UiO : Department of Informatics
University of Oslo

Teaching NLTK Norwegian

Bo Bjerke-Lindstrøm

Master's Thesis Autumn 2017

Abstract

In this thesis I will present an investigation of Natural Language Toolkit (NLTK) and its support for Norwegian Natural Language Processing (NLP). I display what NLTK has to offer for Norwegian NLP, then move on to evaluate and improving some of the offers NLTK has for Norwegian. I will evaluate and improve NLTK's sentences tokenizer and word tokenizer, I will also compare the tokenizers to other available options for Norwegian tokenization. The improvements will be committed to NLTK for possible integration. Then I will integrate a Norwegian corpus with a corpus reader to NLTK.

Acknowledgements

I want to thank my supervisor Jan Tore Lønning, your help has been invaluable.

I want to thank the University of Oslo and the Norwegian education system for giving me the possibility to achieve knowledge.

I want to thank the individuals behind the tools and data I have used.

I also want to thank my family for never ending support in my education.

And last but not least I want to thank the lights of my life, my significant other and daughter.

Contents

Chapter 1

1 Introduction

When learning Norwegian NLP, where would you turn? What book do you pick up, what site do you browse? What kind of tools are available?

The most obvious and natural starting point for me would be at the Department of Computer Science at the University of Oslo (UiO). Here all the information is open for the public and there are a few NLP courses. Most of them have a focus on pure NLP, which of course is key to Norwegian NLP. However some of the courses also has some focus on tasks revolving around Norwegian language.

The best place to start would be "INF1820 - Introduksjon til språk- og kommunikasjonsteknologi", this is the first NLP course a language technology major should take at UiO. It has slides and pod-casts from the lectures, exercises and solutions, every thing one would need to begin to learn 'Norwegian NLP'. After 1820 I would recommend "INF2820 - Datalingvistikk", this course is also represented by slides, pod-casts, exercises and solutions. Then you could just continue to pick classes/subjects that peak your interest.

Since I mention what book to pick up, I'll mention two books, 'Speech and Language Processing' (SLP) by Daniel Jurafsky and James H. Martin and 'Natural Language Processing with Python' (NLPP) by Steven Bird, Ewan Klein, and Edward Loper. SLP touches on almost every subject of NLP, while NLPP will get you started on the right path of being able executing these NLP subjects. NLPP is actually a book for a core tool that we use in some of the NLP courses at UIO, this tool is called NLTK [2001].

NLTK is a collection of tools for research, learning and teaching in symbolic and statistical NLP that is available and in my opinion easy to use! NLTK includes an interface to many corpora and lexical resources such as WordNet. NLTK also includes a suite of text processing libraries for tokenization, stemming, tagging, parsing, classification and semantic reasoning. It even has some Norwegian NLP support! NLTK contains a variety of tools (e.g. classifiers) that can be used for universally any language, it also has specialized tools (e.g. tagger) and data (e.g. corpora) for specific languages. NLTK should provide you with all the basic tools you need to begin your NLP adventure. There is also an active community, where one can discuss, learn and improve currently used tools and methods for NLP. Now that we have the tools to begin I guess we already have some text we want to processes? If not we are in luck! NLTK also contains many corpora to pick from!

1.1 NLTK for Norwegian

Since NLTK is a core tool for some of the courses it merits that we should look into the tools and data for Norwegian included in NLTK. NLTK has a good substructure/support for English language, but how good is the support for Norwegian? I will investigate this and compare the English support against the Norwegian support. I will compare to see what exists and what does not exist for Norwegian in NLTK. For the existing tools and data I will look for apparent errors and try to evaluate how well they really perform for Norwegian. I will then try to see if I can modify some of these tools and data to provide better support for Norwegian NLP.

NLTK is even an open source project, meaning we could try to integrate these improvements. This means we could also try to integrate support for Norwegian NLP that does not already exist in NLTK. The specialized tools and data for Norwegian included in NLTK is a stop-word list, stemmer and a sentence tokenizer. They also have a word tokenizer that we may apply for Norwegian. The stop-word list and stemmer does not have any real competitors to compare with, but there exists several word and sentence tokenizer for Norwegian. I will compare and evaluate some of these existing tokenizers vs the Norwegian tokenizers in NLTK.

As mentioned NLTK contains many corpora, but sadly no Norwegian corpus. However we are still in luck, as we can receive our Norwegian corpora elsewhere! The National Library of Norway's (NB) on-line repository contains a lot of open and available lexicon and corpora of Norwegian text. Since a body of text is needed, this is perfect for us. I will also try to integrate a Norwegian corpus into NLTK, to make Norwegian NLP more accessible to users.

The focus for this thesis will be to learn NLTK Norwegian. To try to elevate the support given for Norwegian NLP to a higher level closer to the support given for English NLP.

1.2 Overview

Chapter 2 An overview of the most common tools for NLP contained in NLTK, I try to map what exists and don't exist for Norwegian NLP in NLTK. And see if we have other alternatives for the parts that don't exist for Norwegian NLP in NLTK

Chapter 3 Revolves around NLTK's sentence tokenizer. It continues with an evaluation of this tokenizer from NLTK and improvements that can be done to it. There is also a comparison to other tokenizers.

Chapter 4 Revolves around NLTK's word tokenizer. It continues with an evaluation of this tokenizer from NLTK and improvements that can be done to it. There is also a comparison to other tokenizers.

Chapter 5 Integration of tokenizer patches. Here I investigate how we can integrate the improvements for the tokenizers into NLTK.

Chapter 6 In this chapter I discuss Norwegian lexical resources. Where can we obtain them, and what can be done to integrate a corpus into NLTK.

Chapter 7 Conclusive chapter, what have I done, what have I achieved? Also resources made and their availability.

Chapter 2

2 NLTK and tools for NLP

As I mentioned NLTK stands for Natural Language Toolkit. It is a collection of tools for research, learning and teaching in symbolic and statistical NLP. Some of the more useful tools and data to notice are sentence tokenizer, word tokenizer, lemmatizer, stemmer, tagger, classifier, and corpora. Most of these tools will be a part of the core to almost every NLP pipeline. And using only NLTK we could use all these tools and data, however not for Norwegian. NLTK has full support for English, but Norwegian is missing some important parts. We might find these parts elsewhere or have to create them ourself.

2.1 Tokenizers

For a computer a text is treated as a linear sequence of symbols, for a human a text is not treated as a linear sequence of symbols, but rather as segments of sentences, words and other meaningful elements. Tokenization is the process of splitting up this linear sequence into these segments of sentences, words and other meaningful elements.

Tokenization might be one of the more basic and easy to comprehend tasks of NLP. However, errors done in the tokenization process will propagate into downstream tasks, therefore correct tokenization is critical for achieving wanted results later on.

I will mainly look at tokenization of 'running well formed text'. By this I mean that I will look at tokenization of a body of text, distinct from headings, footnotes and such. And that it's well formed, constructed according to grammatical rules, such as a sentence should start with a capital letter (expect in some really rare cases). The reason for choosing this part of the natural language is that it is the most "serious" part of our language, books, articles, etc. However text comes in many forms, for instance chat text, HTML text, etc. this is why one would usually create a tokenizer customized for the specific data set, and wanted results from that set. For me the data set will be 'running well formed text' and wanted results are the sentences and tokens from this text.

2.1.1 Sentence tokenizer

Sentence segmentation or sentence tokenization is the task of identifying sentences in running text. In other words it's the task of trying to isolate each 'correct' set of words that is complete in itself. Defining 'correct' in the sense that one wants to find the natural sentence boundaries in the given text, rather than finding every possible sentence within a sentence.

When we want to solve this task the first thing that comes to mind (as a Norwegian), is to split the text at every period we find, '.'. And lets not forget the other sentence delimiters such as question mark and exclamation mark, '?', '!'. However as any NLP task it's not that easy, the most common error to this approach of 'splitting at delimiters' is the ambiguity that arise from abbreviations e.g. 'i.e.'. There are other issues as well, such as interposed sentences and how to handle colon, ':'. Is it more often correct to split a sentence at ':' or not?

If we want to do sentence tokenization of Norwegian language there exist some sentence tokenizers that we could use, an available and easy to use tokenizer is NLTK's [NLTK, 2001, web] sentence tokenizer. However there there might exist better options out there, I will compare and evaluate NLTK's tokenizer to a couple of the other sentence tokenizers for Norwegian. I will also do a failure analysis of NLTK's implementation and see if we can fix some of the most common errors that arise.

Sentence tokenization is arguably the first processes one should perform when wanting to analyze text.

2.1.2 Word tokenizer

Word tokenization or possibly token tokenization is the task of identifying all individual tokens in running text. In other words it is the task of trying to isolate each correct single distinct meaningful element of writing. However we define 'correct' in the sense that one wants to find the natural token boundaries in the given text, rather than finding every possible token (e.g. not every morpheme).

But what does it mean for a token to be complete? For the most part it means splitting on white-space to create word like unites, but what happens for instance with punctuation? Some would want to keep the punctuation attached to the word for a specific downstream task. However the most standard school of thought is to separate the punctuation, except 'infix punctuation' such as hyphens, from the word creating two tokens.

A word tokenizer can be language specific, or have language specific processes. For instance English has some word tokenizers that can also split on word contraction. In Norwegian we do not use contraction in written language, however we do have at least one language dependent structure to consider, large numbers. Large numbers are written with white space before every three digits, '10 000', instead of the more English version '10,000'. Should it be '10' and '000' as two tokens or '10 000' as one token? To answerer this question we have to consider what a token should represents. It should represent the complete meaning as intended in the given text. The meaningful parts of each element extracted in the tokenization process should not be altered or lost. The answer to the question should now be obvious, we want '10 000' as one token, as the meaning of '10' and '000' differ quite from the meaning of '10 000'. And there is also a question, what about collocations, such as 'i dag'?

If we want to do word tokenization of Norwegian language there exist some word tokenizers that we could use, there is one in NLTK that we will take a closer look at. We will also compare it to other options.

Word tokenization is arguably the second process one would perform when analyzing text, since we have the sentences, we now want to have the contents of each sentence.

2.2 Stemmer

Stemming is the raw heuristic process of removing the end of words, in the hope to remove common morphological and inflectional endings from the words. This can for instance be useful when we want to conflate words to increase the number of possible query matches.

