined(self):
        return self.undefined

    def __repr__(self):
        '''
            Returns the MLL representation of the feature.
        '''
        handle = StringIO()
        handle.write("\nTRANSFORMATION BEGIN\n")

        # Detect configurations
        if len( self.__dict__ ) > 0:
            handle.write("    CONFIG BEGIN\n")
            for key, value in self.__dict__.items():
                key = key.upper()
                value = value
                if type( value ).__name__ == 'str':
                    handle.write("        {} = '{}'\n".format(key,value
                        .upper()))
                else:
                    handle.write("        {} = {}\n".format(key,value))

        # Detect configurations
        if len( self.__dict__ ) > 0:
            handle.write("    CONFIG END\n")

        handle.write("    BODY BEGIN\n")
        sourceCodeLines = inspect.getsourcelines( self.
            getTransformedValue )[0][1:]
        for line in sourceCodeLines:
            if line.strip().startswith('#'):
                continue
            # Swap macros and more
            line = line.replace('return featureData', 'OUTPUT VALUE')
            line = line.replace('elif', 'else if')
            line = line.replace('self.getUndefined()', 'UNDEFINED')
            line = line.replace("featureData['val']", 'VALUE')
            line = line.replace("featureData['pos']", 'POSITION')
            line = line.replace("featureData['len']", 'LENGTH')
            line = line.replace("featureData['sit']", 'SITUATION')
            line = line.replace("featureData['start']", 'START')
            line = line.replace("featureData['end']", 'END')
            line = line.replace('featureData.has_key', 'HAS_CONFIG')
            line = line.replace('featureData.get', 'GET_CONFIG')
            line = line.upper()
            # Trim off possible post line comments
            indexOfComment = -1
            lineParts = line.split(' ')
            for i, part in enumerate(lineParts):
                if part.lstrip().startswith('#'):
                    indexOfComment = i
            if indexOfComment >= 0:
                line = "{}\n".format(' '.join( lineParts[0:
                    indexOfComment] ))
            # Write MLL code line
            handle.write("{}".format( line ))
        handle.write("    BODY END\n")
        handle.write("TRANSFORMATION END")
        handle.seek(0)
        content = handle.read()
        handle.close()
        return content
```

A.1.6 MLAlgorithm

Available at `http://TODO.soon`

A.2 Implementations

A.2.1 Algorithms

MLAnomalyDetection

MLAnomalyDetection.py

```
import numpy
from math import pi,sqrt

from getpass import getuser
if getuser() not in ['haaland','fredrik']:
    from quick.util.StaticFile import StaticFile
else:
    import matplotlib.pyplot as plt
    import matplotlib.mlab as mlab

from quick.ml.implementation.factory.MachineLearningFactory import
    MachineLearningFactory
from quick.ml.api.algorithm.MLAlgorithm import MLAlgorithm

class MLAnomalyDetection(MLAlgorithm):
    '''
        Binary Semi-supervised Learning Algorithm.
    '''
    def __init__(self):
        MLAlgorithm.__init__(self)

    # Override
    def arangeTest(self, Xtest, hasLabels=True):
        '''
            Using default built-in method.
            @see: arangeTestDefault()
        '''
        return self.arangeTestDefault( Xtest, hasLabels )

    # Override
    def arangeAll(self, Xtrain, cvfactor=0.0):
        '''
            Copying all class=1 into <em>Xcv</em>,
            leaving all class=0 in <em>Xtrain</em>.

            The cross validation factor (cvfactor) is not used.
        '''
        Xtrain_ = MachineLearningFactory.getSamples( Xtrain,
            lastColumnClass = 0 )
        Xcv_ = MachineLearningFactory.removeSamples( Xtrain,
            lastColumnClass = 0 )
        Xtrain, Ytrain = MachineLearningFactory.splitXandY( Xtrain_ )
        Xcv, Ycv = MachineLearningFactory.splitXandY( Xcv_ )
        self.saveSamplesToDiskAndAppendToResults( Xtrain, Xcv )
        return Xtrain, Ytrain, Xcv, Ycv

    def normalize(self, X):
        '''
            Performs feature scaling to make the
            minimization algorithm converge
            more easily.

            Normalizes only contents, and not classes.
            @see: MachineLearningAlgorithmFactory
        '''
        return MachineLearningFactory.normalize( X )

    # Override
    def learn(self, Xtrain, Ytrain, Xcv, Ycv, minimum=1e-05, lmbda=0,
              learningCurveIntervals=None, thresholdLevel=1000.0, **
                  kwArgs):
        '''
            Not using <em>Ytrain</em> for learning,
            only for cross validating in order
            to find the threshold (Epsilon) .

            Not taking <em>minimum</em> nor
            <em>lambda</em> into account.
        '''
        # Compute mean
        mu = numpy.mean( Xtrain, 0 )

        # Compute variance, ( 0 degrees of freedom )
        # and replace zero with close to zero value
        # to work-around divide by zero
        sigma2 = numpy.var( Xtrain, 0, ddof=0 )
        sigma2 = numpy.add( sigma2, (sigma2 == 0) * 1e-10 )

        # Compute values for cross-validation set to detect epsilon
        vProducts = self.getProducts( Xcv, mu, sigma2 )

        # Find a proper threshold ( detail level of 1/1000 )
        stepSize = (numpy.max(vProducts) - numpy.min(vProducts)) /
            thresholdLevel
        best_epsilon = 0
        best_fscore = 0

        # Iterate over epsilons
        epsilon = stepSize
        while epsilon < numpy.max( vProducts ):

            # Make prediction
            prediction = numpy.matrix( (vProducts < epsilon) * 1.0 ).T

            # Compute precision and recall
            P, R, A, S = MachineLearningFactory.computeMetrics( Ycv,
                prediction, None )

            # Compute F-score
            fscore = MachineLearningFactory.getFscore( P, R )
            # Store if better
            if fscore > best_fscore:
                best_fscore = fscore
                best_epsilon = epsilon

            # Increment epsilon to
            epsilon = epsilon + stepSize

        # Mandatory labels
        classes = ['0','1']
        self.addLearningResult( 'Classes', classes )
        self.addLearningResult( 'Classifier', self.saveToDisk( numpy.
            vstack((mu,sigma2)) ) )
        self.addLearningResult( 'Extra parameters', {
            'Epsilon' :  best_epsilon,
            'Def. Class 0' : 'Not anomaly',
            'Def. Class 1' : 'Anomaly'
        })

        # Possibly generate learning curve ( but has no test-score, so
            only one graph )
        if learningCurveIntervals is not None and Ytrain is not None
            and Xcv is not None and Ycv is not None:
            intervals = min( learningCurveIntervals,
                MachineLearningFactory.getSampleSize( Xtrain ) )
            # Adjusted, manual KFold ( taken from scikit-learn.org )
            X_folds = numpy.array_split( Xtrain, intervals )
            Y_folds = numpy.array_split( Ytrain, intervals )
            X_train = None
            Y_train = None
            for k in range( intervals ):
                # Augment (Concatenate) data-set along the way
                if X_train is None:
                    X_train = X_folds[ k ]
                    Y_train = Y_folds[ k ]
                else:
                    X_train = numpy.vstack( ( X_train, X_folds[ k ] ) )
                    Y_train = numpy.vstack( ( Y_train, Y_folds[ k ] ) )
                # Call upon self ( but not recursively ) to apply
                # both learning and testing in order to find accuracy
                trainingRes = self.learn( X_train, Y_train, Xcv, Ycv,
                    minimum, lmbda,
                                learningCurveIntervals=None,
                                    thresholdLevel=100.0 )
                extra = trainingRes.get('Extra parameters').copy()
                extra['Skip graph generation'] = True
                classifier = MachineLearningFactory.loadFromDisk(
                    trainingRes.get('Classifier') )
                trainingRes = self.test( X_train, Y_train, classifier,
                    classes, extra )
                crossValRes = self.test( Xcv, Ycv, classifier, classes,
                     extra )
                self.addLearningCurveIntervalStep(
                    (
                        ( 1.0 - float( trainingRes.get('Accuracy') ) ),
                        ( 1.0 - float( crossValRes.get('Accuracy') ) )
                    )
                )
        return self.getLearningResults()

    # Override
    def test(self, Xtest, Ytest, classifier, classes, extra={}, graphs=
        False, **kwArgs):
        '''
            Predict the <em>classes</em> of <em>Xtest</em>
            measuring performance with <em>Ytest</em>
            using the <em>classifier</em>, and possibly some
            <em>extra</em> dictionary parameters.

            If threshold is 0, all possibilities are 100%.
        '''
        # Re-construct classifier
        # Hypothesis is a single matrix row (vector)
        # where first half is mu and second half is sigma2.
        classifier = self.loadClassifier( classifier )
        mu = classifier[0]
        sigma2 = classifier[1]

        # Make predictions
        threshold = float( extra.get('Epsilon') )
        Xpredict = self.getProducts( Xtest, mu, sigma2 )
        Ypredict = numpy.matrix( numpy.asarray( ( Xpredict < threshold
            ) * 1.0 ) ).T

        # Make probability container by 'clamping' the values
        # at threshold or higher as max values ( 100% probability )
        Yprobabilities = numpy.add(
            numpy.asarray( Xpredict >= threshold ) * threshold,
            numpy.multiply( Ypredict, Xpredict )
        )

        # Then resize the area from reaching from smallest value to
            threshold [0,1]
        # But if threshold is 0, all values are regarded as 100% likely
             to be anomalies
        if threshold == 0:
            Yprobabilities = numpy.ones( numpy.shape( Yprobabilities )
                )
        else:
            Yprobabilities = numpy.divide( Yprobabilities, threshold )

        # Create inverted probabilities for second anomaly class
        Yprobabilities = numpy.hstack( (Yprobabilities, 1-
            Yprobabilities ) )

        # Add metrics and prediction tracks
        self.addBasicTestMetricResults( Ytest, Ypredict, Yprobabilities
            , classes, threshold )

        # With regards to this being used for generating learning curve
            ,
        # the other graphs are not needed to be generated
        if extra.get('Skip graph generation') is None:

