Using Word Embeddings to Determine Concepts of Values In Insurance Claim Spreadsheets

Vemund Justnes

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Institute for Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Using Word Embeddings to Determine Concepts of Values In Insurance Claim Spreadsheets

Vemund Justnes

Using Word Embeddings to
Determine Concepts of Values In
Insurance Claim Spreadsheets

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Many business decisions are based on data which exist in spreadsheets. In the insurance domain, domain experts use spreadsheet tools for different analytical tasks, such as underwriting, and as a tool for exchanging data with a third party. In business-to-business insurance, a claimant or surveyor may send an overview of damages as a spreadsheet. Data in these spreadsheets are of interest when handling a claim, and can also include meta-information, such as parties, locations, events, etc. For instance, the claim handler will look in the spreadsheet and find the costs related to a damage, and then map those costs to specific insurance coverage types. In order to automate the mapping from a cost to an insurance coverage, both the description of the cost and the total sum need to be extracted from the spreadsheet. Automatically mining these spreadsheets is challenging as they have no standardized structure; i.e., they are semi-structured tables. I explored the idea of classifying the concept of textual values found in spreadsheets as a potential first step for mining data in semi-structured tables in the insurance domain. In order to use supervised learning, I created a data set consisting of 119,963 cell values gathered from spreadsheets linked to actual Norwegian property claims. I tried four different methods for the classification task: rule-based (RB), multinomial naïve Bayes (MNB), k-nearest neighbors (k-NN), and a method where I represented each concept by the mean embedding of all its samples (MI; short for multi-index). The RB used the raw text as input, the MNB used bag-of-character n-grams to featurize the text, and both the k-NN and the MI represented the text by sentence embeddings (where both the composition method and the distributed method was used). Measuring the accuracy of the models by F_1-score (harmonic mean between correct classification and missed classifications), the RB has an accuracy of 52.90%, MNB has an accuracy of 79.11%, k-NN has an accuracy of 87.09%, while the MI has an accuracy of 64.78%. Although k-NN achieved the highest accuracy, it took nearly three hours to evaluate 23,998 test samples, whereas the MNB evaluated them in just under five seconds. Interestingly, I found that the MI (which with an inefficient implementation evaluated in roughly 20 seconds) reached an accuracy on par with the RB with just ten samples – indicating that the MI is a practical method for streamlining annotation of text found in spreadsheets. To conclude, the approach I present is not suitable to extract information that affect the claim handling, as it relies too much on the formatting of values, such as monetary values. However, the approach makes meta-information more accessible, and therefore, the approach can be used to extract information, such as organisation names and locations. Although the k-NN is inefficient at classifying, the method can be used to extract meta-information, as such an approach can be handled as a background process that does not affect the claim handling. Finally, I suggest that some future work is done to improve the data set I created, investigate whether word embeddings fine-tuned to the insurance domain improves the accuracy, and investigate a method for making more efficient classification using the k-NN.

Ackowledgements

I would like to thank my supervisor at the University of Oslo, Peter Csaba Ølveczky, for having provided useful feedback on academic writing, and my leader at Protector Forsikring, Leonard Bijl, who enabled me to write a master's thesis related to the insurance domain. I would also like to thank all of my co-workers that I have met at Protector Forsikring, as they have both motivated me and provided useful insight to how the insurance industry operates. Last but not least, I would like to thank my family and the people that I have met during my studies.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