In NLTK there are a couple of stemmers. One stemmer is called 'Porter stemmer' a stemmer developed by Martin Porter. Porter stemmer is set as 'frozen', this means that it's not being further developed and used as a frozen standard for comparing new stemmers against. Even though Porter stemmer is set to frozen we can create modifications of it, like 'Snowball stemmer'. Snowball is a modification/port of the Porter stemmer, and also included in NLTK. Snowball supports 13 languages, including Norwegian and English. When reading the documentation for Snowball we can see that for stemming Norwegian words Snowball use a list of suffixes for Norwegian language, and it applies one main rule called R1:

> *R1 is the region after the first non-vowel following a vowel, or is*
> *the null region at the end of the word if there is no such non-vowel.*
> *But then R1 is adjusted so that the region*
> *before it contains at least three letters.*
> Stahl and Ljunglof [2001]

This rule is meant to identify the region that defines the end of a word, where the common morphological and inflectional endings would be located. Then if the 'R1 region' ends with one of the suffixes in the suffix list, it will cut away the identified suffix from our word, giving us hopefully the correct stem. Other than this main function there are other functions for catching special cases. One special case that is not included is when a word ends with 'mm' an illegal word ending in Norwegian. This special case can be added with ease, however further improvement is tedious and slow. For instance adding more suffixes to the suffix-lexicon like 'ing', will result in the need for more specialized cases, that can be added. We can continue to make specialized rules until we are satisfied, but we will never be satisfied. As it's an impossible task to try to cover all special cases, as there is no formal neat way that all the words converge to. We can only make broad strokes with a stemmer, and thats why one might say that a stemmer is a bit outdated compared with todays more fine-tuned methods.

2.3 Lemmatizer

Lemmatization is the process of reducing a word to its lemma. A words lemma is the canonical form, dictionary form, or citation form of a lexeme. A lexeme is a unit of lexical meaning that exists regardless of the number of inflectional endings it may have or the number of words it may contain. The lemmas of a dictionary are all lexemes. As with the stemmer, lemmatization can for instance be useful when we want to conflate words to increase the number of possible query matches. A lemmatizer is more fine-tuned but also slower than a stemmer. NLTK contains a lemmatizer, however it only works for English. The lemmatizer is based on WordNet's lexicon. NLTK's internal description of the lemmatizer:

Find a possible base form for the given form, with the given part of speech, by checking WordNet's list of exceptional forms, and by recursively stripping affixes for this part of speech until a form in WordNet is found. Bethard and ... [2001]

To create a lemmatizer we need a word list or lexicon containing as many word forms as possible. We need to group these word forms under lexemes, and after we have grouped these word forms under lexemes we can start to find the lemmas. For every word that we want to know the lemma we search through all the lexeme groups looking for that word. If we find the word in a lexeme group we return the citation form for this group. However depending on application, normally a lemmatizer would also have to disambiguate, for instance by working together with a POS tagger. If not we will end up with all possible lemmas for the input word form. Furthermore there exists Norwegian word lists and lexicon sorted by lexemes, that can be applied for this task, and I will return to where we can

find these later.

2.4 POS Tagger

Part-Of-Speech Tagging is the process of assigning parts of speech to each token, such as 'verb', 'noun', 'conjunction', etc. NLTK has a tagger, however it's for English, it uses the Penn Treebank tag-set. Outside of NLTK we have some options for Norwegian taggers, we have a tagger called Oslo-Bergen Tagger (OBT). OBT works by taking a string of characters as input. Then in the first step it tokenizes the string and multitags the tokens with the help of a lexicon. To multitag a token means to apply all possible tags to the given token, meaning if we get a match in the lexicon, then we add the tag to the token. Then in the next step it applies constraint based rules for disambiguating the tags. In the final step, if enabled, OBT will further disambiguate the tags utilizing a statistical module. We can see OBTs work flow in figur 1.

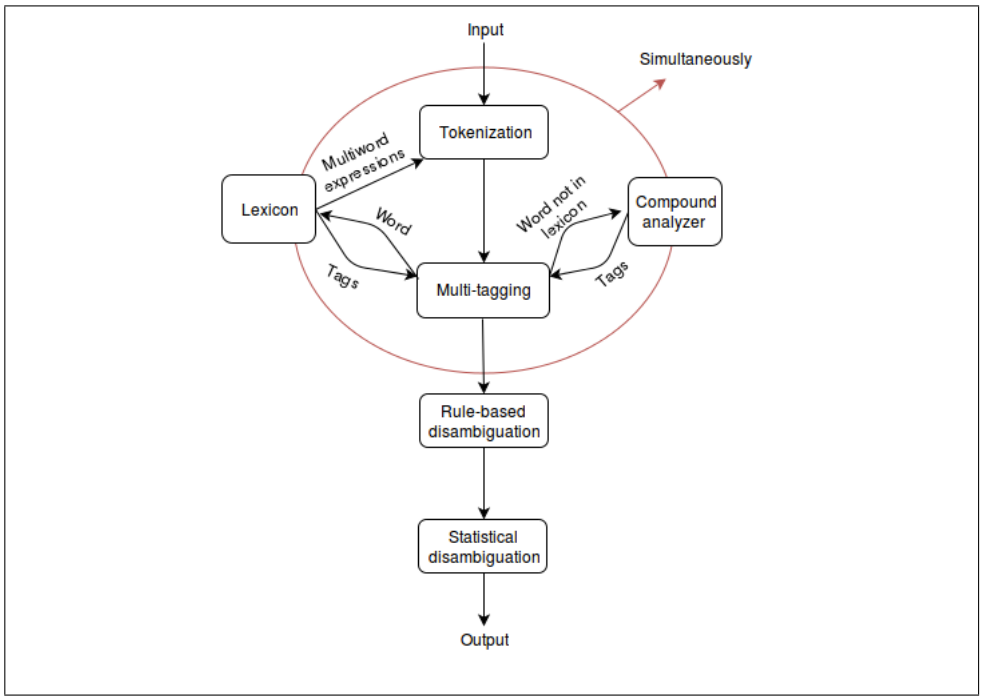

Figure 1: OBT work flow

It is not every day we obtain a totally new addition to the Norwegian language like the word 'app', but creating a new word of already existing words happen daily. For instance if you happen to be in the spotlight, like we saw with former PR-counselor, Sylvi Listhaug, now a Norwegian

politician for the Progress Party and the Minister of Migration and Integration, when she used the word *godhetstyranniet*, "everyone" learned its meaning. Since the spotlight shines on her, she could make this "new" compound a permanent addition to the Norwegian language.

According to Munthe (1972), 10.4 per cent of all words in running text are compounds [Johannessen and Hauglin, 1996, p. 209]

However a word does not have to become a permanent new word to carry meaning. And with context most Norwegian speakers would understand this new word just from it's compounds. If we are to process text written by Norwegians, we should be prepared to handle these compounds. For words not in the lexicon OBT includes a compound analyzer used for creating tags for these new words. The lexicon is also used to look for multiword expressions to store these as one token.

There are other options to OBT, such as UDPipe, Straka and Strakova [2015]. However one problem with all of these are that they require a lot of setup and this makes them less attractive to use. If we would want to include a Norwegian tagger in NLTK it should be something easy to setup, a part of NLTK, instead of a third-party stressful install process.

2.5 Corpus

Corpus - a collection of written texts, especially the entire works of a particular author or a body of writing on a particular subject. To use many of the tools mentioned, one would need a body of text. As mentioned NLTK contains many corpora, next I include a list of the corpora that NLTK showcase on their site.

Corpus	Compiler	Contents
Brown Corpus	Francis, Kucera	15 genres, 1.15M words, tagged, categorized
CESS Treebanks	CLiC-UB	1M words, tagged and parsed (Catalan, Spanish)
Chat-80 Data Files	Pereira & Warren	World Geographic Database
CMU Pronouncing Dictionary	CMU	127k entries
CoNLL 2000 Chunking Data	CoNLL	270k words, tagged and chunked
CoNLL 2002 Named Entity	CoNLL	700k words, pos- and named-entity-tagged
CoNLL 2007 Dependency Treebanks (sel)	CoNLL	150k words, dependency parsed (Basque, Catalan)
Dependency Treebank	Narad	Dependency parsed version of Penn Treebank sample
FrameNet	Fillmore, Baker et al	10k word senses, 170k manually annotated sentences
Floresta Treebank	Diana Santos et al	9k sentences, tagged and parsed (Portuguese)
Gazetteer Lists	Various	Lists of cities and countries
Genesis Corpus	Misc web sources	6 texts, 200k words, 6 languages
Gutenberg (selections)	Hart, Newby, et al	18 texts, 2M words
Indian POS-Tagged Corpus	Kumaran et al	60k words, tagged (Bangla, Hindi, Marathi, Telugu)
MacMorpho Corpus	NILC, USP, Brazil	1M words, tagged (Brazilian Portuguese)
Names Corpus	Kantrowitz, Ross	8k male and female names
NIST 1999 Info Extr (selections)	Garofolo	63k words, newswire, named-entity SGML markup
Nombank	Meyers	115k propositions, 1400 noun frames
NPS Chat Corpus	Forsyth, Martell	10k IM chat posts, POS-tagged, dialogue-act tagged
Open Multilingual WordNet	Bond et al	15 languages, aligned to English WordNet
PP Attachment Corpus	Ratnaparkhi	28k PP's, tagged as noun or verb modifiers
Proposition Bank	Palmer	113k propositions, 3300 verb frames
Question Classification	Li, Roth	6k questions, categorized
Reuters Corpus	Reuters	1.3M words, 10k news documents, categorized
Roget's Thesaurus	Project Gutenberg	200k words, formatted text
RTE Textual Entailment	Dagan et al	8k sentence pairs, categorized
SEMCOR	Rus, Mihalcea	880k words, part-of-speech and sense tagged
Senseval 2 Corpus	Pedersen	600k words, part-of-speech and sense tagged
SentiWordNet	Esuli, Sebastiani	sentiment scores for 145k WordNet synonym sets
Shakespeare texts (selections)	Bosak	8 books in XML format
Stopwords Corpus	Porter et al	2,400 stopwords for 11 languages
Swadesh Corpus	Wiktionary	comparative wordlists in 24 languages
Switchboard Corpus (selections)	LDC	36 phonecalls, transcribed, parsed
Penn Treebank (selections)	LDC	40k words, tagged and parsed
TIMIT Corpus (selections)	NIST/LDC	audio files and transcripts for 16 speakers
VerbNet 2.1	Palmer et al	5k verbs, hierarchically organized, linked to WordNet
Wordlist Corpus	OpenOffice.org et al	960k words and 20k affixes for 8 languages
WordNet 3.0 (English)	Miller, Fellbaum	145k synonym sets

Table 1: Selection of NLTK corpora

There is no Norwegian corpus in NLTK, not even outside the showcase. However there is an Norwegian entry in one of the corpora, in the Stopwords Corpus, there exists a Norwegian stop word list!

2.6 Stop word list

Stop words usually refer to the most common words in a language. The reason they are called stop words is because in many cases we want to stop and remove them from the text, or if not remove at least ignore them. The reason for this is that many classifiers use frequency and/or relationship between words as a base for the machine learning algorithm. This means the most common words have the highest frequency and their relationship to other words might not be as interesting as the less common words appearing before and after the stop word. However to remove stop words is not always correct, some tools specifically avoid removing these stop words to support for instance phrase search.

NLTK contains a corpus called 'Stopwords Corpus' which has lists of stopwords for several language, including Norwegian! The stopword list for Norwegian contains what we would like to find in a stopword list. High-frequency words with low lexical content like 'jeg', 'og', 'deg'. However, it also contains an issue, the list is a mash-up of 'Nynorsk' and 'Bokmål', the two official forms of written Norwegian. This is an issue as some of the words that are stopwords in one written norm are words carrying meaning in the other. The words that have no meaning in the other written norm does not really matter as these word forms will not occur in the specified text. Examples of stopwords 'dykk', 'same', 'skal':

Stopwords\Written norm	Bokmål	Nynorsk
dykk	dive	you
same	Indigenous person in Norway	same (identical)
skal	will	shell

Table 2: Stopwords → Meaning

There is also another issue, some of these stopwords are also homonym words, ambiguous words, 'dykk' can mean 'you' or 'dive' in 'nynorsk'. However what is clear is that we should split the two stopword lists, into two, one for 'Bokmål' and one for 'Nynorsk'.