            # Calculate only the values of the classes with a predicted
                 hit
            onlyHits = numpy.multiply( Xpredict, (( Ypredict * 1 ) ==
                Ytest ) )
            onlyHits = MachineLearningFactory.removeZeroSamples(
                onlyHits )
            onlyHits = numpy.prod( onlyHits, axis=1 ) # Verify
            onlyHits = [ float(onlyHits[i]) for i in xrange( numpy.size
                ( onlyHits, 0 ) ) ]
            allValues = [ float(Xpredict[i]) for i in xrange( numpy.
                size( Xpredict, 0 ) ) ]
            thresholdLine = [ threshold for i in range( len(allValues)
                ) ]
            predictedClasses = [ int(Ypredict[i]) for i in xrange( len(
                allValues) ) ]
            if Ytest is not None:
                originalClasses = [ int(Ytest[i]) for i in xrange( len(
                    allValues) ) ]

            self.addPredictionTracks( allValues, numpy.matrix(
                predictedClasses), Yprobabilities, classes, extra )

            if graphs and Ytest is not None:
                # Visualize
                self.addHistogramGeneralValuesRPlot( allValues )
                self.addHistogramHitsOnlyValuesRPLot( onlyHits )
                self.addSpanGraphRPlot( thresholdLine, allValues )
                self.addInspectionGraphRPLot( thresholdLine, allValues,
                     predictedClasses, originalClasses )
                self.addProbabilityGraphForClasses(
                    classes,
                    numpy.hstack( ( Yprobabilities, Yprobabilities ) ),
                    threshold = 1.0
                )

        # Count outlier's
        self.addTestResult( 'Anomalies detected', numpy.sum( Ypredict )
             )

        return self.getTestResults()

    #===========================================#
    #    HELPER METHODS                         #
    #===========================================#

    def getProducts(self, X, mu, sigma2 ):
        '''
            Returns computes product of Gaussian curve
            predictions applied along feature axis.
        '''
        # Compute once for speed-up
        sigma = numpy.sqrt( sigma2 )
        # Vector-iced version
        a = numpy.divide( numpy.exp( -0.5 * numpy.power( numpy.divide(
                numpy.subtract(X, mu), sigma ), 2 ) ), sigma * sqrt( 2
                    * pi ) )
        # In case any of the predictions are zero
        # they will be replaced with 1.0 which
        # will not affect the predictions (ignore it)
        # If not, all predictions would be 0, which gives no meaning.
        a = a + ( a == 0 )
        return numpy.prod( a, 1 )

    #===========================================#
    #    VISUALIZATION                          #
    #===========================================#

    def vizualize(self,*args):
        '''
            Visualize the model.
        '''
        if getuser() in ['haaland','fredrik']:
            # Fetch arguments
            mu = args[0]
            sigma2 = args[1]
            XYtest = args[2]
            epsilon = args[3]

            # Fetch other useful variables
            yvalue = 0.5
            normalIcon = 'o'
            anomalIcon = 'x'
            graphCount = MachineLearningFactory.getFeatureSize( XYtest
                ) - 1
            Xnormal = MachineLearningFactory.getSamples( XYtest,
                lastColumnClass = 0 )
            Xanomal = MachineLearningFactory.getSamples( XYtest,
                lastColumnClass = 1 )
            sampleCountNormal = MachineLearningFactory.getSampleSize(
                Xnormal )
            sampleCountAnomal = MachineLearningFactory.getSampleSize(
                Xanomal )

            # Create accessible figure
            fig = plt.figure()

            a = [0,5,7]

            if graphCount == -1: # Special case, TODO : Build
                Multivariate graph
                pass

            else: # Multi dimensional

                normalProducts = [ 1.0 for i in range(sampleCountNormal
                    ) ]
                anomalProducts = [ 1.0 for i in range(sampleCountAnomal
                    ) ]
                for indx in xrange( graphCount ):
                    # Normal curve plot
                    ax = fig.add_subplot( graphCount, 2, 2+(2*indx)-1 )
                    mean = mu[ 0, indx ]
                    sigma = sqrt( sigma2[ 0, indx ] )
                    x = numpy.linspace(mean-(sigma*3.5),mean+(sigma
                        *3.5),150)
                    ax.plot( x, mlab.normpdf(x,mean,sigma), linewidth=1
                        )
                    ax.set_ylabel( "[%d]" % (a[indx]), fontsize=10 )
                    yMax = 0
                    yMin = 0
                    for i in xrange( sampleCountNormal ):
                        xval = float(Xnormal[i,indx])
                        yval = mlab.normpdf(xval,mean,sigma)
                        if yval > 0:
                            normalProducts[i] = normalProducts[i] *
                                yval
                        if yval < yMin:
                            yMin = yval
                        if yval > yMax:
                            yMax = yval
                        ax.plot( xval, yval, color='white', marker=
                            normalIcon )
                    for i in xrange( sampleCountAnomal ):
                        xval = float(Xanomal[i,indx])
                        yval = mlab.normpdf(xval,mean,sigma)
                        if yval > 0:
                            anomalProducts[i] = anomalProducts[i] *
                                yval
                        if yval < yMin:
                            yMin = yval
                        if yval > yMax:
                            yMax = yval
                        ax.plot( xval, yval, color='black', marker=
                            anomalIcon )
                    ax.set_xlim([mean-(sigma*3.5),mean+(sigma*3.5)])
                    ax.set_ylim([yMin,max(yMax*1.1, mlab.normpdf(mean,
                        mean,sigma))])
                    ax.set_yticks([])
                    ax.set_xticks([])
                    # Anomaly plot
                    ax = fig.add_subplot( graphCount, 2, 2+(2*indx) )
                    for i in xrange(len(normalProducts)):
                        ax.plot( normalProducts[i], yvalue, color='
                            white', marker=normalIcon )
                    for i in xrange(len(anomalProducts)):
                        ax.plot( anomalProducts[i], yvalue, color='
                            black', marker=anomalIcon )
                    # Anomaly boundary
                    ax.plot( epsilon, yvalue, color='black', marker='|'
                         )
                    #ax.set_ylabel( "[%d]" % (indx), fontsize=10 )
                    ax.set_ylabel( "[%d]" % (a[indx]), fontsize=10 )
                    #ax.set_xlim([ 1, (epsilon*(1+graphCount-indx)) ])
                    ax.set_ylim([0,1])
                    ax.set_yticks([])
                    ax.set_xticks([])
                plt.show()
        else:
            # TODO : Make rpy plot
            pass
```

MLArtificialNeuralNetwork

MLArtificialNeuralNetwork.py

```

import numpy

from random import randint
from getpass import getuser
if getuser() not in ['haaland','fredrik']:
    from quick.util.StaticFile import StaticFile
else:
    import matplotlib.pyplot as plt
    from matplotlib.pyplot import cm

from scipy.optimize import fmin_cg
from quick.ml.implementation.factory.MachineLearningFactory import
    MachineLearningFactory
from quick.ml.api.algorithm.MLAlgorithm import MLAlgorithm

class MLArtificialNeuralNetwork(MLAlgorithm):
    '''
        Supervised (multi-class) learning algorithm.

        It is used 1 hidden layer, having twice the size
        as the feature size by default.
    '''
    def __init__(self):
        MLAlgorithm.__init__(self)
        # Arguments used in inner-loop
        # for generating learning curve
        self.setLearningCurveArguments( None )
        self.setDoublePrecision( 3 )

    # Override
    def arangeTest(self, Xtest, hasLabels=True):
        '''
            Returns the <em>Xtest</em> and its labels <em>Ytest</em>.
        '''
        return self.arangeTestDefault( Xtest, hasLabels )

    # Override
    def arangeAll(self, Xtrain, cvfactor=0.2):
        '''
            Using default built-in method for
            supervised learning algorithms.
            @see: arangeAllMultiClassSupervised()
        '''
        return self.arangeAllMultiClassSupervised( Xtrain, cvfactor )

    # Override
    def learn(self, Xtrain, Ytrain, Xcv, Ycv, minimum=1e-05, lmbda=0,
        learningCurveIntervals=None, layerFactor=2.0, **kwArgs ):
        '''
            Will in later implementation make use of
            <em>learningCurveIntervals</em>.
        '''
        # Parse optional argument(s)
        if kwArgs.has_key('maxiter'):
            maxiter = kwArgs.get('maxiter')
        else:
            maxiter = 200

        labels = MachineLearningFactory.countDistinctClasses( Ytrain,
            True ).keys()
        num_labels = len( labels )
```

```
input_layer_size  = MachineLearningFactory.getFeatureSize(
    Xtrain )
hidden_layer_size = int( MachineLearningFactory.getFeatureSize(
     Xtrain ) * layerFactor )

# Replace classes with numerical identifiers
for i in range( MachineLearningFactory.getSampleSize( Ytrain )
    ):
    Ytrain[i] = labels.index( Ytrain[i] )

initial_Theta1 = MachineLearningFactory.randInitializeWeights(
    hidden_layer_size,
    input_layer_size,
    includeThetaZero=True
)

initial_Theta2 = MachineLearningFactory.randInitializeWeights(
    num_labels,
    hidden_layer_size,
    includeThetaZero=True
)

# Unroll parameters
initial_theta = numpy.hstack(
    ( numpy.ravel( initial_Theta1 ), numpy.ravel(
        initial_Theta2 ) )
)

# Possibly generate learning curve
if learningCurveIntervals is not None and Xcv is not None and
    Ycv is not None:
    self.setLearningCurveArguments(
        [ Xtrain, Ytrain, Xcv, Ycv, input_layer_size,
          hidden_layer_size,
         num_labels, labels, learningCurveIntervals ]
    )

# Initialize minimization
xopt, fopt, func_calls, grad_calls, warnflag, allvec = fmin_cg(
    f = self._cost,
    x0 = initial_theta,
    fprime = self._grad,
    args = ( Xtrain, Ytrain, lmbda, input_layer_size,
        hidden_layer_size,
             num_labels, learningCurveIntervals, Xcv, Ycv ),
    callback = self._curve,
    maxiter = maxiter,
    full_output = True,
    disp = False,
    retall = True
)

# Mandatory labels
self.addLearningResult( 'Classes', labels )
self.addLearningResult( 'Classifier', self.saveToDisk( xopt ) )

# Store extra parameters for making predictions in test()
self.addLearningResult( 'Extra parameters', {
    'Input layer size': input_layer_size,
            'Hidden layer size': hidden_layer_size,
            'Label count': num_labels
        })

        # Store learning results
        self.addLearningResult( 'Minimization function calls',
            func_calls )
        self.addLearningResult( 'Minimization gradient calls',
            grad_calls )
        self.addLearningResult( 'Minimization calls', func_calls )
        self.addLearningResult( 'Minimized cost', fopt )
        if warnflag == 0:
            self.addLearningResult( 'Learning status', 'Successfully
                converged' )
        elif warnflag == 1:
            self.addLearningResult( 'Learning status', 'Too many
                function evaluations or too many iterations' )
        else:
            self.addLearningResult( 'Learning status', 'Stopped
                unexpectedly' )

        return self.getLearningResults()

    # Override
    def test(self, Xtest, Ytest, classifier, classes, extra={}, graphs=
        False, **kwArgs):
        '''
            Predict the <em>classes</em> of <em>Xtest</em>
            measuring performance with <em>Ytest</em>
            using the <em>classifier</em>, and possibly some
            <em>extra</em> dictionary parameters.
        '''
        # Re-construct classifier
        hypothesis = self.loadClassifier( classifier )
        input_layer_size = extra.get('Input layer size')
        hidden_layer_size = extra.get('Hidden layer size')
        num_labels = extra.get('Label count')