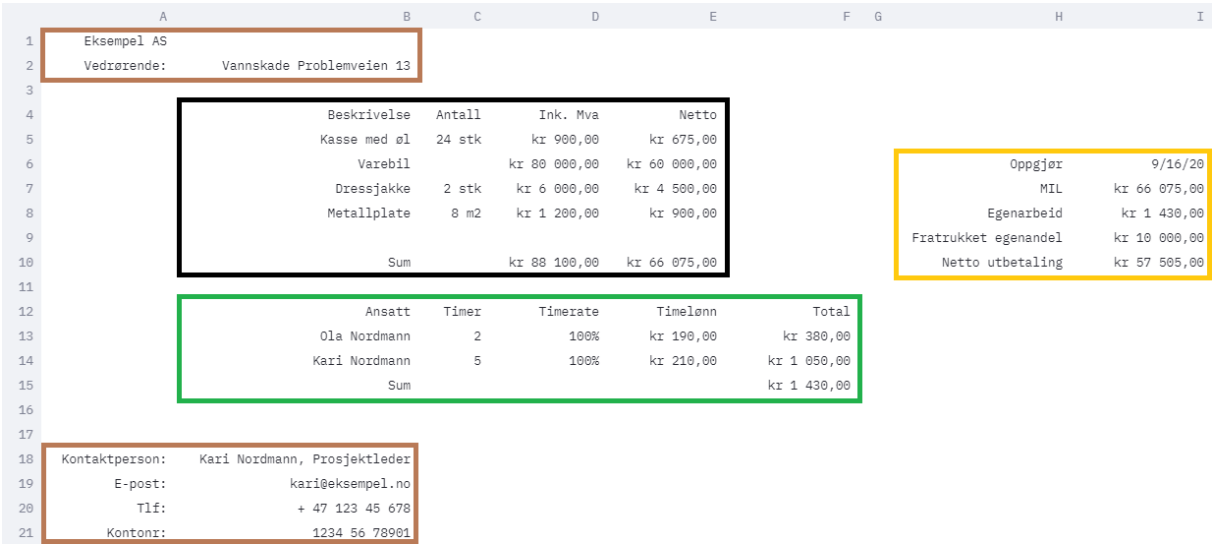
The amount of data in the real world is massive, but using them for machine learning to automate mundane tasks can be challenging. Data are often categorized as either structured data or unstructured data. Structured data are precise facts regarding a specific concept, e.g., product price or customer name, that are organized in a standardized format (e.g., table format using relational databases) where they are accessible and have integrity w.r.t. what concepts the facts describe. Unstructured data have no standardized structure and exist in a variety of formats, e.g., HTML, PDF and plain text, where the precise facts are hidden in text written in natural language.

Common estimates categorize 80% of data as unstructured.[1] However, formats such as HTML and PDFs are structured to some extent, i.e., titles, contents, tables, paragraphs, etc., are sectioned and often contain a topic. For instance, the first paragraph is about data (topic), where facts such as "structured data" and "unstructured data" are used to describe categories of data (concept). The main issue with these types of formats is that the facts are more broadly grouped by a topic where natural language is used to describe relations between the facts (in contrast to having a standardized structure denote the relations). These files can, therefore, be regarded as semi-structured data.

In the insurance domain, spreadsheets are widely used by domain experts in day-to-day business operations, which rely on data in these spreadsheets to make important business decisions such as analyzing risk of a potential customer. Spreadsheets are also used as a means to share data between third parties, such as the insured, claimant or a damage surveyor. In business-to-business (B2B) insurance, a property damage claim is more complex than a claim for an insured home, as the daily operation of the business can be affected by the damage (business interruption) and has additional costs related to the property damage. All costs must be surveyed, documented and presented to a claim handler. In some claims, the cost overview is sent by the claimant or surveyor as a spreadsheet, which contains information about costs and other information (e.g., contact person). As the spreadsheets are created by different individuals, they do not have a standardized structure.

Spreadsheets are semi-structured as it is known that a cell represents a fact, but the concept of the fact is unknown when seen in an automation perspective. To provide an example of how the structure of a spreadsheet related to an insurance claim may be, the spreadsheet in Figure 1.1 was manually created. Although a single spreadsheet can be seen as a single table, Figure 1.1 illustrates that several tables can be embedded in a single spreadsheet. The colored borders indicate different types of topics; information such as contact person, insured, location (brown), damaged inventory (black), work hours spent on damage (green), and a temporary setup of potential payout based on costs (yellow). The table marked as yellow is how a settlement offer could look like in the example. In this example, the costs related to "Kasse med øl", "Varebil", "Dressjakke" and "Metallplate" have been mapped to the *machine, inventory and movables* in-

[1] https://www.altexsoft.com/blog/structured-unstructured-data/



Eksempel AS								
Vedrørende:	Vannskade Problemveien 13							
	Beskrivelse	Antall	Ink. Mva	Netto				
	Kasse med øl	24 stk	kr 900,00	kr 675,00				
	Varebil		kr 80 000,00	kr 60 000,00				
	Dressjakke	2 stk	kr 6 000,00	kr 4 500,00				
	Metallplate	8 m2	kr 1 200,00	kr 900,00				
	Sum		kr 88 100,00	kr 66 075,00				
	Ansatt	Timer	Timerate	Timelønn	Total			
	Ola Nordmann	2	100%	kr 190,00	kr 380,00			
	Kari Nordmann	5	100%	kr 210,00	kr 1 050,00			
	Sum				kr 1 430,00			
Kontaktperson:	Kari Nordmann, Prosjektleder							
E-post:	kari@eksempel.no							
Tlf:	+ 47 123 45 678							
Kontonr:	1234 56 78901							

Oppgjør	9/16/20
MIL	kr 66 075,00
Egenarbeid	kr 1 430,00
Fratrukket egenandel	kr 10 000,00
Netto utbetaling	kr 57 505,00

Figure 1.1: Manually created example of a spreadsheet illustrating different types of information.

surance coverage (MIL), and the costs related to work hours spent are mapped to the *own work* insurance coverage (Egenarbeid), and a deductible of 10,000 NOK is subtracted from the costs.