Chapter 3

3 Sentence tokenizer

Sentence segmentation or sentence tokenization is the task of identifying sentences in running text. In other words it's the task of trying to isolate each 'correct' set of words that is complete in itself. Defining 'correct' in the sense that one wants to find the natural sentence boundaries in the given text, rather than finding every possible sentence within a sentence.

NLTK contains a sentence tokenizer, this sentence tokenizer uses pre-trained models that NLTK supplies. These models are trained for several languages, including Norwegian. The models stem from the research done by Tibor Kiss and Jan Strunk. They developed an algorithm called 'Unsupervised Multilingual Sentence Boundary Detection', an unsupervised classifier, used to train and produce these models.

... a language-independent, unsupervised approach to sentence boundary detection. It is based on the assumption that a large number of ambiguities in the determination of sentence boundaries can be eliminated once abbreviations have been identified. ... Quantitatively, abbreviations are a major source of ambiguities in sentence boundary detection since they often constitute up to 30% of the possible candidates for sentence boundaries in running text [Kiss and Strunk, 2006, p. 485-486]

The point being that the end of a sentence does not have a specific pattern, but abbreviations do follow a specific pattern. And most of the ambiguities come from the fact that an abbreviation should always end in a final period, as sentences should also. By characterizing the abbreviations we can eliminate a lot of possible candidates for sentence boundaries.

Kiss and Strunk assume that abbreviations are collocations of the truncated word and the following period, by this they derive that we can use methods for detecting collocations to detect abbreviations. They offer a characterization of abbreviations in terms of three major properties:

[...] an abbreviation looks like a very tight collocation in that the abbreviated word preceding the period and the period itself form a close bond. [...]

[...] abbreviations have the tendency to be rather short. [...] the likelihood of being an abbreviation declines if candidates become longer [...]

[...] the occurrence of word-internal periods contained in many abbreviations. [Kiss and Strunk, 2006, p. 487]

I will not go more into detail, however we get the main idea, first train over large amounts of text to classify abbreviations, then apply the trained model to not split sentences on period if it's a period contained in a found classified abbreviation. The model used in NLTK, or 'punkt model' as it's called, stems as mentioned from this work by Kiss and Strunk. Each model must be trained for each language and the material used to train the Norwegian punkt model is displayed in table 3 below.

Source	Contents	Size of training corpus(in tokens)
Center for Humanities Information Technologies, Bergen	Bergens Tidende	479,000

Table 3: The material used to train the Norwegian punkt model

3.1 How to evaluate the tokenizer

To evaluate the sentence tokenizer I am going to find the f-score for two different measurements, sentence 'segment score' and 'split score'. The sentence split score is a measure of how proficient the tokenizer is at finding the correct positions to cut sentences. Sentence segment score, on the other hand, is a measure of how proficient the tokenizer is at finding the correct complete sentences. Simplified in figure 2. I will also compare NLTK to a couple of other tokenizers.

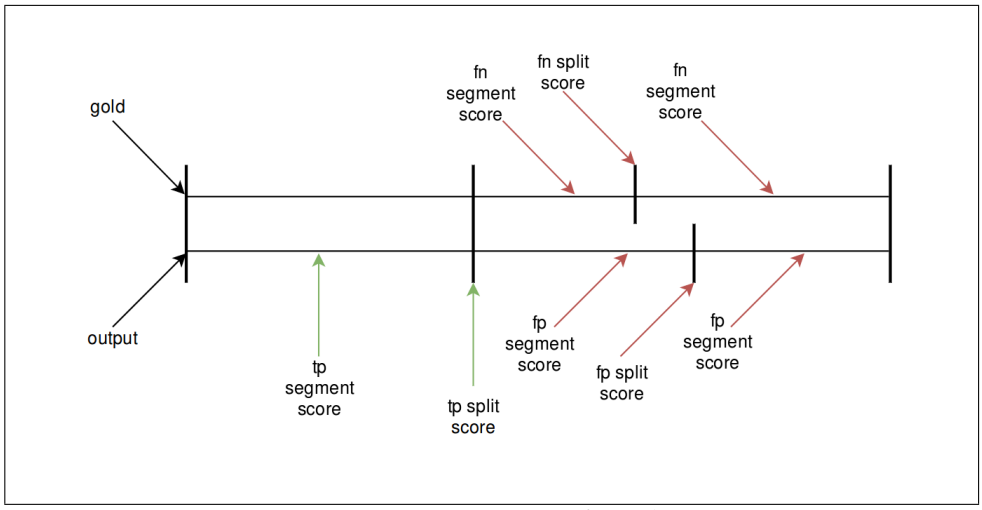

Figure 2: Two measurements for tokenization

For gold I will be using Norwegian Dependency Treebank (NDT). It is stored in CoNLL format which makes it easy to count and collect all the sentences. (CoNLL format is a file where each line represents a single word with a series of tab-separated fields, I'll return to this later.) However we do not want to run our tokenizer on CoNLL format, we want to run it on running text. How to achieve this?

One way would be to detokenize the CoNLL format back to 'running well formed' text. That way our gold consist of known running text. Ideally I would have wanted a gold consisting of pure natural language, but detokenized CoNLL will suffice, as I am interested in well formed text anyway.

Since we are also doing failure analysis and later trying to fix the most common errors, we are going to need to split the material. When choosing how to split the material in NDT, there exists a split by Petter Hohle:

Our split of the Norwegian Dependency Treebank splits the data into three data sets, viz., training, development and testing. 80% of the data is used in the training, 10% in the development and the final 10% resides in the held-out test data set, which is a commonly used data set split for machine learning in NLP. [Hohle, 2016a, p. 36]

However this split does not consider that the material consists of documents, meaning it splits documents in the middle of paragraphs. This might be insignificant, but when considering for instance the "split score", it would be in my opinion ideal to make a more "clean" cut by keeping the documents intact, i.e. not creating any artificial splits. NDT has neatly sorted all the text in different categories, this could also play a part in the split. There is no real need for a development set, as there is no "machine learning" involved. It could have been useful, but using only the train set for development will work just fine. Therefore I will only create a 80-20 split, instead of a 80-10-10.

Untouched NDT contains 20045 sentences and 311277 tokens divided unevenly over 59 documents, separated in 232 files:

Table 4: Documents contained in NDT

Source	Amout of documents
Aftenposten (ap)	15
Bergens Tidende (bt)	6
Blogs (blogg-bm)	3
Dagbladet	13
Klassekampen (kk)	8
Government reports (nou)	4
Sunnmørsposten (sp-bm)	3
Parliament transcripts (st)	5
Verdens Gang (vg)	2

If we use the following split (table 5), we will keep the documents intact and include all categories in test and training while still achieving something close to 80-20 (table 6):

Table 5: Split of NDT

Source	Trainset, **doc ID**	Testset, **doc ID**
ap	1-13	14-15
blogg-bm	2-3	1
bt	1-4, 6	5
db	1-6,8-13	14
kk	2-8	1
nou	1,3-4	2
sp-bm	1-2	3
st	1-3	5
vg	1	2

	Trainset	Testset
Sentences	15868 (79.162%)	4177 (20.838%)
Tokens	249786 (80.246%)	61491 (19.754%)

Table 6: NDT split overview

Before we examine the baseline score for the tokenizer, there is another issue I want to address. Whether to remove headlines or not, as headlines differ from running text (there is no delimiter). I ended up on choosing to remove headlines, to remove the noise it creates when considering 'running well formed text'. Removing the headlines shrinks the material from 20045 to 19012 sentences.

To obtain the baseline I will use the NLTK sentence tokenizer on the defined training material. See table 7.

Table 7: Normal NLTK sentence tokenization, with Norwegian settings - Baseline

	Sentence split score	Sentence segment score
TP	14217	13374
FP	94	938
FN	839	1683
recall	0.94427	0.88822
precision	0.99343	0.93446
fscore	0.96823	0.91076

3.2 Failure analysis and patch creation

When doing failure analysis and patch creation for the NLTK sentence tokenizer I used a simple empirical bootstrapping approach. I simply looked at a small amount of errors for patterns, then tried to fix the errors. Then I repeated this process of fixing and looking until I was satisfied with the patch. Using this approach I discovered five patterns that when patched should provide us with a better result.

The first and biggest error I discovered was the colon, ':'. Is it more often correct to split a sentence at ':' or not? Even though the model is trained for Norwegian, it is trained only to recognize abbreviations. The parts of the tokenizer that consider what is a valid delimiter, lays outside the model. Therefore it does not split at colon, as this is possibly the right choice in some countries. However for Norwegian we might want to split at colon, and it would seem like that is the right choice from this sample space of Norwegian language. However we should not split at every colon, but every colon followed by white-space, as we do not want to split for instance infix colon, '13:37'.

The second error discovered was interpose sentences, these posed an issue as how to separate what was a ”real” from a ”fake” sentence boundary. Such as here, the tokenizer would create two sentences when there should be one:

Gold: 'Siden Hagen (intil videre?) fortsatt kommer til å [...]'
NLTK: ∗ 'Siden Hagen (intil videre?)' | 'fortsatt kommer til å [...]'

However thanks to our well defined language we can employ the rule of ”each real sentence should begin with a capital letter”, meaning if a sentence is output without a capital letter it should belong to the previous sentence.

The third error discovered stem from when 'a certain non-alphabetic symbol following the delimiter' makes it so that the sentence is not split. The symbol causing the flow of errors in this case is guillemets, ». Such as this sentence:

Gold: '[...] regninger og sånn?»' | 'Det var liksom ikke [...]'
NLTK: ∗'[...] regninger og sånn?» Det var liksom ikke [...]'

Guillemets are language specific for Norwegian and should be patched. This means we should split sentences ending with a delimiter followed by », to make this safe one should also make sure it's followed by white-space and upper case.

The tokenizer also seems to have a problem with understanding when to split sentences at multiple delimiters such as '...', by default it does not split. If we split on sentences that have multiple delimiters followed by upper case we remove some of the errors.

Gold: '[...] meg ned i bilen i går kveld ...' | 'Hvordan resten [...]'
NLTK: ∗'[...] meg ned i bilen i går kveld ... Hvordan resten [...]'

The last error-pattern is that sometimes a sentence output contains only non-alphabetic characters. Then it should be part of the next sentence or previous sentence output, however as there is no real pattern to define where it belongs we will say it belongs to the next sentence. Since we use ”itemization” in running text, it's more likely to belong to the next sentence.

Gold: '3. Alternativ til statskirkeordningen og [...]'
NLTK: ∗'3.' | 'Alternativ til statskirkeordningen og [...]'