        # Make sure classes are of correct type (floats)
        for i in range( len( classes ) ):
            classes[ i ] = float( classes[ i ] )

        # Unroll parameters
        Theta1 = numpy.reshape(
            hypothesis[0:((input_layer_size+1) * (hidden_layer_size))],
            ((input_layer_size+1),(hidden_layer_size))
        ).T

        Theta2 = numpy.reshape(
            hypothesis[((input_layer_size+1) * (hidden_layer_size)):],
            ( (num_labels), (hidden_layer_size+1) )
        )

        # Activation function
        sigmoid = MachineLearningFactory.sigmoid

        # Useful value
        m = numpy.size( Xtest, 0)
```

```
h1 = sigmoid( numpy.matrix( numpy.hstack( (numpy.ones( (m, 1) )
    , Xtest) ), copy=False ) * Theta1.T )
h2 = sigmoid( numpy.matrix( numpy.hstack( (numpy.ones( (m, 1) )
    , h1) ), copy=False ) * Theta2.T )

# Adjust prediction values to probabilities
Yprobabilities = numpy.copy( h2 )
for i in range( MachineLearningFactory.getSampleSize( h2 ) ):
    Yprobabilities[i,:] = numpy.divide( h2[i,:], numpy.sum( h2[
        i,:] ) )

# Make predictions
Ypredict = numpy.matrix( numpy.argmax( Yprobabilities, axis=1 )
    ).T

# .. and map position back to its *class* value identifier
Yprobability_best = list()
for i in range( MachineLearningFactory.getSampleSize( Ypredict
    ) ):
    Ypredict[i] = classes[ Ypredict[i] ]
    Yprobability_best.append(
        # Fetch the element in the array of highest value and
            divide it by total sum
        float( Yprobabilities[ i, int( numpy.argmax(
            Yprobabilities[i] ) ) ] )
    )

# Add metrics and prediction tracks
if Ytest is not None:
    # Calculate only the values of the classes with a predicted
         hit
    onlyHits = numpy.argmax( numpy.multiply( Yprobabilities, (
        Ypredict.T == Ytest )[:,0] ), axis=1 )
    onlyHits = MachineLearningFactory.removeZeroSamples(
        onlyHits )
    onlyHits = [ float(onlyHits[i]) for i in range( numpy.size(
         onlyHits, 0 ) ) ]
    YroundedOriginal = numpy.around( Ytest*100, self.
        getDoublePrecision() )
    YroundedPredicted = numpy.around( numpy.matrix( numpy.
        asarray( Ypredict ) *100 ), self.getDoublePrecision() )
    self.addBasicTestMetricResults(
        YroundedOriginal,
        YroundedPredicted,
        Yprobabilities,
        classes,
        threshold = None
    )

# Add metrics and prediction tracks
self.addPredictionTracks(
    Yprobability_best,
    Ypredict.ravel().tolist()[0],
    Yprobabilities,
    classes,
    extra
)
```

```
        # With regards to this being used for generating learning curve
            ,
        # the other graphs are not needed to be generated
        if extra.get('Skip graph generation') is None and graphs:
#             self.addProbabilityGraphForClasses( [ int(label) for label
     in classes ], Yprobabilities )
#             self.addHistogramHitsOnlyValuesRPlot( onlyHits )
#             self.addHistogramGeneralValuesRPlot(
#                 [ float(Ypredict[i]) for i in range( numpy.size(
    Ypredict, 0 ) ) ]
#             )
            # Visualize
            self.vizualize( Xtest, len(classes), h1, h2 )

        return self.getTestResults()

    #===========================================#
    #    HELPER METHODS                         #
    #===========================================#

    def getLearningCurveArguments(self):
        return self.learningCurveArguments

    def setLearningCurveArguments(self, args):
        self.learningCurveArguments = args

    def _cost(self, theta, *args):
        self._compute( theta, *args )
        return self.cost

    def _grad(self, theta, *args):
        # Normal grad
        self._compute( theta, *args )
        # Gradient check
        #epsilon = 0.001
        #self._compute( numpy.subtract(theta, epsilon), *args )
        #grad_s = self.grad
        #self._compute( numpy.add(theta, epsilon), *args )
        #grad_a = self.grad
        #print "f() = ", numpy.sum( numpy.subtract( grad_a, grad_s ).
            ravel() ) / 2*epsilon
        return self.grad

    def _curve(self, theta):
        '''
            A callback for computing learning curve,
            inside minimization algorithm.
        '''
        args = self.getLearningCurveArguments()
        if args is not None:
            # While not knowing total amount of calls
            # the points are added by chance, which by the
            # law of large numbers should be OK
            intervals = args[8]
            if randint(0,intervals) == 0:
                Xtrain = args[0]
                Ytrain = args[1]
                Xcv = args[2]
                Ycv = args[3]
                extra = {
                    'Input layer size': args[4],
                    'Hidden layer size': args[5],
                    'Label count': args[6],
                    'Labels': args[7],
                    'Skip graph generation': True,
                    # Has to be randomly changed in order for metrics
                        to be re-calculated
                    'Epsilon' : 1.0 / float( randint(0,1000000) ),
                    'Hypothesis' : theta
                }
                trainingRes = self.test( Xtrain, Ytrain, theta, args
                    [7], extra=extra )
                validationRes = self.test( Xcv, Ycv, theta, args[7],
                    extra=extra )
                self.addLearningCurveIntervalStep(\
                    (
                        ( 1.0 - float( trainingRes['Accuracy'] )),
                        ( 1.0 - float( validationRes['Accuracy'] ))
                    )
                 )

    def _compute(self, theta, *args):

        Xtrain = args[0]
        Ytrain = args[1]
        lmbda = args[2]
        input_layer_size = args[3]
        hidden_layer_size = args[4]
        num_labels = args[5]

        # Unroll parameters
        Theta1 = numpy.reshape(
            theta[0:((input_layer_size+1) * (hidden_layer_size))],
            ((input_layer_size+1),(hidden_layer_size))
        ).T

        Theta2 = numpy.reshape(
            theta[((input_layer_size+1) * (hidden_layer_size)):],
            ( (num_labels), (hidden_layer_size+1) )
        )

        # Setup some useful variables
        m = MachineLearningFactory.getSampleSize( Xtrain )

        #% You need to return the following variables correctly
        Theta1_grad = numpy.zeros( ( numpy.shape( Theta1 ) ) )
        Theta2_grad = numpy.zeros( ( numpy.shape( Theta2 ) ) )

        # Activation function
        sigmoid = MachineLearningFactory.sigmoid
        sigmoidGradient = MachineLearningFactory.sigmoidGradient

        # Cost calculation
        J = 0
        m1 = numpy.matrix('1')

        for i in range( m ):
            # ==== Feed-forward ==== #
            A1 = numpy.matrix( numpy.hstack( ( m1, Xtrain[i,:]) ), copy
                =False )
            Z2 = Theta1 * A1.T
            A2 = sigmoid( Z2 ).T
            Z3 = Theta2 * numpy.matrix( numpy.hstack( ( numpy.ones( (
                MachineLearningFactory.getSampleSize( A2 ), 1 ) ), A2 )
                ), copy=False ).T
            A3 = sigmoid( Z3 )
            # K class adjustments
            z = numpy.zeros( ( 1, num_labels ) )
            z[ 0, Ytrain[i,0] ] = 1
            # Cost calculation
            try:
                # Divide by zero exception has occurred in numpy.log or
                     numpy.subtract
                J = J + numpy.sum( numpy.subtract( ((-z*numpy.log(A3)))
                    , ((1-z)*numpy.log(1-A3))))
            except FloatingPointError, e:
                self.addTestResult("Error while calculating cost", e )

            # ==== Backpropagation ==== # (gradient finding / error
                detection)
            D3 = A3 - numpy.matrix( z ).T
            D2 = numpy.multiply( Theta2.transpose()[1:,:] * D3,
                sigmoidGradient( Z2 ) )
            Theta2_grad = Theta2_grad + ( D3 * numpy.matrix( numpy.
                hstack( ( m1, A2 ) ), copy=False ) )
            Theta1_grad = Theta1_grad + ( D2 * A1 )

        # Last cost step, divide by sample size
        Theta1_grad = numpy.divide( Theta1_grad, float(m) )
        Theta2_grad = numpy.divide( Theta2_grad, float(m) )
        J = J / float(m)

        if lmbda <= 0:
            self.cost = J
        else:
            Theta1_grad[:,1:] = Theta1_grad[:,1:] + numpy.multiply(
                Theta1[:,1:] , lmbda/float(m) )
            Theta2_grad[:,1:] = Theta2_grad[:,1:] + numpy.multiply(
                Theta2[:,1:] , lmbda/float(m) )
            R = numpy.sum( numpy.power( Theta1[1:,:], 2 ) )
            R = R + numpy.sum( numpy.power( Theta2[1:,:], 2 ) )
            self.cost = J + lmbda/(2*m) * R

        # Unroll parameters
        self.grad = numpy.hstack(
            ( numpy.ravel( Theta1_grad ), numpy.ravel( Theta2_grad ) )
        )

    #===========================================#
    #    VIZUALIZATION                          #
    #===========================================#

    def vizualize(self,*args):
        '''
            Visualize the model.
        '''
        if getuser() in ['haaland','fredrik']:
```

```
# Fetch arguments
Xtest = args[0]
classCount = args[1]
hiddenLayer = args[2]
predictions = args[3]

# Other useful variables
colormap = cm.get_cmap('Greys')
intpol = [None,'nearest'][0]
featureCount = MachineLearningFactory.getFeatureSize( Xtest
     )
sampleCount =  MachineLearningFactory.getSampleSize( Xtest
    )
totalSize = sampleCount + featureCount + classCount + 3
maxInputFeatureValue = float( numpy.max( Xtest.ravel() ) )
minInputFeatureValue = float( numpy.min( Xtest.ravel() ) )

# Input layer visualization
for indx in range( sampleCount ):
    ax = plt.subplot2grid((1,totalSize), (0, indx))
    ax.imshow(
        Xtest[indx,:].T,
        vmin=minInputFeatureValue,
        vmax=maxInputFeatureValue,
        cmap=colormap,
        interpolation=intpol
    )
    ax.set_title( indx, fontsize=10 )
    ax.set_xticks([])
    if indx == 0:
        ax.set_yticks( [ i for i in range(featureCount) ] )
    else:
        ax.set_yticks([])

# Hidden layer visualization
layerAx = plt.subplot2grid((1,totalSize),(0,sampleCount+1),
    colspan=featureCount)
layerAx.imshow( hiddenLayer.T, cmap=colormap, interpolation
    =intpol )
layerAx.set_yticks([])
layerAx.set_xticks([])
layerAx.set_title("Hidden Layer", fontsize=10 )