A simplified explanation of the claim handling, is that: costs are mapped to an insurance coverage, the claim handler use the insurance policy to determine what risks the insured is covered against and to find the deductible for such an insurance event, and then any missing documentation or information about the insurance event is requested from the insured. As the lists of costs can be quite long, and as the claim handler has to determine the insurance coverage, using machine learning to classify the insurance coverage based on a cost description could streamline the claim handling process. For instance, "Kasse med øl" $\mapsto$ MIL, while the text "Shutdown of business due to damage" would map to the *business interruption* insurance coverage. Before we can use machine learning to map the costs in Figure 1.1 to the appropriate insurance coverage, we would need a method for extracting the damage information from the spreadsheets.

In order to automatically extract the data from spreadsheets, a method for detecting separate tables is needed, but also, the concept of each cell must be known in order to map them in to a standardized structure. Knowing the concept of a cell is not only an issue when working with non-standardized spreadsheets, but is also an issue in structured systems, like relational databases, where the columns have weak or no validation rules in place. For instance, public B2B insurance customers, such as municipalities, often send a spreadsheet containing all the properties they want insurance for. These spreadsheets are usually a list of properties where a single column is used to describe a specific property. The problem is that this column includes several concepts, where a single fact represents a building (either by name, such as schools, or address(es)) or a technical facility (e.g., "water pumping station", "public toilet") in plain text. When these lists are imported directly in to a relational database, the integrity of the description field is diminished if the facts are expected to be a concept such as a locatable address, which makes it particularly challenging to automate processes dependant on that data. Even using such data for supervised machine learning becomes impractical, as they cannot be trusted to have the correct concept assigned as a label.

As knowing the concept of a fact is an issue both with spreadsheets and structured systems with weak validation, the objective of this thesis is to explore possible methods for representing a textual fact with numerical features so that classification algorithms could be tested as a method for determining the concept of a fact. Possible use cases for such an approach is to simplify

Concept	Label	Fact
Quantity	QNT	16 kg
Date	DATE	2/19/15
Date and time	DATETIME	19.09.2018 21:00
Percent	PERCENT	19,1 %
Organisation	ORG	Easet Help as
Money	MONEY	kr 20,50
Time	TIME	14:18:20
Text	TEXT	GRUNNLAG MVA
Product	PROD	1 stk Ocean Optics Jaz Spectrometer 0376
Event	EVT	Innbrudd Halvorsen Offshore Angholmen 02.03.2015
Facility	FAC	Sunkost Amfi Alta
Cardinal	CARDINAL	268 761,10
Location	LOC	Fjellveien 6

Table 1.1: Samples from annotated data set.

information extraction from spreadsheets and data wrangling (mapping raw data to a new structure) in legacy structured systems. For this purpose, I created a data set consisting of 119,963 annotated facts from spreadsheets linked to actual Norwegian property claims at Protector Forsikring. The annotations are a basic set of concepts, e.g., product, money, organisation, location, facility, date, etc., which are typically found in an insurance claim spreadsheet (see Table 1.1 for examples). The data set was split into a training set (80%) and a test set (20%). I used the the training set to train different machine learning models, using different numerical representations of the textual facts, and I evaluated each method using the test set.

I initiated the annotation process by defining rules, in the form of regular expressions, where the aim was to use them to obtain samples of each concept that I had defined. The reason for using such rules was to streamline the process of building an initial training set that could

Method	Macro avg. F_1	Numeric	String
Rule-based	52.90%	73.54%	28.83%
Naïve Bayes (character 3-gram, normalized)	79.11%	87.31%	69.55%
5-NN (FastText CBOW, normalized)	**87.09%**	**96.97%**	**75.57%**
5-NN (Sentence-BERT, raw)	82.26%	95.13%	67.25%
Multi-Index (FastText Skip-gram, normalized)	64.78%	77.19%	50.30%