Itemized the error patterns found are:

- Split at colon
- Split after delimiter and guillemet
- Split after multiple delimiters
- Stitch interpose sentences
- Stitch non-alphabetic sentences to a sentence

If we apply these changes to our baseline output from NLTK we get these scores:

Table 8: Sentence split score, Baseline vs Patch

	NLTK	NLTK+patch
TP	14217	15008
FP	94	77
FN	839	48
recall	0.94427	0.99681
precision	0.99343	0.9949
fscore	0.96823	0.99585

	NLTK	NLTK+patch
TP	13374	14893
FP	938	193
FN	1683	164
recall	0.88822	0.98911
precision	0.93446	0.98721
fscore	0.91076	0.98816

Table 9: Sentence segment score, NLTK vs Patch

As we can see in table 23 & 24, the resulting sentence tokenized material is a lot closer to the gold text. We can check the failure rate to see how much it has improved, to obtain the failure rate I will look at the relationship between the difference in the f-score.

Split failure rate:

$$(1 - 0.99585)/(1 - 0.96823) = \underline{\underline{0.13063}}$$

Segment failure rate:

$$(1 - 0.98816)/(1 - 0.91076) = \underline{\underline{0.13268}}$$

We can see that the failure rate has decreased significantly for both segment and split score, which is the preferred outcome! With more words, what we observe is that before the patch the tokenizer got 0.96823 of the splits correct, while after the patch it got 0.99585. We invert the numbers to obtain the rate of failure instead of the success rate of the f-score. Giving us

a failure rate of 0.03177 before and 0.00415 after patch. Looking at the these numbers we can see that the tokenizer fail almost 8 times less after patch. By doing the same with the segment score we can notice that to is almost 8 times less likely to fail!

3.3 Detailed failure analysis of patch

When doing this detailed failure analysis I will look at the remaining errors and classify them. I will classify if I created the error with the patch or if it remains from the original tokenizer. Then I will classify what type of errors they are. There are 77 false positive in the split score. Out of these errors I created 51 of them while 26 are left from the original tokenizer. There are 48 sentences that should not have been split at colon, that I split e.g:

Output: ∗'Til eneste mulige destinasjon:' | 'Djibouti.'
Gold: 'Til eneste mulige destinasjon: Djibouti.'

There are 3 sentences that should not have been split at '...', that I split due to it beeing followed by a capital letter:

Output: ∗'Min lillesøster ...' | 'Aktivisten, sangeren, mammaen [...]'
Gold: 'Min lillesøster ... Aktivisten, sangeren, mammaen [...]'

The errors still left are the errors I did not create. There are 18 errors that stems from abbreviations unknown to the model such as:

Output: ∗'[...] under ledelse av dr. juris.' | 'Carsten Smith ble [...]'
Gold: '[...] under ledelse av dr. juris. Carsten Smith ble [...]'

There are 6 interposed sentence errors that are not fixed, sometimes due to that the first charactar following the interposed sentences is not a lowercase character:

Output: ∗'[...] og i alle fall ikke onepiece (!' | '), (motstanderne [...]'
Gold: '[...] og i alle fall ikke onepiece (!), (motstanderne [...]'

Last there are 2 special errors, where two sentences are combined to one sentence:

Output: ∗'[...] banner yet wave.' | 'O'er the land of the free [...]'
Gold: '[...] banner yet wave. O'er the land of the free [...]'

It is not written correctly, as it's poetry, it should not be a delimiter, there is usually a newline. The tokenizer can certainly not understand that this should be one sentence. The 48 false negative are of course sentences that are not split where they should have been split.

There are 18 errors that I created, 2 are non-alphabetic stitch errors such as:

Output: ∗'1991. Det er såre profetisk:'
Gold: '1991.' | 'Det er såre profetisk:'

The remaining 16 are sentences I stitched together that should not have been stitched due to the fact that the sentence began with a small letter:

Output: ∗'Justin, hvor går du når du vil ha brød? spør Clough.'
Gold: 'Justin, hvor går du når du vil ha brød?' | 'spør Clough.'

The rest are errors created by the tokenizer, there are still 10 sentences that end with delimiter and guillemets that are not split. Mainly due to not being followed by a capital letter. There are 7 errors where the line is not split at multiple delimiters such as '...', also mainly due to not being followed by capital letter:

Output: ∗'kan dere jo bare tenke dere til ... #tragikomisk'
Gold: 'kan dere jo bare tenke dere til ...' | '#tragikomisk'

There are 3 times a sentence ends with an abbreviation and the tokenizer does not split:

Output: ∗'[...] med fokus på vegdata etc. Tenk Cappelens [...]'
Gold: '[...] med fokus på vegdata etc.' | 'Tenk Cappelens [...]'

I found one sentence missing delimiter, meaning it will always fail here:

Output: ∗'[...] det er lærer man skulle vært;) Bønna har hatt [...]'
Gold: '[...] det er lærer man skulle vært;)' | 'Bønna har hatt [...]'

For the errors left created by the tokenizer, there are some special cases. For instance since the tokenizer is trained for abbreviations, it does not split a sentence if it ends with a single letter followed by period, as this is treated as an abbreviation. In this material a sentences ends with a single letter 7 times, 5 of those is due to the sentences ending with 'i':

Output: ∗'[...] vi kunne gå inn i. Tonje Gjevjon, styreleder [...]'
Gold: '[...] vi kunne gå inn i.' | 'Tonje Gjevjon, styreleder [...]'

The tokenizer will not split a sentence if it ends with the word 'bås.' or 'fenomen.' followed by most words. However it does not happen with all words, if followed by for instance 'Det' it will split the sentence. This happend with 2 sentences in the material.

Output: ∗'[...] tilgivelse som fenomen. Kirken prediker [...]'
Gold: '[...] tilgivelse som fenomen.' | 'Kirken prediker [...]'

This might have something to do with the model being over trained. I ran a list with 700 000 words in it to see how many words where affected, I found 6 words acting in the described way:

'hi', 'bås', 'fenomen', 'sifre', 'pol', 'startstreken'

3.4 Comparison

Now that we have looked at all the errors left, let us compare how NLTK holds up compared to other sentence tokenizers. The University of Oslo has a site called "Language Analysis Portal" - lap.clarino.uio.no, this site allows us to upload sentences and tokenize them. A description of the site from the site:

The UiO Language Analysis Portal (LAP) enables non-technical end users to employ state-of-the-art tools for natural language processing (NLP) at the click of a few buttons; ... The work is carried out at the University of Oslo (UiO) as a joint effort by the Language Technology Group (LTG) at the Department of Informatics and of the Department of Research Computing at the University Center for Information Technology. [UiO, 2016, web]

This site has the same NLTK sentence tokenizer, but it also has a couple of others. 'Rule-Based Sentence Segmenter' (CIS) [Nagel, 2010, web] and 'Oslo Bergen Tagger' (OBT) [UiO, 1998, web]. I will not go into detail about how these work, we will simply use them as something to compare to our baseline. But I will however explain how to receive the wanted results from LAP.

First I will upload the detokenized training material to the site, then run the different sentence tokenizers on it. Then we have to word tokenize and export the material to CoNLL format, before we are allowed to download the result of the sentence tokenizer. To word tokenize the documents LAP offers two options, 'NLTK' and 'REPP'. I used 'NLTK' word tokenizer as the other option 'REPP' changes the number of characters in the output material vs the input material, for instance converting '...' to a single triple dot character i.e converting three characters to one. The only thing left to do is export to CoNLL format and analyse the output.

Table 10: Sentence split score, baseline vs online options vs patch

	NLTK	LAP-NLTK	Rule-Based	OBT	NLTK+patch
TP	14217	14181	12847	14623	15008
FP	94	130	35	171	74
FN	839	875	2209	433	48
recall	0.94427	0.94188	0.85328	0.97124	0.99681
precision	0.99343	0.99092	0.99728	0.98844	0.9949
fscore	0.96823	0.96578	0.91968	0.97977	0.99585

	NLTK	LAP-NLTK	Rule-based	OBT	NLTK+patch
TP	13374	13312	11016	14212	14893
FP	938	1000	1867	583	193
FN	1683	1745	4041	845	164
recall	0.88822	0.88411	0.73162	0.94388	0.98911
precision	0.93446	0.93013	0.85508	0.96059	0.98721
fscore	0.91076	0.90653	0.78854	0.95216	0.98816

Table 11: Sentence segment score, baseline vs online options vs patch

First thing to notice is that the online LAP version of NLTK does not give the same result as the local NLTK. I can find the new errors, but I do not know why they appear. When using the online NLTK version it splits the sentences differently, take for instance this gold sentence:

Utvalget skal legge fram sin utredning innen utgangen av 2005.”

After the online sentence tokenizer is done, the ’ ” ’ is included in the next sentence, this does not happen when using the local NLTK sentence tokenizer.

If we look at the scores not considering the patched NLTK. We can see that the rule-based has the best precision and lowest fscore, while OBT has the best recall and fscore when only looking at the split score. In the segment score OBT has the best recall, precision and fscore, while rule-based has the lowest recall, precision and fscore.

Something worth to mention is that OBT was used as a helping tool when creating NDT, however they are not equal as we can see from the scores. For instance OBT uses multi-word expressions something that is not included in NDT. But there is a problem with OBT, when extracting the results from LAP, OBT contains extra character.

For instance in OBT, '&' is written as '&amp', and not just '&' like in gold. I had to deal with this problem before I could acquire a relevant score, by aligning OBT to gold, as we score by strict character measurements and any extra characters will skew everything. To fix the problem we have to find the excess information and deal with it, align the output from OBT with the gold. After we have aligned them we will be able to measure OBT against gold.

3.5 Extra

It might be fun to see how the same patch works if applied to the other sentence tokenizer, I will include tables for it here.

Table 12: Extra sentence split score, online options vs online options + patch

	Rule-Based	Rule-Based+patch	OBT	OBT+patch
TP	12847	13560	14623	14745
FP	35	94	171	183
FN	2209	1496	433	311
recall	0.85328	0.90064	0.97124	0.97934
precision	0.99728	0.99312	0.98844	0.98774
fscore	0.91968	0.94462	0.97976	0.98352

	Rule-based	Rule-Based+patch	OBT	OBT+patch
TP	11016	12288	14212	14415
FP	1867	1367	583	1134
FN	4041	2769	845	642
recall	0.73162	0.8161	0.94388	0.95736
precision	0.85508	0.89989	0.96059	0.92707
fscore	0.78855	0.85595	0.95216	0.94197

Table 13: Extra sentence segment score, online options vs online options + patch

We can see that the patch overall seem to reduce the FN in both split and segment score. The patch also seems to help the Rule-Based achieve a better result, however OBT does not agree with the patch, we obtain a slightly better fscore in the split score, however a slightly worse fscore in the segments. That being said, NLTK+patch still has the best score.

3.6 Conclusion

To finish it off we still have to look at the results from running it on the held-out test set. First I'll run our baseline NLTK on the test set and then compare the results of using the baseline NLTK+patch.