# Result layer visualization
for indx in range(classCount):
    resultAx = plt.subplot2grid((1,totalSize), (0,
        sampleCount+featureCount+indx+2) )
    resultAx.imshow( predictions[:,indx], vmin=0.0, vmax
        =1.0, cmap=colormap, interpolation=intpol )
    resultAx.set_xticks([])
    resultAx.set_title("C%d" % indx, fontsize=10)
    if indx == 0:
        resultAx.set_yticks([ i for i in range(sampleCount)
             ])
    else:
        resultAx.set_yticks([])

# Display!
```

```
            plt.show()
```

MLDecisionTree

MLDecisionTree.py

```
import time
import numpy

from getpass import getuser

from sklearn.tree import export_graphviz
from sklearn.tree import ExtraTreeClassifier
from quick.ml.api.algorithm.MLAlgorithm import MLAlgorithm
from quick.ml.implementation.factory.MachineLearningFactory import
    MachineLearningFactory
from quick.ml.implementation.structure.MLPredicter import MLPredicter

class MLDecisionTree(MLAlgorithm):
    '''
        Supervised Multiple Class Learning Algorithm.
    '''

    def __init__(self):
        MLAlgorithm.__init__(self)
        self.setDoublePrecision( 3 )

    # Override
    def arangeTest(self, Xtest, hasLabels=True):
        '''
            Using default built-in method.
            @see: arangeTestDefault()
        '''
        return self.arangeTestDefault( Xtest, hasLabels )

    # Override
    def arangeAll(self, Xtrain, cvfactor=0.2):
        '''
            If possible, all sets are returned according to the
            built-in method for supervised learning algorithms.
            @see: arangeAllMultiClassSupervised()

            If not enough labels in Xcv, then it will be merged
            with Xtrain and returned as None ( both Xcv and Ycv ).
        '''
        Xtrain, Ytrain, Xcv, Ycv = self.arangeAllMultiClassSupervised(
            Xtrain, cvfactor )
        num_labels_train = len( MachineLearningFactory.
            countDistinctClasses( Ytrain, True ).keys() )
        num_labels_cv = len( MachineLearningFactory.
            countDistinctClasses( Ycv, True ).keys() )
        if num_labels_train == num_labels_cv:
            return Xtrain, Ytrain, Xcv, Ycv
        else:
            # Concatenate Xcv with Xtrain, return None as cross-
                validation
            Xtrain = numpy.vstack( (Xtrain,Xcv) )
            Ytrain = numpy.vstack( (Ytrain,Ycv) )
            return Xtrain, Ytrain, None, None

    # Override
    def learn(self, Xtrain, Ytrain, Xcv, Ycv,
        minimum=1e-05, lmbda=0, learningCurveIntervals=None, **kwArgs):
        '''
            Supervised Learning Algorithm.

            Not using <em>Ytrain</em> for learning,
            only for cross validating in order
            to find the threshold (Epsilon) .

            Not taking <em>minimum</em>, <em>lambda</em>
            nor <em>maxiter</em> into account.

            Will in later implementation make use of
            <em>learningCurveIntervals</em>.
        '''
        # Parse optional argument(s)
        if kwArgs.has_key('outputDotGraph'):
            outputDotGraph = kwArgs.get('outputDotGraph')
        else:
            outputDotGraph = False
            outputDotGraph = True

        # Detect classes
        classValues = MachineLearningFactory.countDistinctClasses(
                            Ytrain, returnAggregation=True ).keys()

        # Map matrices into lists in order to
        # work together with the sklearn API.
        Xlist = Xtrain.tolist()
        Ytrain, encoder = MachineLearningFactory.
            getLabelsAsIntegerRepresentation( Ytrain )
        Ylist = Ytrain.ravel().tolist()

        # Learning step
        clf = ExtraTreeClassifier(
            criterion='entropy',
            compute_importances = True
        )
        clf.fit( Xlist, Ylist )

        # Mandatory labels
        self.addLearningResult( 'Classes', classValues )
        self.addLearningResult( 'n-Class', len(classValues) )
        self.addLearningResult( 'Classifier', self.saveToDisk( clf ) )

        self.addLearningResult( 'Extra parameters', {
            'Label encoder' : encoder,
            'criterion' : 'entropy'
        })

        # Possibly generate learning curve
        if learningCurveIntervals is not None and Xcv is not None and
            Ycv is not None:
            self.createLearningGraph( clf, Xtrain, Ytrain, Xcv, Ycv,
                learningCurveIntervals )
        # Export DOT graph, if set
        if outputDotGraph:
            dotGraph = export_graphviz( clf )
            dotGraph.close()
            dotGraph = open( dotGraph.name, 'r' )
            dotContent = ""
            for line in dotGraph:
                dotContent = dotContent + line
            dotSource = self.saveToDisk( dotContent, serialize=False )
            self.addLearningResult( 'DOT graph', str( dotSource ) )
            if getuser() in ['haaland','fredrik']:
                import os
                source = open('/tmp/dot_display.txt','w')
                source.write( dotContent )
                source.close()
                os.system("dot /tmp/dot_display.txt -Tpng -o /tmp/
                    dotgraph.png")
                #os.system('display /tmp/dotgraph.png')

        return self.getLearningResults()

    # Override
    def test(self, Xtest, Ytest, classifier, classes, extra={}, graphs=
        False, **kwArgs):
        '''
            Predict the <em>classes</em> of <em>Xtest</em>
            measuring performance with <em>Ytest</em>
            using the <em>classifier</em>, and possibly some
            <em>extra</em> dictionary parameters.
        '''
        # Re-construct classifier
        clf = self.loadClassifier( classifier )

        # Make sure classes are of correct type (int)
        for i in range( len( classes ) ):
            classes[ i ] = int( float( classes[ i ] ) )

        # Time predictions
        startTime = time.time()

        # Generate class representations
        classRepresentation = range( len( classes ) )

        # Containers to fill
        predicter = MLPredicter( clf, classRepresentation, Xtest, Ytest
            , self.getDoublePrecision() )
        Yprobabilities = predicter.getProbabilitiesForAllLabels()
        Yprobabilities_all = predicter.getProbabilitiesForPredictions()
        Yprobabilities_hits = predicter.
            getProbabilitiesForPositiveSamples()
        Ypredicted = numpy.matrix(
                        MachineLearningFactory.
                            getLabelsFromIntegerRepresentation(
                            predicter.getPredictions(),
                            extra.get('Label encoder') ), copy=False )

        # Assign time results
        self.addTestResult('Prediction runtime', '{} seconds'.format(
            time.time()-startTime))
        # Add graphs
        if graphs:
            self.addProbabilityGraphForClasses( classes, Yprobabilities
                 )
            self.addHistogramGeneralValuesRPlot( Yprobabilities_all )
            self.addHistogramHitsOnlyValuesRPLot( Yprobabilities_hits )

        # Add metrics and prediction tracks
        self.addPredictionTracks(
            Yprobabilities_all,
            Ypredicted,
            Yprobabilities,
            classes,
            extra
        )
        self.addEstimatesTracks(
            classes,
            Yprobabilities,
            extra
        )

        if Ytest is not None:
            self.addBasicTestMetricResults(
                Ytest,
                Ypredicted,
                Yprobabilities,
                classes,
                threshold = None
            )

        return self.getTestResults()
```

MLKNearestNeighbor

MLKNearestNeighbor.py

```
import time
import numpy
from sklearn.neighbors import KNeighborsClassifier
from quick.ml.api.algorithm.MLAlgorithm import MLAlgorithm
from quick.ml.implementation.factory.MachineLearningFactory import
    MachineLearningFactory

from getpass import getuser
from quick.ml.implementation.structure.MLPredicter import MLPredicter
if getuser() not in ['haaland','fredrik']:
    from quick.util.StaticFile import StaticFile
else:
    import matplotlib.pyplot as plt
    import matplotlib.mlab as mlab

class MLKNearestNeighbor(MLAlgorithm):
    '''
        Supervised Multiple Class Learning Algorithm.

        Ylist elements are integers due to an implementation detail
        inside the sklearn API method predict_proba().
    '''
    def __init__(self):
        MLAlgorithm.__init__(self)
        self.setDoublePrecision( 3 )

    # Override
    def arangeTest(self, Xtest, hasLabels=True):
        '''
            Using default built-in method.
            @see: arangeTestDefault()
        '''
        return self.arangeTestDefault( Xtest, hasLabels )

    # Override
    def arangeAll(self, Xtrain, cvfactor=0.2):
        '''
            Use built-in method for supervised learning algorithms.
            @see: arangeAllMultiClassSupervised()
        '''
        return self.arangeAllMultiClassSupervised( Xtrain, cvfactor )

    def normalize(self, X):
        '''
            Performs feature scaling to make the
            minimization algorithm converge
            more easily.

            Normalizes only contents, and not classes.
            @see: MachineLearningAlgorithmFactory
        '''
        return MachineLearningFactory.normalize( X )

    # Override
    def learn(self, Xtrain, Ytrain, Xcv, Ycv,
        minimum=1e-05, lmbda=1.0, learningCurveIntervals=None, **kwArgs
           ):
        '''
            Supervised Learning Algorithm.
        '''
        # Detect classes
        classValues = MachineLearningFactory.countDistinctClasses(
                            Ytrain, returnAggregation=True ).keys()

        # Map matrices into lists in order to
        # work together with the sklearn API.
        Xlist = Xtrain.tolist()
        Ytrain, encoder = MachineLearningFactory.
           getLabelsAsIntegerRepresentation( Ytrain )
        Ylist = Ytrain.ravel().tolist()

        # Learning step (using default parameters as default )
        clf = KNeighborsClassifier(
            n_neighbors = 5,
            weights     = 'distance',
            algorithm   = 'auto',
            leaf_size   = 30,
            warn_on_equidistant = False
        )
        clf.fit( Xlist, Ylist )
        # Mandatory labels
        self.addLearningResult( 'Classes', classValues )
        self.addLearningResult( 'n-Class', len(classValues) )
        self.addLearningResult( 'Classifier', self.saveToDisk( clf ) )

        # Add results and other parameters
        self.addLearningResult( 'Extra parameters', {
            'Label encoder' : encoder,
            'n_neighbors' : clf.n_neighbors,
            'weights': 'distance',
            'algorithm' : clf.algorithm,
            'leaf_size' : clf.leaf_size,
            'warn_on_equidistant' : clf.warn_on_equidistant
        })

        # Possibly generate learning curve
        if learningCurveIntervals is not None and Xcv is not None and
            Ycv is not None:
            self.createLearningGraph( clf, Xtrain, Ytrain, Xcv, Ycv,
                learningCurveIntervals )

        return self.getLearningResults()

    # Override
    def test(self, Xtest, Ytest, classifier, classes, extra={}, graphs=
        False, **kwArgs):
        '''
            Predict the <em>classes</em> of <em>Xtest</em>
            measuring performance with <em>Ytest</em>
            using the <em>classifier</em>, and possibly some
            <em>extra</em> dictionary parameters.
        '''
        # Re-construct classifier
        clf = self.loadClassifier( classifier )

        # Time predictions
        startTime = time.time()

        # Generate class representations
        classRepresentation = range( len( classes ) )

        # Containers to fill
        predicter = MLPredicter( clf, classRepresentation, Xtest, Ytest
            , self.getDoublePrecision() )
        Yprobabilities = predicter.getProbabilitiesForAllLabels()
        Yprobabilities_all = predicter.getProbabilitiesForPredictions()
        Yprobabilities_hits = predicter.
            getProbabilitiesForPositiveSamples()
        Ypredicted = numpy.matrix(
                        MachineLearningFactory.
                            getLabelsFromIntegerRepresentation(
                            predicter.getPredictions(),
                            extra.get('Label encoder') ), copy=False )
        # Assign time results
        self.addTestResult('Prediction runtime', '{} seconds'.format(
            time.time()-startTime))

        # Add graphs
        if graphs and Ytest is not None:
            self.addProbabilityGraphForClasses( classes, Yprobabilities
                 )
            self.addHistogramGeneralValuesRPlot( Yprobabilities_all )
            self.addHistogramHitsOnlyValuesRPLot( Yprobabilities_hits )
            self.vizualize( Xtest, Ytest, classes, Ypredicted, clf )

        # Add metrics and prediction tracks
        self.addPredictionTracks(
            Yprobabilities_all,
            Ypredicted,
            Yprobabilities,
            classes,
            extra
        )
        self.addEstimatesTracks(
            classes,
            Yprobabilities,
            extra
        )

        if Ytest is not None:
            self.addBasicTestMetricResults(
                Ytest,
                Ypredicted,
                Yprobabilities,
                classes,
                threshold = None
            )

        return self.getTestResults()

    #===========================================#
    #    VISUALIZATION                          #
    #===========================================#

    def vizualize(self,*args):
        '''
            Visualize the model.
        '''
        if getuser() in ['haaland','fredrik']:

            # Fetch arguments
            Xtest = args[0]
            Ytest = args[1]
            classes = args[2]
            predictions = args[3]
            clf = args[4]

#             X = clf.kneighbors_graph(
#                     Xtest,
#                     n_neighbors=None,
#                     mode='distance'
#             ).todense()
#             folder = '/home/%s/HB/ml/test/quick/ml/implementation/
    factory/files/' % getuser()
#             # Write Prediction data
#             fileName = "{}{}".format( folder, '
    kneighbors_graph_predict.npy' )
```

```
#            numpy.savetxt( fileName, X, fmt='%.2f', delimiter='\t',
    newline='\n' )
#            #print MachineLearningFactory.getSampleSize( X ), 'x',
    MachineLearningFactory.getFeatureSize( X )

            # Create dotgraph
            import os
            import pydot
            from pydot import Node, Edge, Subgraph

            # Add a graph with subgraph of class nodes
            graph = pydot.Graph("knn", graph_type='digraph', strict=
                False, simplify=False)
            subGraph = Subgraph("knnclasses", rank='same')

            # Create class nodes (centroids)
            # And create per class relations as subgraphs
            classNodes = list()
            samplesGraphs = list()
            for i, cls in enumerate(classes):
                classNode = Node(
                    "Class {}".format(cls),
                    color='black',
                    style='filled',
                    fontcolor='white',
                    fontsize='12'
                )
                subGraph.add_node( classNode )
                classNodes.append( classNode )
                sampleGraph = Subgraph("Samplesclass{}".format(i))
                samplesGraphs.append( sampleGraph )
                graph.add_subgraph( sampleGraph )
            graph.add_subgraph( subGraph )

            # Connect sample nodes to class nodes
            for i in range( MachineLearningFactory.getSampleSize(Xtest)
                 ):
                prediction = int( predictions[i] )
                classNode = classNodes[ prediction ]
                sampleNode = Node(
                    "S{} [{}]".format(i,int(Ytest[i])),
                    fontsize = '10',
                    style = 'dotted' if Ytest[i] != predictions[i] else
                         'solid',
                    fontcolor = 'red' if Ytest[i] != predictions[i]
                        else 'black'
                )
                graph.add_edge( Edge( sampleNode, classNode ) )
                sampleGraph = samplesGraphs[ prediction ]
                sampleGraph.add_node( sampleNode )

            # Display graph
            source = open('/tmp/dot_display.txt','w')
            source.write( graph.to_string() )
            source.close()
            #os.system("twopi /tmp/dot_display.txt -Tpng -o /home/
                haaland/HB/dotgraph{}.png".format(time.time()
                *10000000000000))
            os.system("twopi /tmp/dot_display.txt -Tpng -o /tmp/
                dotgraph.png")
            os.system('display /tmp/dotgraph.png')

        else:
            # TODO : Make rpy plot
            pass


    def vizualizeLearn(self, X):
        import os
        import pydot
        from pydot import Node, Edge
        from math import ceil

        graph = pydot.Graph("knndistance", graph_type='digraph', strict
            =True, simplify=False)

        nodes = dict()

        for i in range( MachineLearningFactory.getSampleSize(X) ):
            for j in range( i, MachineLearningFactory.getFeatureSize(X)
                 ):
                distance = int( ceil(X[i,j]) )
                if distance > 0:
                    srcKey = "S{}".format(i)
                    dstKey = "S{}".format(j)
                    # Source node
                    if nodes.has_key( srcKey ):
                        srcNode = nodes.get( srcKey )
                    else:
                        srcNode = Node( srcKey )
                        graph.add_node( srcNode )
                        nodes[ srcKey ] = srcNode
                    # Destination node
                    if nodes.has_key( dstKey ):
                        dstNode = nodes.get( dstKey )
                    else:
                        dstNode = Node( dstKey )
                        graph.add_node( dstNode )
                        nodes[ dstKey ] = dstNode
                    # Add edge between them
                    distance = distance + 1
                    graph.add_edge( Edge( srcNode, dstNode, len=str(
                        distance), minlen='1', dir='both' ) )

        # Display graph
        source = open('/tmp/knn_dot_display.txt','w')
        source.write( graph.to_string() )
        source.close()
        os.system("fdp /tmp/knn_dot_display.txt -Tpng -o /tmp/
            knn_dotgraph.png")
        os.system('display /tmp/knn_dotgraph.png')
```

MLLogisticRegressionRegularized

MLLogisticRegressionRegularized.py

```
import numpy
import time

from sklearn.linear_model import LogisticRegression

from quick.ml.api.algorithm.MLAlgorithm import MLAlgorithm
from quick.ml.implementation.factory.MachineLearningFactory import
    MachineLearningFactory

from getpass import getuser
from quick.ml.implementation.structure.MLPredicter import MLPredicter
if getuser() not in ['haaland','fredrik']:
    from quick.util.StaticFile import StaticFile
else:
    import matplotlib.pyplot as plt
    import matplotlib.mlab as mlab

class MLLogisticRegressionRegularized(MLAlgorithm):
    '''
        Supervised (Multi-class) Learning Algorithm.

        Author: <fredrhaa> Fredrik Haaland
    '''
    def __init__(self):
        MLAlgorithm.__init__(self)

    # Override
    def arangeTest(self, Xtest, hasLabels=True):
        '''
            Using default built-in method.
            @see: arangeTestDefault()
        '''
        return self.arangeTestDefault( Xtest, hasLabels )

    # Override
    def arangeAll(self, Xtrain, cvfactor=0.2):
        '''
            Using default built-in method for
            supervised learning algorithms.
            @see: arangeAllBinaryClassSupervised()
        '''
        return self.arangeAllMultiClassSupervised( Xtrain, cvfactor )

    def normalize(self, X):
        '''
            Performs feature scaling to make the
            minimization algorithm converge
            more easily.

            Normalizes only contents, and not classes.
            @see: MachineLearningAlgorithmFactory
        '''
        return MachineLearningFactory.normalize( X )

    # Override
    def learn(self, Xtrain, Ytrain, Xcv, Ycv, minimum=1e-05, lmbda=0,
            learningCurveIntervals=None, **kwArgs):
        '''
            Returns dictionary with hypothesis and threshold entry.
            Class weight may be a dictionary of class weights.
            Default is string 'auto'.
        '''
        # Parse optional argument(s)
        if kwArgs.has_key('class_weight'):
            class_weight = kwArgs.get('class_weight')
        else:
            class_weight = 'auto'

        # Detect classes
        classValues = MachineLearningFactory.countDistinctClasses(
                            Ytrain, returnAggregation=True ).keys()

        # Map matrices into lists in order to
        # work together with the sklearn API.
        Xlist = Xtrain.tolist()
        Ytrain, encoder = MachineLearningFactory.
            getLabelsAsIntegerRepresentation( Ytrain )
        Ylist = Ytrain.ravel().tolist()

        # Adjust regularization parameter to it's opposite,
        # the smaller C the bigger regularization
        C = 1e-20 if lmbda == 0 else 1.0 / float( lmbda )

        # Learning step
        clf = LogisticRegression(
            penalty = 'l2',
            dual = False,
            tol = minimum,
            C = C,
            fit_intercept = True,
            class_weight = class_weight,
            intercept_scaling = 1
        )
        clf.fit( Xlist, Ylist )

        # Mandatory labels
        self.addLearningResult( 'Classes', classValues )
        self.addLearningResult( 'n-Class', len(classValues) )
        self.addLearningResult( 'Classifier', self.saveToDisk( clf ) )

        # Add results and other parameters
        self.addLearningResult( 'Extra parameters', {
            'Label encoder' : encoder,
            'penalty' : clf.penalty,
            'dual' : clf.dual,
            'tol' : clf.tol,
            'C' : clf.C,
            'fit_intercept' : clf.fit_intercept
        })

        # Possibly generate learning curve
        if learningCurveIntervals is not None and Xcv is not None and
            Ycv is not None:
            self.createLearningGraph( clf, Xtrain, Ytrain, Xcv, Ycv,
                learningCurveIntervals )

        return self.getLearningResults()
```

```
# Override
def test(self, Xtest, Ytest, classifier, classes, extra={}, graphs=
    False, **kwArgs):
    '''
        Predict the <em>classes</em> of <em>Xtest</em>
        measuring performance with <em>Ytest</em>
        using the <em>classifier</em>, and possibly some
        <em>extra</em> dictionary parameters.
    '''
    # Re-construct classifier
    clf = self.loadClassifier( classifier )