Table 14: Sentence split score, Baseline vs Patch - testset

	NLTK	NLTK+patch
TP	3786	3943
FP	24	22
FN	168	11
recall	0.95751	0.997218
precision	0.9937	0.99445
fscore	0.97527	0.99583

	NLTK	NLTK+patch
TP	3609	3914
FP	202	52
FN	346	41
recall	0.91252	0.98963
precision	0.947	0.98689
fscore	0.92944	0.98826

Table 15: Sentence segment score, NLTK vs Patch - testset

As we did earlier we can look at the relationship between the difference in the f-score to see if we acquire comparable results to earlier.

Split failure rate:

$$(1 - 0.99583)/(1 - 0.97527) = \underline{\underline{0.16862}}$$

Segment failure rate:

$$(1 - 0.98826)/(1 - 0.92944) = \underline{\underline{0.16638}}$$

It is close, not quite as good, but very close to the results we achieved with the training material!

Chapter 4

4 Word tokenizer

Word tokenization or possibly token tokenization is the task of identifying all individual tokens in running text. In other words it's the task of trying to isolate each correct single distinct meaningful element of writing. Defining 'correct' in the sense that one wants to find the natural token boundaries in the given text, rather than finding every possible token (e.g. not every morpheme).

The word tokenizer that is contained in the NLTK package is the 'Penn Treebank Tokenizer'. It is created for the English language, however the language dependent part of the tokenizer revolves around contractions and will not interfere with our output. The Penn Treebank tokenizer use regular expressions to find tokens of the type seen in the Penn Treebank. For every token it finds it inserts a white-space in between the tokens and when it has processed the whole string (sentence) it uses the built in python function '.split()', this means that the string is split at every white-space, creating a list of tokens from the sentences.

And thats it, the tokens does not undergo any form from case folding. Collocations become multiple tokens. Removes most punctuation, except 'infix punctuation', from the token. It will not remove '.' from a token unless it's the last token in the sentence, due to abbreviations.

> Det var (svart-hvitt) TV o.l. i New York.
> ⌜Det⌝⌜var⌝⌜(⌝⌜svart-hvitt⌜)⌝⌜TV⌝⌜o.l.⌝⌜i⌝⌜New⌝⌜York⌝⌜.⌝

One thing that is worth to notice is that the tokenizer does some weird choices when considering quotation marks:

> Det var ”mange” 'romvesen'.
> ⌜Det⌝⌜var⌝⌜“⌝⌜mange⌝⌜”⌝⌜'romvesen⌝⌜'⌝⌜.⌝

We can see that it converts the single character ” to two ‘ characters. We can also see that it keeps the first ' attached to the word 'romvesen' but separates the other one from the word.

NLTK's word tokenizer will always try to sentence tokenize first, so we have to turn on Norwegian settings if we want it to recognize the abbreviations.

4.1 How to evaluate the tokenizer

To evaluate the word tokenizer I am going to find the f-score for the same two measurements I used on sentence evaluation, word 'segment score' and 'split score'. The word split score is a measure of how proficient the tokenizer is at finding the correct positions to cut between tokens. Word segment score, is then the measure of how proficient the tokenizer is at finding the correct complete tokens. Simplified in figure 2. I will also compare NLTK to a couple of other tokenizers.

I will be using the same gold material, NDT. Since it's stored in CoNLL format it means every line contains a token, the tokenizers job is then to match these tokens. I will of-course still detokenize the material such that we feed the tokenizer 'running well formed' text. To acquire the baseline I used the normal NLTK sentences tokenizer + word tokenizer. See table 16.

Table 16: Normal NLTK word tokenization, with Norwegian settings - Baseline

	Word split score	Word segment score
TP	243710	242206
FP	159	1664
FN	1682	3201
recall	0.99315	0.98696
precision	0.99935	0.99318
fscore	0.99624	0.99006

4.2 Failure analysis

There are not many errors left after the NLTK word tokenization, however some notable errors can be fixed. All the errors, both FP and FN, seem to revolve around punctuation. In the 148 false positive there are 71 errors where '.' has been removed from the word when it should not have been:

NLTK: ∗ ⌜Fr.p⌝⌜.⌝⌜trapper⌝⌜opp⌝ [...]
Gold: ⌜Fr.p.⌝⌜trapper⌝⌜opp⌝ [...]

There are 46 errors that spawn due to apostrophe being removed from the connecting word:

NLTK: ∗ [...] ⌜troikaen⌝⌜og⌝⌜Hellas⌝⌜'⌝⌜internasjonale⌝ [...]
Gold: [...] ⌜troikaen⌝⌜og⌝⌜Hellas'⌝⌜internasjonale⌝ [...]

In 16 cases it's due to the 'at' sign, the tokenizer splits emails at @:

NLTK: ∗ [...] ⌜eiliv⌝⌜@⌝⌜db.no⌝⌜eller⌝ [...]
Gold: [...] ⌜eiliv@db.no⌝⌜eller⌝ [...]

Twelve of the errors come from splitting words connected with ':':

NLTK: ∗ [...] ⌜dette⌝⌜≪GH⌝⌜:⌝⌜WT≫⌝⌜en⌝ [...]
Gold:[... ⌜dette⌝⌜≪ ⌝⌜GH:WT⌝⌜≫⌝⌜en⌝ [...]

One should note that this does not happen if it's a number on each side of the colon. There are 3 special cases when a word start with a punctuation like '#':

NLTK: ∗ [...] ⌜tenke⌝⌜dere⌝⌜til⌝⌜...⌝⌜#⌝⌜tragikomisk≫⌝
Gold:[...] ⌜tenke⌝⌜dere⌝⌜til⌝⌜...⌝⌜#tragikomisk≫⌝

In the 1697 false negative I will not count every error, however I will do some machine counting and receive some rough estimates. It would seem that more than 1400 of the errors stem from guillemets, ≫ and ≪:

NLTK: ∗ [...] ⌜led⌝⌜≪opprørerne≫⌝⌜nederlag⌝ [...]
Gold: [...] ⌜led⌝⌜≪⌝⌜opprørerne⌝⌜≫⌝⌜nederlag⌝ [...]

NLTK does not split on '/', meaning we receive errors where we should have split, 71 in this case:

NLTK: ∗ [...] ⌜og⌝⌜statlige/kommunale⌝⌜bevilgninger⌝⌜.⌝
Gold: [...] ⌜og⌝⌜statlige⌝⌜/⌝⌜kommunale⌝⌜bevilgninger⌝⌜.⌝

Then the remaining errors are errors such as the ones we saw earlier when introducing the NLTK word tokenizer:

NLTK: ∗ [...] ⌜ et⌝⌜'jeg⌝⌜'⌝⌜der⌝ [...]
Gold: [...] ⌜et⌝⌜'⌝⌜jeg⌝⌜'⌝⌜der⌝ [...]

i.e. not removing the first apostrophe connected to a word.

4.3 Norwegian version

As for sentences tokenization we also want Norwegian word tokenization. As I mentioned the NLTK 'Penn Treebank Tokenizer' is created for English, but the language dependent parts does not interfere with Norwegian. That means we can try to create a patch for this tokenizer so that it fits better for Norwegian, rather that creating a brand new tokenizer.

However to to patch a word tokenizer is harder than patching a sentences tokenizer. The reason for this is that when we processes the list with tokens output from the word tokenizer we have lost all information regarding where white-space used to be. Take for instance the example above containing ⌜GH:WT⌝, after the tokenizer is done it leaves the tokens

as ⌜≪GH⌝ ⌜:⌝ ⌜WT≫⌝. Since the tokens now are separated and we have lost all information about where white-space used to be, we can not fix this problem post-hoc.

As white-space is key to word tokenization post-patching will not be as efficient as for instance making changes to current tokenizer or recreation of the tokenizer. However as the results from the baseline are quite good, there is no need to change the original or create a new tokenizer, I will still just post-patch the errors fixable at this point.

Some possible fixes can be to split at '/', remove guillements from connected word and stitch together emails.

- Split at slash
- Disconnect guillement from connected word
- Stitch emails

I mentioned earlier that we have some Norwegian language dependent structures for word tokenization such as large number and collocations. In NDT collocations are written as multiple tokens, this is understandable. As to recognizing collocations would require third-party help such as a lexicon over all know Norwegian collocations. However what is harder to agree upon is that NDT also stores large numbers as multiple tokens, this would not require any form of outside help, here it would suffice just to keep all digits following each other as one token. I will therefore also create a patch that disagrees with gold and keeps all large numbers as one token and list the score:

Table 17: Word split score, Baseline vs Patch

	NLTK	NLTK+patch	NLTK+largnum-patch
TP	243710	245290	245252
FP	159	173	173
FN	1682	102	140
recall	0.99315	0.99958	0.99943
precision	0.99935	0.9993	0.9993
fscore	0.99624	0.99944	0.99936

	NLTK	NLTK+patch	NLTK+largenum-patch
TP	242206	245033	244961
FP	1663	431	465
FN	3201	374	446
recall	0.98696	0.99848	0.99818
precision	0.99318	0.99824	0.99811
fscore	0.99006	0.99836	0.99814

Table 18: Word segment score, NLTK vs Patch

As we can see in table 17 & 18, the resulting word tokenized material is a lot closer to the gold text. We can check the failure rate to see how much it has improved, to obtain the failure rate I compare the f-score:

Split failure rate:

$$(1 - 0.99944)/(1 - 0.99624) = \underline{\underline{0.14894}}$$

Segment failure rate:

$$(1 - 0.99836)/(1 - 0.99006) = \underline{\underline{0.16499}}$$

We can see that the failure rate has decreased significantly for both segment and split score, it's now between 6-7 times less likely to fail. When fixing large numbers as one token we can see that we receive a lower score than just the patch. This is a direct conflict with the gold material so it will obviously trigger errors.

4.4 Comparison

Now that we have improved NLTK's word tokenizer, let's compare how NLTK holds up compared to other word tokenizers. I will still be using the LAP site for the other options. The site has the same NLTK word tokenizer, but it also has a couple of others. 'REPP' [Dridan and Oepen, 2012] and 'OBT' [UiO, 1998, web]. I will not go into details about how these work; we will again simply use them as something to compare with.

To acquire the wanted results from LAP, it's the same process as earlier: upload - sentence tokenize - word tokenize - export to CoNLL format. We have to sentence tokenize to do word tokenization, meaning we have to choose a sentence tokenizer on the site before word tokenizing, I will use the NLTK sentence tokenizer from the site, even tough we do not obtain the exact same sentences splits as the baseline it will be close enough. I also used the 'Norwegian:NDT' settings for REPP.

Table 19: Word split score, baseline vs online options

	NLTK	LAP-NLTK	REPP	OBT	NLTK+patch
TP	243710	244497	245236	242919	245290
FP	159	4528	131	46	173
FN	1682	909	170	2473	102
recall	0.99315	0.9963	0.99931	0.98992	0.99958
precision	0.99935	0.98182	0.99947	0.99981	0.9993
fscore	0.99624	0.989	0.99939	0.99484	0.99944

	NLTK	LAP-NLTK	REPP	OBT	NLTK+patch
TP	242206	241241	244972	241071	245033
FP	1663	7785	396	1895	431
FN	3201	4166	435	4322	374
recall	0.98696	0.98302	0.99823	0.98239	0.99848
precision	0.99318	0.96874	0.99837	0.9922	0.99824
fscore	0.99006	0.97583	0.99831	0.98727	0.99836

Table 20: Word segment score, baseline vs online options

Even though OBT played a helping hand in creating NDT, NDT did not give it much of a helping hand here. Most of the errors in OBT stems from the use of multi-word tokens. Not considering the patch we can see that REPP has the overall best score!