    # Time predictions
    startTime = time.time()

    # Generate class representations
    classRepresentation = range( len( classes ) )

    # Containers to fill
    predicter = MLPredicter( clf, classRepresentation, Xtest, Ytest
         )
    Yprobabilities = predicter.getProbabilitiesForAllLabels()
    Yprobabilities_all = predicter.getProbabilitiesForPredictions()
    Yprobabilities_hits = predicter.
        getProbabilitiesForPositiveSamples()
    Ypredicted = numpy.matrix(
                    MachineLearningFactory.
                        getLabelsFromIntegerRepresentation(
                        predicter.getPredictions(),
                        extra.get('Label encoder') ), copy=False )

    # Assign time results
    self.addTestResult('Prediction runtime', '{} seconds'.format(
        time.time()-startTime))

    # Add graphs
    if graphs:
        self.addProbabilityGraphForClasses( classes, Yprobabilities
             )
        self.addHistogramGeneralValuesRPlot( Yprobabilities_all )
        self.addHistogramHitsOnlyValuesRPLot( Yprobabilities_hits )

    # Add metrics and prediction tracks
    self.addPredictionTracks(
        Yprobabilities_all,
        Ypredicted,
        Yprobabilities,
        classes,
        extra
    )
    self.addEstimatesTracks(
        classes,
        Yprobabilities,
        extra
    )

    if Ytest is not None:
        self.addBasicTestMetricResults(
            Ytest,
            Ypredicted,
            Yprobabilities,
            classes,
            threshold = None
        )

    return self.getTestResults()
```

MLMultipleLinearRegression

MLMultipleLinearRegression.py

```
import numpy
import time

from sklearn.linear_model import LinearRegression
from sklearn.metrics import zero_one_score

from quick.ml.api.algorithm.MLAlgorithm import MLAlgorithm
from quick.ml.implementation.factory.MachineLearningFactory import
    MachineLearningFactory

from getpass import getuser
from quick.ml.implementation.structure.MLPredicter import MLPredicter
if getuser() not in ['haaland','fredrik']:
    from quick.util.StaticFile import StaticFile
else:
    import matplotlib.pyplot as plt
    import matplotlib.mlab as mlab

class MLMultipleLinearRegression(MLAlgorithm):
    '''
        Supervised (Multi-class) Learning Algorithm.

        Author: <fredrhaa> Fredrik Haaland
    '''
    def __init__(self):
        MLAlgorithm.__init__(self)

    # Override
    def arangeTest(self, Xtest, hasLabels=True):
        '''
            Using default built-in method.
            @see: arangeTestDefault()
        '''
        return self.arangeTestDefault( Xtest, hasLabels )

    # Override
    def arangeAll(self, Xtrain, cvfactor=0.2):
        '''
            PLits the data based on the cross validation factor.
        '''
        numpy.random.shuffle( Xtrain )
        Xtrain, Xcv = MachineLearningFactory.splitSamples( Xtrain,
            cvfactor )
        Xtrain, Ytrain = MachineLearningFactory.splitXandY( Xtrain )
        Xcv, Ycv = MachineLearningFactory.splitXandY( Xcv )
        return Xtrain, Ytrain, Xcv, Ycv

    # Override
    def learn(self, Xtrain, Ytrain, Xcv, Ycv, minimum=1e-05, lmbda=0,
              learningCurveIntervals=None, **kwArgs):
        '''
            Returns dictionary with hypothesis and threshold entry.

            Class weight may be a dictionary of class weights.
            Default is string 'auto'.
        '''
        # Detect classes
        classValues = MachineLearningFactory.countDistinctClasses(
                          Ytrain, returnAggregation=True ).keys()

        # Map matrices into lists in order to
        # work together with the sklearn API.
        Xlist = Xtrain.tolist()
        Ylist = Ytrain.tolist()

        # Learning step
        clf = LinearRegression(
            fit_intercept = True,
            normalize = False,
            copy_X = False
        )
        clf.fit( Xlist, Ylist, n_jobs=10 )

        # Mandatory labels
        self.addLearningResult( 'Classes', classValues )
        self.addLearningResult( 'n-Class', len(classValues) )
        self.addLearningResult( 'Classifier', self.saveToDisk( clf ) )

        # Add results and other parameters
        self.addLearningResult( 'Extra parameters', {
            'Label encoder' : 'None',
            'fit_intercept' : clf.fit_intercept,
            'normalize'     : False,
            'copy_X'        : False
        })

        # Possibly generate learning curve
        if learningCurveIntervals is not None and Xcv is not None and
            Ycv is not None:
            self.createLearningGraph( clf, Xtrain, Ytrain, Xcv, Ycv,
                learningCurveIntervals )

        return self.getLearningResults()

    # Override
    def test(self, Xtest, Ytest, classifier, classes, extra={}, graphs=
        False, **kwArgs):
        '''
            Predict the <em>classes</em> of <em>Xtest</em>
            measuring performance with <em>Ytest</em>
            using the <em>classifier</em>, and possibly some
            <em>extra</em> dictionary parameters.
        '''
        # Re-construct classifier
        clf = self.loadClassifier( classifier )

        # Time predictions
        startTime = time.time()

        # Predictions
        predicter = MLPredicter( clf, classes, Xtest, Ytest,
            computeProbabilities=False )

        # Make predictions
        Ypredicted = predicter.getPredictions()

        # Assign time results
        self.addTestResult('Prediction runtime', '{} seconds'.format(
            time.time()-startTime))

        chromosome = extra.get('Chromosome') if extra.get('Chromosome')
             is not None else 'N/A'
        genome = extra.get('Genome') if extra.get('Genome') is not None
             else 'N/A'
        start = extra.get('Start position') if extra.get('Start
            position') is not None else 0
        end = extra.get('End position') if extra.get('End position') is
             not None else numpy.size(Ypredicted,0)

        # Add class prediction track
        source = self.getPointTrack( genome, { chromosome: Ypredicted.
            ravel().tolist()[0] },
                                        'function', start=start, end=end )
        content = "# Class Prediction Track\n{}".format(source.read())
        self.addTestResult(
            'Class Prediction Track', self.saveToDisk( content,
                serialize=False )
        )
        source.close()
        # Add region track
        source = self.getPointTrack( genome, { chromosome: [(start,end)
            ] }, 'segments' )
        content = "# Region Track\n{}".format(source.read())
        self.addTestResult(
            'Region Track', self.saveToDisk( content, serialize=False )
        )

        if Ytest is not None:
            self.addTestResult('Accuracy', zero_one_score( Ytest,
                Ypredicted ) )

        return self.getTestResults()
```

MLSupportVectorMachine

MLSupportVectorMachine.py

```
import time
import numpy
from sklearn import svm
from sklearn.grid_search import GridSearchCV
from copy import copy

from quick.ml.api.algorithm.MLAlgorithm import MLAlgorithm
from quick.ml.implementation.factory.MachineLearningFactory import
    MachineLearningFactory
from quick.ml.implementation.structure.MLPredicter import MLPredicter

class MLSupportVectorMachine(MLAlgorithm):
    '''
        Supervised Learning Algorithm.
    '''
    def __init__(self, kernel='rbf'):
        MLAlgorithm.__init__(self)
        self.setKernel( kernel )

    # Override
    def arangeTest(self, Xtest, hasLabels=True):
        '''
            Using default built-in method.
            @see: arangeTestDefault()
        '''
        return self.arangeTestDefault( Xtest, hasLabels )

    def normalize(self, X):
        '''
            Performs feature scaling to make the
            minimization algorithm converge
            more easily.

            Normalizes only contents, and not classes.
            @see: MachineLearningAlgorithmFactory
        '''
        return MachineLearningFactory.normalize( X )

    # Override
    def arangeAll(self, Xtrain, cvfactor=0.2):
        '''
            Use built-in method for supervised learning algorithms.
            @see: arangeAllMultiClassSupervised()
        '''
        return self.arangeAllMultiClassSupervised( Xtrain, cvfactor )

    # Override
    def learn(self, Xtrain, Ytrain, Xcv, Ycv,
        minimum=1e-05, lmbda=0, learningCurveIntervals=None, **kwArgs):
        '''
            Un-supervised Learning Algorithm.

            Not using <em>Ytrain</em> for learning,
            only for cross validating in order
            to find the threshold (Epsilon) .

            Not taking <em>minimum</em>, <em>lambda</em>
            nor <em>maxiter</em> into account.

            Will in later implementation make use of
            <em>learningCurveIntervals</em>.
        '''
```

```
# Map matrices into lists in order to
# work together with the sklearn API.
Xlist = Xtrain.tolist()
Ytrain, encoder = MachineLearningFactory.
    getLabelsAsIntegerRepresentation( Ytrain )
Ylist = Ytrain.ravel().tolist()

# Detect classes
classValues = MachineLearningFactory.countDistinctClasses(
                    Ytrain, returnAggregation=True ).keys()

# Detect parameters using a grid-search
parameters = {
    'C': MachineLearningFactory.getRegularizationValues().
        remove( 0.0 ), # May not be zero
    'gamma': MachineLearningFactory.getThresholdValues()
}

# Select C if lambda is chosen set
if lmbda >= 0:
    parameters['C'] = [ 1 ] if lmbda == 0 else [ 1.0 / float(
        lmbda ) ]

try:
    svc = svm.SVC(
        kernel=self.getKernel(),
        tol=self.getTolerance(),
        probability = True,
        shrinking = False
    )
    clf = GridSearchCV( svc, parameters, n_jobs=10, verbose=0 )
    clf = clf.fit( Xlist, Ylist )
    clf = clf.best_estimator_
except:
    # Parameter search is not supported directly,
    # so try doing it the *manual* way..
    best_score = 0
    best_clf = None
    for i in range(len(parameters['C'])):
        C = parameters['C'][i]
        for j in range(len(parameters['gamma'])):
            gamma = parameters['gamma'][j]
            clf = svm.SVC(
                C = C,
                gamma = gamma,
                probability = True,
                shrinking = False,
                kernel = self.getKernel(),
                tol = minimum
            )
            clf.fit( Xlist, Ylist )
            score = clf.score( Xlist, Ylist )
            if score > best_score or best_clf is None:
                best_clf = copy( clf )
                best_score = score
    clf = best_clf

# Mandatory labels
self.addLearningResult( 'Classes', classValues )
```

```
        self.addLearningResult( 'n-Class', len( classValues ) )
        self.addLearningResult( 'Classifier', self.saveToDisk( clf ) )

        # Add results and other parameters
        self.addLearningResult( 'Extra parameters', {
            'Label encoder' : encoder,
            'C': clf.C,
            'Gamma': clf._gamma,
            'probability' : clf.probability,
            'shrinking' : clf.shrinking,
            'kernel' : self.getKernel(),
            'tol' : minimum
        })

        # Possibly generate learning curve
        if learningCurveIntervals is not None and Xcv is not None and
            Ycv is not None:
            self.createLearningGraph( clf, Xtrain, Ytrain, Xcv, Ycv,
                learningCurveIntervals )

        return self.getLearningResults()

    # Override
    def test(self, Xtest, Ytest, classifier, classes, extra={}, graphs=
        False, **kwArgs):
        '''
            Predict the <em>classes</em> of <em>Xtest</em>
            measuring performance with <em>Ytest</em>
            using the <em>classifier</em>, and possibly some
            <em>extra</em> dictionary parameters.
        '''
        # Re-construct classifier
        clf = self.loadClassifier( classifier )

        # Time predictions
        startTime = time.time()

        # Generate class representations
        classRepresentation = range( len( classes ) )

        # Containers to fill
        predicter = MLPredicter( clf, classRepresentation, Xtest, Ytest
            , self.getDoublePrecision() )
        Yprobabilities = predicter.getProbabilitiesForAllLabels()
        Yprobabilities_all = predicter.getProbabilitiesForPredictions()
        Yprobabilities_hits = predicter.
            getProbabilitiesForPositiveSamples()
        Ypredicted = numpy.matrix(
                        MachineLearningFactory.
                            getLabelsFromIntegerRepresentation(
                            predicter.getPredictions(),
                            extra.get('Label encoder') ), copy=False )

        # Assign time results
        self.addTestResult('Prediction runtime', '{} seconds'.format(
            time.time()-startTime))

        # Add graphs
        if graphs:
            self.addProbabilityGraphForClasses( classes, Yprobabilities
                 )
            self.addHistogramGeneralValuesRPlot( Yprobabilities_all )
            self.addHistogramHitsOnlyValuesRPLot( Yprobabilities_hits )

        # Add metrics and prediction tracks
        self.addPredictionTracks(
            Yprobabilities_all,
            Ypredicted,
            Yprobabilities,
            classes,
            extra
        )
        if Ytest is not None:
            self.addBasicTestMetricResults(
                Ytest,
                Ypredicted,
                Yprobabilities,
                classes,
                threshold = None
            )

        return self.getTestResults()

    #============================================#
    #    HELPER METHODS                          #
    #============================================#

    def setKernel(self, kernel):
        '''
            Sets the kernel to be used.
            @see: sklearn library for allowed kernels
        '''
        self.kernel = kernel

    def getKernel(self):
        '''
            Returns the name of the kernel to be used.
        '''
        return self.kernel
```

A.2.2 Measures

Response measures

MLResponsePointExists.py

```
from quick.ml.api.feature.MLResponse import MLResponse

class MLResponsePointExists(MLResponse):
    '''
        Checks if a point is present or not.

        Return
        ----------------------------------------
        Returns 1 if a point is present,
        or 0 otherwise.
        Notice
        ----------------------------------------
        Does not check point value, only existence.

    '''
    def __init__(self):
        MLResponse.__init__(self)

    def getValue(self,trackState,n):
        if trackState.isPoint() or trackState.isInsideSegment():
            return 1
        else:
            return 0
```

MLResponsePointValue.py

```
from quick.ml.api.feature.MLResponse import MLResponse

class MLResponsePointValue(MLResponse):
    '''
        Checks if a point is within another point,
        which implies that any point inside segments will return 1 also
            .
    '''
    def __init__(self):
        MLResponse.__init__(self)

    def getValue(self,trackState,n):
        if trackState.isPoint() or trackState.isInsideSegment():
            return trackState.getCurrentValue()
        else:
            return self.getUndefined()
```

MLResponseSegmentExists.py

```
from quick.ml.api.feature.MLResponse import MLResponse

class MLResponseSegmentExists(MLResponse):
    '''
        Checks if a segment is present.

        Return
        --------------------------------------
        Return 1 if the segment is present,
        otherwise 0.

    '''
    def __init__(self):
        MLResponse.__init__(self)

    def getValue(self,trackState,n):
        if trackState.isInsideSegment():
            return 1
        else:
            return 0
```

MLResponseSegmentValue.py

```
from quick.ml.api.feature.MLResponse import MLResponse

class MLResponseSegmentValue(MLResponse):
    '''
        Checks if a valued segment is present.

        Return
        --------------------------------------
        Return the value of the segment is present,
        otherwise 0.
    '''
    def __init__(self):
        MLResponse.__init__(self)

    def getValue(self,trackState,n):
        if trackState.isInsideSegment():
            return trackState.getCurrentValue()
        else:
            return self.getUndefined()
```

MLResponseFunctionValue.py

```
from quick.ml.api.feature.MLResponse import MLResponse
from quick.ml.implementation.feature.transformation.
    MLTransformationRoundOff import MLTransformationRoundOff

class MLResponseFunctionValue(MLResponse):
    '''
        Returns the current track state's value.
    '''
    def __init__(self, roundOff=3):
        '''
            Appends the roundoff transformation in order to
            keep a relative small number of classes.
        '''
        MLResponse.__init__(self)
        self.addTransformation( MLTransformationRoundOff() )
        self.setMetaData( 'roundoff', roundOff )

    def getValue(self,trackState,n):
        return trackState.getCurrentValue()
```

Feature measures

MLFeaturePositionRelative.py

```
from quick.ml.api.feature.MLFeature import MLFeature

class MLFeaturePositionRelative(MLFeature):
    '''
        Returns the relative position from
        the tracks start point to the current point.
    '''
```

```python
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        return trackState.getCurrentPosition() / float( n - 1 )
```

MLFeaturePositionRelativeInverted.py

```python


from quick.ml.api.feature.MLFeature import MLFeature

class MLFeaturePositionRelativeInverted(MLFeature):
    '''
        Returns the inverted relative position
        Relative position regarding the track view end point.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        return 1.0 - trackState.getCurrentPosition() / float( n - 1 )
```

MLFeaturePositionRelativeCenter.py

```python

from quick.ml.api.feature.MLFeature import MLFeature

from math import floor

class MLFeaturePositionRelativeCenter(MLFeature):
    '''
        Returns the relative position
        to the tracks midpoint.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        k = trackState.getCurrentPosition()
        midpoint = floor( n / 2 )
        if k == midpoint:
            return 0.0
        elif k < midpoint:
            return 1.0 - ( float(k) / midpoint )
        else:
            return - 1.0 + ( float(k) / midpoint )
```

MLFeaturePositionRelativeSides.py

```python

from quick.ml.api.feature.MLFeature import MLFeature

class MLFeaturePositionRelativeSides(MLFeature):
    '''
        Relative position regarding the track view center point.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        k = trackState.getCurrentPosition()
        midpoint = ( n / 2 )
        if k == midpoint:
            return 1.0
        elif k < midpoint:
            return (1.0 * k) / midpoint
        else:
            return 2.0 - ((1.0 * k) / midpoint)
```

MLFeaturePointDistanceLast.py

```
from quick.ml.api.feature.MLFeature import MLFeature

class MLFeaturePointDistanceLast(MLFeature):
    '''
        Returns the number of positions from the current
        position or the start of the current element
        to the end of the last seen element.

        If no such element, then the undefined value is returned.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        if trackState.getLastPoint() is None:
            # If ran past current segment,
            # but not yet entered the next
            if trackState.getCurrentPosition() > trackState.
               getCurrentEndPosition():
                return trackState.getCurrentPosition() - trackState.
                    getCurrentEndPosition() - 1
            else:
                return self.getUndefined()
        # Else a last point exists
        elif trackState.isPoint() or trackState.isInsideSegment():
            # Inside an element
            return trackState.getCurrentStartPosition() - trackState.
               getLastPoint()[0] - 1
        elif trackState.getCurrentPosition() == trackState.
            getCurrentEndPosition() + 1:
            # If at the edge of the current element
            return 0
        else:
            # Outside an element
            return trackState.getCurrentPosition() - trackState.
               getCurrentEndPosition() - 1
```

MLFeaturePointDistanceFuture.py

```
from quick.ml.api.feature.MLFeature import MLFeature

class MLFeaturePointDistanceFuture(MLFeature):
    '''
        Returns the number of positions from the current
        position or the end of the current element
        and to the start of the future element.

        If no such element, then the undefined value is returned.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        if trackState.getCurrentPosition() < trackState.
            getCurrentStartPosition():
            # Border case, have not entered the current element yet
            return trackState.getCurrentStartPosition() - trackState.
                getCurrentPosition()
        else:
            if trackState.getFuturePoint() is None:
                # Border case, have no more points to discover
                return self.getUndefined()
            # Future point exists
            else:
                if trackState.isPoint() or trackState.isInsideSegment()
                    :
                    # Inside an element
                    return trackState.getFuturePoint()[0] - trackState.
                        getCurrentEndPosition() - 1
                else:
                    # Outside an element
                    return trackState.getFuturePoint()[0] - trackState.
                        getCurrentPosition() - 1
```

MLFeaturePointDistanceOuter.py

```
from quick.ml.api.feature.MLFeature import MLFeature

class MLFeaturePointDistanceOuter(MLFeature):
    '''
        The distance to the closest point (or segment)
        outside the current element.

        If point or inside segment,
        the outer sides determines the outer distance.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        # Fetch last point
        if trackState.getLastPoint() is None:
            # If ran past current segment,
            # but not yet entered the next
            if trackState.getCurrentPosition() > trackState.
                getCurrentEndPosition():
                lastval = trackState.getCurrentPosition() - trackState.
                    getCurrentEndPosition() - 1
            else:
                lastval = self.getUndefined()
        # Else a last point exists
        elif trackState.isPoint() or trackState.isInsideSegment():
            # Inside an element
            lastval = trackState.getCurrentStartPosition() - trackState
                .getLastPoint()[0] - 1
        elif trackState.getCurrentPosition() == trackState.
            getCurrentEndPosition() + 1:
            # If at the edge of the current element
            lastval = 0
        else:
            # Outside an element
            lastval = trackState.getCurrentPosition() - trackState.
                getCurrentEndPosition() - 1
        # Fetch future point
        if trackState.getCurrentPosition() < trackState.
            getCurrentStartPosition():
            # Border case, have not entered the current element yet
            futureval = trackState.getCurrentStartPosition() -
                trackState.getCurrentPosition()
        else:
            if trackState.getFuturePoint() is None:
                # Border case, have no more points to discover
                futureval = self.getUndefined()
            else:
                # Future point exists
                futurePointPosition = trackState.getFuturePoint()[0]
                currentEndPosition = trackState.getCurrentEndPosition()
                fullDistance = futurePointPosition - currentEndPosition
                     - 1
                if trackState.isPoint() or trackState.isInsideSegment()
                    :
                    # Inside an element
                    futureval = float( fullDistance )
                else:
                    # Outside an element
                    currentPosition = trackState.getCurrentPosition()
                    futureval = float( futurePointPosition -
                        currentPosition - 1 )
        # Decide smallest
        if lastval == self.getUndefined():
            return futureval
        elif futureval == self.getUndefined():
            return lastval
        else:
            return min( lastval, futureval )
```

MLFeaturePointDistanceInner.py

```
from quick.ml.api.feature.MLFeature import MLFeature

from math import floor

class MLFeaturePointDistanceInner(MLFeature):
    '''
        The distance to center (midpoint)
        of the current point or segment.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        '''
            Calculates the distances from the
            <em>trackState</em>'s current position
            to it's sides, giving the center position
            the highest value of all inner values.