4.5 Conclusion

And of course to finish it off we have to look at the results from running it on the held-out test set. First I'll run our baseline NLTK on the test set and then compare the results of using the baseline NLTK+patch and NLTK+largnum-patch.

Table 21: Word split score, Baseline vs Patch - testset

	NLTK	NLTK+patch	NLTK+largnum-patch
TP	60159	60197	60185
FP	27	29	29
FN	47	9	21
recall	0.99922	0.99985	0.99965
precision	0.99955	0.99952	0.99952
fscore	0.99939	0.99968	0.99958

	NLTK	NLTK+patch	NLTK+largenum-patch
TP	60107	60162	60139
FP	80	65	76
FN	104	49	72
recall	0.99827	0.99919	0.9988
precision	0.99867	0.99892	0.99874
fscore	0.99847	0.99905	0.99877

Table 22: Word segment score, NLTK vs Patch - testset

Like before we can inspect the relationship between the difference in the f-score to see if we receive comparable results to earlier.

Split failure rate:

$$(1 - 0.99968)/(1 - 0.99939) = \underline{\underline{0.52459}}$$

Segment failure rate:

$$(1 - 0.99905)/(1 - 0.99847) = \underline{\underline{0.620915}}$$

These results are not on the same level when compared to the training material. I fixed some errors, but there are not enough errors in the test set to receive the same amount of benefit from the patch as we got in the training set.

Chapter 5

5 Integration of tokenizer patches

As we have seen there are some issues with NLTK. Luckily NLTK is an open source, community-driven project. This means that it's possible to integrate the changes we want into this project. NLTK like many other open source projects use Git [2005] for the project. Git or GitHub is a development platform that provides one with the opportunity to contribute for open source projects alongside millions of other developers. I created an issue on NLTK's GitHub to see if they had an interest in the patch.

5.1 Alternative ways of integrating

There are many ways of integrating the patches into NLTK. One way would be to add a couple of lines to the original code:

Original code:

```
def sent_tokenize(text, language='english'):
    tokenizer =
        load('tokenizers/punkt/{0}.pickle'.format(language))
    return tokenizer.tokenize(text)


def word_tokenize(text, language='english'):
    return [token for sent in sent_tokenize(text, language)
            for token in _treebank_word_tokenize(sent)]
```

Edited code:

```
import patch

def sent_tokenize(text, language='english'):
    if language='norwegian':
        tokenizer =
            load('tokenizers/punkt/{0}.pickle'.format(language))
        return patch.sent_patch(tokenizer.tokenize(text))
    else:
        tokenizer =
            load('tokenizers/punkt/{0}.pickle'.format(language))
        return tokenizer.tokenize(text)


def word_tokenize(text, language='english'):
    if language='norwegian':
        return [token for sent in sent_tokenize(text, language)
                for token in
                    patch.word_patch(_treebank_word_tokenize(sent))]
    else:
        return [token for sent in sent_tokenize(text, language)
                for token in _treebank_word_tokenize(sent)]
```

Another approach is to add a single file to the folder containing the patch and a copy of the original tokenizer code, this however would result in the users having to use a different function, something like nltk.nor_sent_tok('xyz') for norwegian tokenization instead of just the standard nltk.sent_tokenize('xyz', 'norwegian'). It also seems redundant. A third option is by changing the main tokenize function. In my opinion the first option would be the ideal solution, it is simple and easy, work done.

By this I mean that we have mapped a set of error patterns, created a patch to see if they improve the results, now we just have to apply it. This would be the easiest solution if we were the only one to use the patch. However this might not be the "correct" way to approach the problem when we want to integrate it into a project. I learned this after trying to suggest this approach to the problem for the NLTK community.

5.2 Integrating sentence tokenizer patch

When suggesting my "easy" way to fix the problem I learned that it's not desirable to have a post-hoc string manipulation patch. As such the community wanted a different approach to solve the problem. They wanted to find the core of the problem and fix it there, or to retrain the model on NDT. Lets follow those paths and see where those lead.

Retrain the model - if we want to retrain the model, we have to first remember what we train the model to find. The model is trained to find abbreviations. Are the errors fixed by the patch, errors related to abbreviations, no. The path of retraining the model on NDT is a dead end.

Going to the core of the problem and fix it there - this requires us to study how the tokenizer work inside NLTK. As we can see in the code above the sentence tokenizer uses a function 'load' on the specified pickle, for instance 'norwegian.pickle'. The 'load' functions comes from the nltk.data, this function is used to load a variety of different files contained in NLTK, when 'load' recognizes that the input is a pickle it will return 'pickle.load(open(x.pickle))' (a python native function). This means the variable 'tokenizer' above is now a python object with the values it obtained when training on the specified language. When inspecting this object at runtime we can see that it calls the functions stored in the file 'nltk.punkt', this means 'tokenize' in 'tokenizer.tokenize' from above is a function stored in 'nltk.punkt'. This means that to fix the problem at it's core we have to change 'nltk.punkt'. And this might have some merit to it. There is a class in the punkt tokenizer called PunktLanguageVars, it says:

```
"Stores variables, mostly regular expressions, which may be
language-dependent for correct application of the algorithm.
An extension of this class may modify it's properties to suit
a language other than English; an instance can then be passed
as an argument to PunktSentenceTokenizer and PunktTrainer
constructors."
```

Kiss and Strunk [2001]

The purpose of this class seems to be, to provide languages the possibility for language dependent variables. For instance there is a variable in this class called 'sentence_end_chars' this variable stores information on valid sentence delimiters, by default those are '.', '!', '?', if we add ':' to this variable we fix the biggest error pattern found for the sentence tokenizer, the colon. However we can not change this part of the punkt tokenizer as it would affect every language. But what we can do is to create a subclass of PunktLanguageVars, for instance NorwegianLanguageVars where we add colon to 'sentence_end_chars'. We can just add this to the punkt tokenizer:

```
class NorwegianLanguageVars(PunktLanguageVars):

    sent_end_chars = ('.', '?', '!', ':')
```

Now as the current Norwegian pickle still calls PunktLanguageVars. The correct way would be to retrain the model with NorwegianLanguageVars listed in the pickle at training time instead of PunktLanguageVars.

However we can just ”hack” the current pickle as we can still use the same model. We just need to change the pickle to call NorwegianLanguageVars instead of PunktLanguageVars. There are several ways to ”hack” the pickle. One way is to unpickle the pickle and change the variable 'object._lange_vars=PunktLanguageVars' to 'object._lange_vars=NorwegianLanguageVars' inside the loaded object and save it as a new pickle. Another way is just to open the pickle in a text editor and change the line containing 'PunktLanguageVars' to 'NorwegianLanguageVars'. Now if we add the new pickle to the pickle folder we are already getting a better result.

Table 23: Sentence split score, Baseline vs Patch vs Hacked pickle

	NLTK	NLTK+patch	NLTK+hack
TP	14217	15008	14931
FP	94	77	143
FN	839	48	125
recall	0.94427	0.99681	0.9917
precision	0.99343	0.9949	0.99051
fscore	0.96823	0.99585	0.99111

	NLTK	NLTK+patch	NLTK+hack
TP	13374	14893	14682
FP	938	193	393
FN	1683	164	375
recall	0.88822	0.98911	0.97509
precision	0.93446	0.98721	0.97393
fscore	0.91076	0.98816	0.97451

Table 24: Sentence segment score, NLTK vs Patch vs Hacked pickle

However it's still not at the level as the patch and we can not implement all of the fixes for the error patters this way. For instance when we stitch interposed sentences, this is nothing we can fix by adding values to the LanguageVars.

5.3 Integrating word tokenizer patch

When wanting to integrate my word patch I discovered a new issue. If we recall the most prominent error in the word tokenization for Norwegian, it was the guillemets. However that was in the NLTK version I was working with, 'NLTK 3.2.2', these guillemets had been fixed in 'NLTK 3.2.3' released: May 2017. This was achieved by adding the guillemets to the PennTreeBank regexs. This was an early idea I had, to add the regexs, however as we possibly wanted a patch for large numbers I disregarded the idea of adding the regexes and created a post-hoc patch. (Large numbers containing white-space, would not be possible to add in the regexes. As the regexes is for creating white space for splitting at those in the end.) If we test the new tokenizer on the training set we acquire these results:

Table 25: Word split score, Baseline vs Patch vs New-NLTK

	NLTK-3.2.2	3.2.2+patch	NLTK-3.2.3
TP	243710	245290	245222
FP	159	173	159
FN	1682	102	170
recall	0.99315	0.99958	0.99931
precision	0.99935	0.9993	0.99935
fscore	0.99624	0.99944	0.99933

	NLTK-3.2.2	3.2.2+patch	NLTK-3.2.3
TP	242206	245033	244938
FP	1663	431	444
FN	3201	374	469
recall	0.98696	0.99848	0.99809
precision	0.99318	0.99824	0.99819
fscore	0.99006	0.99836	0.99814

Table 26: Word segment score, NLTK vs Patch vs New-NLTK

By comparing the patch with the new-NLTK version we receive these numbers:

Split failure rate:

$$(1 - 0.99944)/(1 - 0.99933) = \underline{\underline{0.83582}}$$

Segment failure rate:

$$(1 - 0.99836)/(1 - 0.99814) = \underline{\underline{0.88172}}$$

The word patch now does next to nothing, and does not have the numbers to support further support.