            Returns undefined if not inside a point or a segment.

            Outputs a relative distance divided by <em>n</em>,
            but is 1 by default, resulting in no default action.
        '''
        if not trackState.isInsideSegment():
            return self.getUndefined()
        elif trackState.isPoint() or trackState.size() == 2:
            return 0
        else: # Inside segment, size > 2
            center = floor( trackState.size() / 2 )
            offset = int( trackState.getOffset() )
            if trackState.size() % 2 == 0 :
                # Double center
                if offset == center or offset == center - 1:
                    return 1
                elif offset < center:
                    return offset / (center - 1)
                else:
                    return (trackState.size() - offset - 1) / ( center
                        - 1 )
            else:
                # Single center
                if offset == center:
                    return 1
                elif offset < center:
                    return offset / center
                else:
                    return (trackState.size() - offset - 1) / center
```

MLFeatureFunctionSlope.py

```
from quick.ml.api.feature.MLFeature import MLFeature

class MLFeatureFunctionSlope(MLFeature):
    '''
        Returns the average slope of the points on both sides
        of the training track state value, or undefined
        if track state is None (not provided).
    '''
    def __init__(self):
        MLFeature.__init__(self)

    def getValue(self,trackState,n):
        if trackState is None:
            return self.getUndefined()
        else:
            if trackState.getFuturePoint() is not None and trackState.
                getLastPoint() is not None:
                # Both point are present, return the average slope
                return ( trackState.getFuturePoint()[1] - trackState.
                    getLastPoint()[1] ) / 3
            elif trackState.getFuturePoint() is None and trackState.
                getLastPoint() is None:
                return self.getUndefined()
            elif trackState.getFuturePoint() is None and trackState.
                getLastPoint() is not None:
                return ( trackState.getCurrentValue() - trackState.
                    getLastPoint()[1] ) / 2
            else:
                return ( trackState.getFuturePoint()[1] - trackState.
                    getCurrentValue() ) / 2
```

MLFeatureFunctionStrand.py

```
from quick.ml.api.feature.MLFeature import MLFeature

class MLFeatureFunctionStrand(MLFeature):
    '''
        Returns 1 if the training track state value
        is greater than or equal to zero, 0 if below zero,
        or undefined if track state is None (not provided).
    '''
    def __init__(self):
        MLFeature.__init__(self)

    def getValue(self,trackState,n):
        if trackState.getCurrentValue() != trackState.getUndefined():
            return trackState.getCurrentValue()
        else:
            if trackState.getCurrentValue() < 0:
                return 0
            else:
                return 1
```

MLFeatureSegmentStartPosition.py

```
from quick.ml.api.feature.MLFeature import MLFeature

class MLFeatureSegmentStartPosition(MLFeature):
    '''
        Returns 1 if State is a segment, and
        if the current position equals the start position,
        otherwise 0.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        if trackState.isAtStartPosition():
            return 1
        else:
            return 0
```

MLFeatureSegmentEndPosition.py

```
from quick.ml.api.feature.MLFeature import MLFeature

class MLFeatureSegmentEndPosition(MLFeature):
    '''
        Returns 1 if State is a segment, and
        if the current position equals the end position,
        otherwise 0.
    '''
    def __init__(self):
        MLFeature.__init__(self)

    # Override
    def getValue(self,trackState,n):
        if trackState.isAtEndPosition():
            return 1
        else:
            return 0
```

A.2.3 Transformations

MLTransformationFavourLeft.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationFavourLeft(MLTransformation):
    '''
        Returns the relative position within the
        element, concretely, a floating point from 0 to 1,
        where 0 is at the start of the element and 1 is
        the end of the element.

        In, the special case of a point, the relative
        length will be reported as 1.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        if featureData['len'] == 1:
            featureData['val'] = 1
        else:
            relativePosition = 1 - (featureData['pos'] / float(
                featureData['len']-1))
            addition = featureData['val'] * relativePosition
            featureData['val'] = featureData['val'] + addition

        return featureData
```

MLTransformationFavourRight.py

```

from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationFavourRight(MLTransformation):
    '''
        Returns the relative position within the
        element, concretely, a floating point from 0 to 1,
        where 0 is at the start of the element and 1 is
        the end of the element.

        In, the special case of a point, the relative
        length will be reported as 1.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        if featureData['len'] == 1:
            featureData['val'] = 1
        else:
            relativePosition = featureData['pos'] / float(featureData['
                len']-1)
            if relativePosition > 0:
                addition = (featureData['val'] * relativePosition)
                featureData['val'] = featureData['val'] + addition

        return featureData
```

MLTransformationFavourCenter.py

```

from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationFavourCenter(MLTransformation):
    '''
        Favors the data at the center of the element range,
        gradually decreasing towards the sides.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        addition = 0
        if featureData['len'] == 3:
            if featureData['pos'] == 1:
                addition = featureData['val']
        elif featureData['len'] > 3:
            size = float(featureData['len'])
            half = max(1,(size / 2.0) )
            center = featureData['start'] + half - 1
            if featureData['pos'] < center:
                offset = featureData['pos'] - featureData['start'] + 1
                addition = featureData['val'] * (offset / (half-1))
            elif featureData['pos'] == center:
                offset = featureData['pos'] - featureData['start']
                addition = featureData['val']
            else:
                offset = featureData['end'] - featureData['pos'] - 1
                addition = featureData['val'] * (offset / (half-1))
        featureData['val'] = featureData['val'] + addition
        return featureData
```

MLTransformationFavourSides.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationFavourSides(MLTransformation):
    '''
        Favors the data at the sides of the element range,
        gradually decreasing towards the center.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        addition = 0
        if featureData['len'] == 3:
            if featureData['pos'] == 0 or featureData['pos'] == 2:
                addition = featureData['val']
        elif featureData['len'] > 3:
            size = float(featureData['len'])
            half = max(2,(size / 2.0) )
            center = featureData['start'] + half - 1
            if featureData['pos'] < center:
                offset = featureData['start'] + featureData['pos'] - 1
                addition = featureData['val'] * (1-(offset / (half-1)))
            elif featureData['pos'] > center:
                offset = featureData['pos'] - center
                addition = featureData['val'] * (offset / (half-1))
        featureData['val'] = featureData['val'] + addition
        return featureData
```

MLTransformationRelative.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationRelative(MLTransformation):
    '''
        Returns the value divided by the 'length'
        key, defaulted to track length.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        featureData['val'] = float( featureData['val'] ) / featureData[
            'len']
        return featureData
```

MLTransformationLogarithmic.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

from math import log10

class MLTransformationLogarithmic(MLTransformation):
    '''
        Returns the base 10 logarithmic value of
        the value inside <em>featureData</em>.

        Returns an 'undefined' value, for value 0
        and negative numbers.

        The 'undefined' value is the key
        from <em>featureData</em> if set,
        or else the lowest possible integer.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        if featureData['val'] < 0:
            featureData['val'] = self.getUndefined()
        else:
            featureData['val'] = log10( 1 + featureData['val'] )
        return featureData
```

MLTransformationConditionEqualsValue.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationConditionEqualsValue(MLTransformation):
    '''
        Returns 1 if value equals the given
        value of the 'comparison' key,
        or otherwise 0.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        comparison = featureData.get('comparison',None)
        if comparison is not None:
            if featureData['val'] == comparison:
                featureData['val'] = 1
            else:
                featureData['val'] = 0
        return featureData
```

MLTransformationConditionLessThanValue.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationConditionLessThanValue(MLTransformation):
    '''
        Returns 1 if value is less than
        the given value of the 'comparison' key,
        or otherwise 0.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):

        comparison = featureData.get('comparison',None)
        if comparison is not None:
            if featureData['val'] < comparison:
                featureData['val'] = 1
            else:
                featureData['val'] = 0
        return featureData
```

MLTransformationConditionGreaterThanValue.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationConditionGreaterThanValue(MLTransformation):
    '''
        Returns 1 if value is greater than
        the given value of the 'comparison' key,
        or otherwise 0.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        comparison = featureData.get('comparison',None)
        if comparison is not None:
            if featureData['val'] > comparison:
                featureData['val'] = 1
            else:
                featureData['val'] = 0
        return featureData
```

MLTransformationRoundOff.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation
```

```
class MLTransformationRoundOff(MLTransformation):
    '''
        Returns the rounded value of
        the value inside <em>featureData</em>.

        The number of decimals defaults to 5,
        but may be overridden by setting
        the mete-data key 'round' to the number
        of decimals.

        Does not support negative numbers.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        if featureData.has_key('roundoff'):
            featureData['val'] = round( featureData['val'], featureData
                .get('roundoff') )
        else:
            featureData['val'] = round( featureData['val'], 5 )
        return featureData
```

MLTransformationPolynomial.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationPolynomial(MLTransformation):
    '''
        Returns the value raised top the power
        of the 'polynomial' key, defaults to 1.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        if featureData.has_key('polynomial'):
            featureData['val'] = featureData['val'] ** featureData['
                polynomial']
        return featureData
```

MLTransformationAngle.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

from math import atan

class MLTransformationAngle(MLTransformation):
    '''
        Returns the inverse tangent (arc tan) of the value.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)
    def getTransformedValue(self, featureData):
        featureData['val'] = atan( featureData['val'] )
        return featureData
```

MLTransformationProduct.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

class MLTransformationProduct(MLTransformation):
    '''
        Returns the product of the 'factor' key
        and the value inside <em>featureData</em>.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        if featureData.has_key('factor'):
            featureData['val'] = float( featureData['val'] ) *
                featureData['factor']
        return featureData
```

MLTransformationExponential.py

```
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation

from math import expm1

class MLTransformationExponential(MLTransformation):
    '''
        Returns the exponential value of
        the value inside <em>featureData</em>.

        Returns the 'undefined' value, for 0
        and negative numbers.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        if featureData['val'] < 0:
            featureData['val'] = self.getUndefined()
        else:
            featureData['val'] = expm1( featureData['val'] )
        return featureData
```

MLTransformationSquareRoot.py

```
from math import sqrt
from quick.ml.api.feature.transformation.MLTransformation import
    MLTransformation
```

```
class MLTransformationSquareRoot(MLTransformation):
    '''
        Returns the square root of the value.
    '''
    def __init__(self, situation='all'):
        MLTransformation.__init__(self, situation)

    def getTransformedValue(self, featureData):
        featureData['val'] = sqrt( featureData['val'] )
        return featureData
```