Chapter 6

6 Norwegian text resources

NB has an on-line repository and this repository contains many open and available lexicon and corpora of Norwegian text. This is fantastic news for anyone wanting to get their hands on much needed Norwegian data for NLP. As there is no Norwegian corpus in NLTK and we wanted to integrate a Norwegian corpus, we can acquire our Norwegian corpus to integrate from here. To obtain a more a clearer view on what NB has to offer I will list some of the available resources in NB:

Name	Compiler	Contents
N-grams - NBdigital	NB	(n=1-3) from books and newspapers at NB
N-grams for Norwegian Bokmål	NB	(n=1-6) (based on NST news text)
Tagged texts - NBdigital	NB	4808 morphologically tagged texts - Bokmål
Norwegian Wordnet Bokmål	Kaldera språkteknologi	50.000 synonym sets (synsets)
Norwegian Wordnet Nynorsk	Kaldera språkteknologi	50.000 synonym sets (synsets)
Norwegian parliamentary debates	Rosén, Victoria. UiB	Treebank of transcriptions of debates
Norwegian Newspaper Corpus Annotated	Uni Research AS	35 692 210 tokens, annotated and classified
Norwegian Newspaper Corpus Nynorsk	Uni Research AS	large monitor corpus of contemporary
Norwegian Newspaper Corpus Bokmål	Uni Research AS	large monitor corpus of contemporary
Text material from Forskning.no	CLARINO	Data set containing texts from forskning.no
NorGram NDT, LFG, Norwegian Nynorsk	UiB	Norwegian Dependency Treebank
NorGram NDT, LFG, Norwegian Bokmål	UiB	Norwegian Dependency Treebank
NorGramBank Newspaper text from LBK	UiB	Treebank, syntactically annotated corpus
NorGramBank Non-fiction text (LBK)	UiB	Treebank, syntactically annotated corpus
NorGramBank television subtitles (LBK)	UiB	Treebank, syntactically annotated corpus
NorGrambank children's fiction	Rosén, Victoria. UiB	106434 sentences, 1043260 words, 76 docs
Norwegian Dependency Treebank	NB	Norwegian Dependency Treebank - Bokmål
INESS NorGramBank collection	Rosén, Victoria. UiB	A parsebank of Norwegian (LFG)
Annotations of non-fiction text	Rosén, Victoria. UiB	Syntactically annotated from Nynorskkorpuse
Annotations of fiction text	Rosén, Victoria. UiB	Syntactically annotated from Nynorskkorpuse
Norwegian Acquis Communautaire	NB	5414 docs from Acquis Communautaire
Freely available texts from NBDigital	NB	26344 books, 10756 different authors
Norsk Ordbank in Norwegian Bokmål	UiO & Språkrådet	Fullform lexical database, Bokmål
SCARRIE Lexical Resource	Rosén, Smedt & Nordgård	Fullform lexical resource, Bokmål
NST Lexical database	NB	Fullform lexical database, Bokmål
Wordlists from Språkrådet	Språkrådet	various types of word lists
Translation memories	NB	English to Norwegian Bokmål and Nynorsk
Hyphenations from the NB	NB	a list of hyphenated words in Norwegian

Here there are many corpora that we could integrate into NLTK. However one stands out, we can see that NB has a corpus called 'Norwegian Dependency Treebank' (NDT), this has been called the gold corpus for Norwegian. It is a syntactic treebank for Norwegian, the great thing about NDT is that it is new (2014) and it is manually syntactically and morphologically annotated. The annotation process of the treebank was supported by the OBT and then manually corrected by annotators. It was developed at NB in collaboration with the UiO. It is the first publicly available treebank for Norwegian. A treebank is a parsed text corpus that annotates syntactic or semantic sentence structure.

6.1 Norwegian Corpus

While working with Norwegian NLP we need Norwegian text, and that is why we want to integrate a Norwegian corpus into NLTK. And the corpus to add is 'Norwegian Dependency Treebank' NDT [2014]. The corpus comes in CoNLL format. There are many different CoNLL formats since CoNLL is a different shared task each year. However the format used in dependency parsing comes from CoNLL-X format further revised version CoNLL-U Format, CoNLL [2009]. The CoNLL-U Format has three types of lines:

1. Word lines containing the annotation of a word/token in 10 fields separated by single tab characters; see below.
2. Blank lines marking sentence boundaries.
3. Comment lines starting with hash (#).

Sentences consist of one or more word lines, and word lines contain the following fields:

1. ID: Word index, integer starting at 1 for each new sentence; may be a range for multiword tokens; may be a decimal number for empty nodes.
2. FORM: Word form or punctuation symbol.
3. LEMMA: Lemma or stem of word form.
4. UPOSTAG: Universal part-of-speech tag.
5. XPOSTAG: Language-specific part-of-speech tag; underscore if not available.
6. FEATS: List of morphological features from the universal feature inventory or from a defined language-specific extension; underscore if not available.
7. HEAD: Head of the current word, which is either a value of ID or zero (0).
8. DEPREL: Universal dependency relation to the HEAD (root iff HEAD = 0) or a defined language-specific subtype of one.
9. DEPS: Enhanced dependency graph in the form of a list of head-deprel pairs.
10. MISC: Any other annotation.

The fields must additionally meet the following constraints:

- Fields must not be empty.
- Fields other than FORM and LEMMA must not contain space characters.
- Underscore (_) is used to denote unspecified values in all fields except ID. Note that no format-level distinction is made for the rare cases where the FORM or LEMMA is the literal underscore - processing in such cases is application-dependent.

NDT follows almost every rule set by the CoNLL-U Format. However there are some differences. In NDT column 9 and 10 are left as unspecified/'_'. Also column 4 and 5 contain identical information in NDT, this means we do not have to load both these columns when reading the corpus. Another difference is that NDT does not use the label 'root' if HEAD = 0, it has a different DEPREL name for each type of root. From the corpus documentation:

"The head of the sentence will also have a function called the root function . For finite verbs, the root function will be FINV." [Kinn et al., 2014, p. 23]

This is understandable as every root can easily be identified by the head number. An example of a sentences in NDT, displayed as presented in the corpus in CoNLL format:

1	At	at	sbu	sbu	_	4	SBU
2	jeg	jeg	pron	pron	ent|pers|hum|nom|1	4	SUBJ
3	liksom	liksom	adv	adv	_	4	ADV
4	er	være	verb	verb	pres	0	FINV
5	den	den	pron	pron	mask|fem|ent|pers|3	4	SPRED
6	som	som	sbu	sbu	_	7	SBU
7	tar	ta	verb	verb	pres	5	ATR
8	på	på	prep	prep	_	7	ADV
9	seg	seg	pron	pron	akk|refl	8	PUTFYLL
10	ansvaret	ansvar	subst	subst	appell|nøyt|be|ent	7	DOBJ
11	med	med	prep	prep	_	10	ATR
12	å	å	inf-merke	inf-merke	_	11	PUTFYLL
13	fortelle	fortelle	verb	verb	inf	12	INFV
14	de	de	det	det	dem|fl	16	DET
15	store	stor	adj	adj	fl|pos	16	ATR
16	tingene	ting	subst	subst	appell|mask|be|fl	13	DOBJ
17	.	$.	clb	clb	<punkt>	4	IP

NDT is licensed under the Creative Commons, this means that we are free to share and adapt NDT. Under the following terms: 'Attribution' and 'No additional restrictions'.

The content of the `Creative_Commons-BY (CC-BY):`

You are free to:

Share - copy and redistribute the material in any medium or format
Adapt - remix, transform, and build upon the material
for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution - You must give appropriate credit,
provide a link to the license, and indicate if changes were made.
You may do so in any reasonable manner, but not
in any way that suggests the licensor endorses you or your use.

No additional restrictions -
You may not apply legal terms or technological measures that,
legally restrict others from doing anything the license permits.

Figure 3: Creative_Commons-BY (CC-BY) License

Commons [2001]

If the previous attribution to NB and UiO was not enough let us give credit to the individuals behind the creation of NDT,

Per Erik Solberg,
Språkbanken, The National Library of Norway

Arne Skjærholt and Lilja Øvrelid,
Department of Informatics, University of Oslo

Kristin Hagen and Janne Bondi Johannessen,
Department of Linguistics and Scandinavian Studies,
University of Oslo

There exists an optimized tag-set for NDT created by Petter Hohle. It would be optimal to make these optimized tags available with the corpus in NLTK. When integrating a corpus to NLTK we have to create a corpus reader or use one of the corpus readers available in NLTK. NLTK has a lot of corpus readers, they have one for CoNLL format and one for dependency treebanks. And as NDT is both in CoNLL format and a dependency treebank one of these might fit our corpus, if not we will have to create one.

Before we find out if they fit NDT, we have to know what we expect from a corpus reader. The obvious answer is, we want a reader that gives all the information available in the corpus to the user of the corpus. After inspecting the CoNLL and dependency treebank corpus reader I found out that they do "work" for the corpus, delivering some information to the user, such as words and parsed sentences. However they do not provide the full content of the corpus to the user, as they where designed for slightly different corpora. This means we should create a corpus reader for NDT in NLTK that does bring the whole corpus to the user of it.

6.2 Corpus Reader

When creating a corpus reader in NLTK there exists a main CorpusReader class with an api, we have to make a subclass of the main CorpusReader class and fill in the parts expected from the api and add any wanted extra functionality. This makes the corpus with reader callable by; 'from nltk.corpus import ndt'. When reading the information into the corpus reader we have to plan how we want to store the information. However in this case it's quite easy, as all the information in NDT is contained in 8 columns, we can represent those as 7 ordered lists (as column 4 and 5 are identical). This means that the first time we use the corpus, it will have a small load time, less than one second, to load all the data in NDT into 7 lists. There will also be an eight column. This eight column will contain the optimized tags created by Petter Hohle. The tags are a product of his master

thesis, Hohle [2016a] and can be found at his GitHub, Hohle [2016b].

However there is a minor issue we have to solve before adding this eight column. He has not sorted the files containing the tags alphabetically when considering the filenames. This because as we can recall he wanted to split NDT into 3 parts, train, dev and test, but also consider categories when choosing where to create the split. And all files of the same category begins with the same letter. Now when reading the 232 files contained in NDT we choose an order to read it, usually alphabetically. However we can not do this if we want to match the order of his tags. Hohle's tags are sorted in 3 files instead for instance 3 folders, by this I mean he has glued all files that ended up as train, dev or test together as large files. Instead of creating a more complex corpus reader to match this invisible order of files I have extracted the optimized tags from his 3 files and sorted them to alphabetically match the original 232 files, and added it as an eight column.

To access these columns we can use these functions:

```
ndt.token_indexes(fileids=None):
ndt.words(fileids=None):
ndt.lemmas(fileids=None):
ndt.pos(fileids=None):
ndt.morp_feats(fileids=None):
ndt.head_indexes(fileids=None):
ndt.dep_rels(fileids=None):
ndt.opt_pos(fileids=None):
```

These are the 8 columns that creates NDT in NLTK. The rows are fictional as the lists are ordered, this means that element x in each column corresponds to each other. All function calls returns all elements from start to end, or from specified file ids. When integrating a corpus reader into NLTK one has to create a subclass of NLTK Corpus Reader. This class has some functions that are standard to every corpus reader in NLTK, these have to be implemented. This is a list of the standard functions that have been implemented:

```
ndt.raw()
#returns all the content raw

ndt.readme()
#returns the corpus README

ndt.words(fileids=None)
#returns n number of words from start to end,
# or from specified file ids.

ndt.sents(fileids=None)
#returns n number of sentences from start to end,
# or from specified file ids.

ndt.tagged_words(tagset=None, fileids=None)
#returns n number of tagged words from start to end,
# or from specified file ids.
#If tagset is set to 'opt' it will return the optimized tags.

ndt.tagged_sents(tagset=None, fileids=None)
#returns n number of tagged words from start to end,
# or from specified file ids.
#If tagset is set to 'opt' it will return the optimized tags.

ndt.parsed_sents(tagset=None, fileids=None)
#returns n number of parsed sentences from start to end,
# or from specified file ids.
#If tagset is set to 'opt' it will use the optimized tags.
```

The function 'ndt.parsed_sents' gives dependency parsed sentences, using NLTK's dependency parse reader. This reader has 3 types of input it can take 3, 7 or 10 cells of information. And we have all the needed information to use the 10 cell reader, however there is one thing holding us back. As mentioned NDT does not use the label 'root' in the DEPREL column when HEAD = 0, and this reader needs a defined label for the root DEPREL, like, 'root' or 'top'. We could rewrite the corpus to match this reader, however I will not do this as it would change the information available in the corpus. We could also create a larger parsed_sent method that would rewrite every time we wanted to read a parsed sentence, but this also felt inefficient and unnecessary. The 7 cell parser has the same problem, however we can use the 3 cell parser as it only needs word_form, tag and head. Using the 3 cell parser we receive dependency graph objects that can for instance present trees like this:

```
>>> from nltk.corpus import ndt
>>> ndt.sents()[111]
['Kofi', 'Annan', 'har', 'tatt', 'i', 'bruk', ',', 'og',
    'fastholder', 'stadig', ',', 'det', 'utvidede', 'fredsbegrep',
    ',', 'som', 'inkluderer', 'alle', 'dem', 'som', 'lider', '.']
>>> ndt.parsed_sents()[111]
<DependencyGraph with 23 nodes>
>>> ndt.parsed_sents()[111].tree().pprint()
(har
  (Kofi Annan)
  (tatt (i bruk))
  ,
  (fastholder
    og
    stadig
    ,
    (fredsbegrep
      det
      utvidede
      ,
      (inkluderer som (dem alle (lider som)))))
  .)
```

From this point on I would say it is really up to the user of the corpus. However to make the information a tad bit more available I added some more helper functions. Where the main helper function is a function for extracting information from the corpus.

Next follows a description of this extract function, we can call it with ndt.extract:

```
extract(sent=True, word=True, tags=False, lma=False, morph=False,
    head=False, dep=False, opt=False, fileids=None)
```

With this function we can acquire what we want from the corpus:

```
# Returnes n number of words or sentences, from start to end,
# sent=
# True, get sentences | False, get only words
# word=
# True, get word form | False, not include word form
# tags=
# True, get pos tags | False, not include pos tag
# lma=
# True, get lemmas   | False, not include lemmas
# morph=
# True, get morph feat | False, not include morphological feature
# head=
# True, get head index | False, not include head index
# dep=
# True, get dep rel  | False, not include dependency relation
# opt=
# True, get opt tag  | False, not include optimized tag
# fileids=
# None, get all files | Single fileid or list of fileids
```

Example of using the extract method:

```
>>> from nltk.corpus import ndt
>>> ndt.extract(tags=True, opt=True)
[[('Lam', 'subst', 'subst|appell|ub'), ('og', 'konj', 'konj'),
    ('piggvar', 'subst', 'subst|appell|ub'), ('pa', 'prep',
    'prep'), ('bryllupsmenyen', 'subst', 'subst|appell|be'), ('|',
    'clb', 'clb')], [('Kamskjell', 'subst', 'subst|appell|ub'),
    (',', '<komma>', '<komma>'), ('piggvar', 'subst',
    'subst|appell|ub'), ('og', 'konj', 'konj'), ('lammefilet',
    'subst', 'subst|appell|ub'), ('sto', 'verb', 'verb|fin'),
    ('pa', 'prep', 'prep'), ('menyen', 'subst', 'subst|appell|be'),
    ('under', 'prep', 'prep'), ('den', 'det', 'det'), ('kongelige',
    'adj', 'adj|pos'), ('gallamiddagen', 'subst',
    'subst|appell|be'), ('.', 'clb', 'clb')]] ...
>>> ndt.extract(word=False, lma=True, opt=True)
[[('lam', 'subst|appell|ub'), ('og', 'konj'), ('piggvar',
    'subst|appell|ub'), ('pa', 'prep'), ('bryllupsmeny',
    'subst|appell|be'), ('$|', 'clb')], [('kamskjell',
    'subst|appell|ub'), ('$,', '<komma>'), ('piggvar',
    'subst|appell|ub'), ('og', 'konj'), ('lammefilet',
    'subst|appell|ub'), ('sta', 'verb|fin'), ('pa', 'prep'),
    ('meny', 'subst|appell|be'), ('under', 'prep'), ('den', 'det'),
    ('kongelig', 'adj|pos'), ('gallamiddag', 'subst|appell|be'),
    ('$.', 'clb')]] ...
>>> ndt.extract(word=False, sent=False, morph=True, opt=True)
[('subst|appell|ub', 'appell|noyt|ub|ent'), ('konj', '<ikke-clb>'),
    ('subst|appell|ub', 'appell|mask|ub|ent'), ('prep', '_'),
    ('subst|appell|be', 'appell|mask|be|ent|samset'), ('clb',
    '<overskrift>'), ('subst|appell|ub', 'appell|noyt|ub|fl'),
    ('<komma>', '_'), ('subst|appell|ub', 'appell|mask|ub|ent'),
    ('konj', '<ikke-clb>'), ('subst|appell|ub',
    'appell|mask|ub|ent'), ('verb|fin', 'pret'), ('prep', '_'),
    ('subst|appell|be', 'appell|mask|be|ent'), ('prep', '_'),
    ('det', 'dem|mask|ent'), ('adj|pos', 'be|ent|pos'),
    ('subst|appell|be', 'appell|mask|be|ent'), ('clb', '<punkt>'),
    ('konj', 'clb')] ...
```

In the first extract I ask for sentences with word_forms,
default pos tags and optimized pos tags.

In the second extract I ask for sentences,
no word_forms, only lemma and optmized tags.

In the third extract I ask for words,
no word_forms, only morphological features and optimized tags.

6.3 Lexicon

From the list of resources at NB we can see that we have several lexicon to choose from. I've looked into two of them 'Norsk OrdBank' and 'SCARRIE'. 'Norsk OrdBank' is the same lexicon used to develop the OBT tagger we talked about earlier. 'SCARRIE' was originally a lexical resource developed for automatic proofreading of Norwegian Bokmål.

Table 27: Basic stats of the lexicons

	Norsk OrdBank	SCARRIE
unique word forms	344139	287682
intersection	285437	285437
difference	58702	2245
words	713299	344806

When trying to identify what type of words the 'difference' contains I found that the 2245 words in 'SCARRIE' not in 'Norsk OrdBank' contains mostly written errors and Swedish words. This might be linked to the fact that 'SCARRIE' was developed for automatic proofreading. However it also contains words with hyphens which is somewhat relevant if we where ever to encounter these hyphenates. See table 28. In the 58702 words in 'Norsk OrdBank' not in 'SCARRIE' it's mainly a lot more inflicted words, however these words may be over-generated as 'Norsk OrdBank' include their own inflectional paradigm. It also contains multiword expressions. See table 29.

Table 28: Unique to SCARRIE

Written errors	'romm'	'dettede'	'unnbedende'
Swedish words	'kysst'	'pommes'	'stjernblomer'
-er	'CD-er'	'pm-er'	'pH-er'

inflected words	'est'	'este'	'ene'
Words with white-space	'old boys-klassen'	'ad absurdum'	'i dag'

Table 29: Unique to Norsk OrdBank

Just from the basic looks of it 'Norsk OrdBank' would be the best pick, however it's copyrighted. It is ILN/UiO and Språkrådet that owns the

copyright:

Norsk Ordbank - Bokmål Licence: proprietary

Figure 4: Norsk Ordbank Licence

As these word forms are owned, SCARRIE seems like your best choice, of these two, as a lexicon. Since SCARRIE uses creative commons, you must only give appropriate credit.

Chapter 7

7 Conclusion and resources

I have investigated what support NLTK has for Norwegian NLP, I have evaluated, and tried to improve this support. For sentence and word tokenization I have evaluated and compared NLTK's tokenizers against other tokenizers. I have improve NLTK's sentence and word tokenization and then tried to integrate the improvements into NLTK's open source project. I looked at lexical resources for Norwegian in NB, where we looked at a couple of lexicon and a corpus. I have tried to integrate a Norwegian corpus with corpus reader into NLTK.

7.1 Resources made and their availability

I have created patches for NLTK's sentence and word tokenizer. And I have created the NLTK corpus reader for NDT including optimized tags. All of these are available at my github.com Bjerke-Lindstrøm [2017]. I have of course created many tools for my own use on the way, for instance in making the evaluation and comparison of the tokenizers possible, such as the CoNLL detokenizer, OBT aligner, etc. but I see no reason to publish these.

References

NLTK. Natural language toolkit. *nltk.org/, (visited 14.08.17)*, 2001.

Peter Michael Stahl and Peter Ljunglof. Source code for nltk.stem.snowball. *nltk.org/_modules/nltk/stem/snowball.html, (visited 14.08.17)*, 2001.

Steven Bethard and Steven Bird ... Source code for nltk.corpus.reader.wordnet. *nltk.org/_modules/nltk/corpus/reader/wordnet.html, (visited 14.08.17)*, 2001.

Janne Bondi Johannessen and Helge Hauglin. An automatic analysis of norwegian compounds. *Papers from the 16th Scandinavian Conference of Linguistics*, pages 209–220, 1996.

Milan Straka and Jana Strakova. Udpipe. *ufal.mff.cuni.cz/udpipe, (visited 14.08.17)*, 2015.

Tibor Kiss and Jan Strunk. Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4):485–487, 2006.

Petter Hohle. Optimizing a pos tag set for norwegian dependency parsing. *Master's Thesis, Deparment of informatics, University of Oslo*, page 36, 2016a.

UiO. Language analysis portal. *lap.clarino.uio.no, (visited 14.08.17)*, 2016.

Sebastian Nagel. Rule-based sentence segmenter. *cis.uni-muenchen.de/ wastl/misc/, (no longer working)*, 2010.

UiB UiO. Oslo-bergen-taggern. *tekstlab.uio.no/obt-ny/, (visited 14.08.17)*, 1998.

Rebecca Dridan and Stephan Oepen. Tokenization: returning to a long solved problem a survey, contrastive experiment, recommendations, and toolkit. *ACL '12 Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2*, pages 378–382, 2012.

Git. Github. *github.com/, (visited 14.08.17)*, 2005.

Tibor Kiss and Jan Strunk. Source code for nltk.tokenize.punkt. *www.nltk.org/_modules/nltk/tokenize/punkt.html, (visited 14.08.17)*, 2001.

NDT. Norwegian dependency treebank. *nb.no/sprakbanken/show?serial=sbr-10, (visited 14.08.17)*, 2014.

CoNLL. Conll-u. *universaldependencies.org/format.html, (visited 14.08.17)*, 2009.

Kari Kinn, Per Erik Solberg, and Pål Kristian Eriksen. Ndt guidelines for morphological and syntactic annotation. *nb.no/sbfil/dok/20140314_guidelines_ndt_english.pdf, (visited 14.08.17)*, 2014.

Creative Commons. Attribution 2.0 generic (cc by 2.0). *creativecommons.org/licenses/by/2.0/, (visited 14.08.17)*, 2001.

Petter Hohle. ndt-tools. *github.com/petterhh/ndt-tools/tree/master/dataset/bokmaal, (visited 14.08.17)*, 2016b.

Bo Bjerke-Lindstrøm. Resources made. *github.com/toolsNLTK, (visited 14.08.17)*, 2017.