Table 1.2: Results of the methods (using parameters with highest accuracy); best score in **bold**, **Numeric** shows the accuracy of numeric concepts (*QNT, DATE, DATETIME, PERCENT, MONEY, TIME, CARDINAL*) and **String** shows the accuracy of string concepts (*ORG, TEXT, PROD, EVT, FAC, LOC*).

be used by a multinomial naïve Bayes classifier (MNB), which was thought to annotate more accurately given enough training samples. However, although the annotations were assigned automatically, each annotation was manually validated as both the rule-based approach and the MNB had limitations which decreased the accuracy. The rule-based approach only included rules for certain concepts, as the concepts of facilities, events and products are hard to capture with generalized rules, while the MNB was sensitive to the distribution of the different concepts (due to the fact that the MNB models the data distribution). The MNB uses the prior probability of a concept (given as the distribution of a concept in the data set) which led it to find few samples of, e.g., quantity and event, which has a distribution of 0.29% and 0.08%, respectively, in the final data set. With the final data set, the rule-based approach reached an F_1-score (harmonic mean between correct classification and missed classifications, used as a measure of accuracy) of 52.90%, while the MNB reached 79.11% by using normalized text as input and representing the text as a bag-of-character 3-grams (see Table 1.2 for results).

Furthermore, I experimented with more modern methods for representing text as numerical features. As the facts in spreadsheets include more than one word, two different approaches for obtaining sentence embeddings (i.e., numerical feature vectors of sentences) were included in the experiment. The composition method, which aggregates embeddings using a pooling strategy (mean, sum, or max values in each embedding), was performed by using the mean of pre-trained word embeddings from FastText [1], [2]. The distribution method, where sentence embeddings are trained, e.g., using a neural network, was applied by using the pre-trained Sentence-BERT [3] model. In order to compare the two methods, the k-nearest neighbors (k-NN) classifier was used – which classifies based on the training samples nearest to the input data (the majority class of k is selected as the class for the input), as it can determine if the embeddings of the same concept are close to each other in vector space.

The k-NN classifier achieves higher accuracy than the MNB baseline with both the composition and distribution method for feature representation with 87.09% and 82.26%, respectively (see Table 1.2). I found that the best composition strategy was to use the mean of word embeddings trained with the continuous bag-of-words model (CBOW; trained to predict word based on context), while also normalizing the text by separating digits, alphabetical and special characters with a single whitespace and case-folding so that all characters were lowercase. For instance, with normalization, the text "19,1 %" was transformed to "19 , 1 %" and "Sunkost Amfi Alta" to "sunkost amfi alta", and the text was split into tokens by whitespace and each token got its own word embedding which was used in the pooling step (i.e., without normalization "19,1" was interpreted as a single token while "19 , 1" was interpreted as three tokens). A possible explanation for why the composition method achieves better scores than the distribution method is that Sentence-BERT is trained on grammatically correct sentences, while the facts in spreadsheets can be interpreted as bag-of-keywords, causing the composition method to be a more suitable approach as it is trained to capture the semantics of individual words.

In all the experiments with the k-NN classifier, k was set to 5, meaning that predicted concept needed the majority among the top 5 nearest samples. However, the computational complexity of computing the nearest neighbors in k-NN is $O(n)$, where n is the number of training samples ($n = 95,965$ in the data set that I created, i.e., evaluating all test samples requires $95,965 \times 23,998$ comparisons). To reduce the number of comparisons needed for each test sample, I conducted an experiment where each concept was represented by the mean of all its training samples and setting $k = 1$, reducing the number of comparisons to $13 \times 23,998$ (called multi index (MI) as it represents a concept by single indexed embedding). With the CBOW model, both using raw and normalized input achieved scores on par with the rule-based classifier. However, the skip-gram model (trained to predict context based on current word) achieved a score of 34.83% on raw input while 64.78% on normalized input; i.e., using raw input with skip-gram achieves a worse score than selecting the most probable concept (*TIME*: 35.12% of the data set), but normalized input boosts the method to be significantly more accurate than rule-based classification. Although the

embeddings of concepts were obtained using all 95,965 training samples in the MI, I tried to train it using only a few samples per concept. I found that with only ten samples per concept, the MI (using the skip-gram on normalized text) was able to produce an accuracy (51.30%) on par with the rule-based classifier. This observation indicates that representing a concept by the average of all word embeddings in the concept (i.e., average of samples represented as composite word embeddings) is a more efficient method for streamlining an annotation process – rather than writing rules that aims at capturing distinct features.

In order to evaluate how good a method is for the classification of a data set, both accuracy (measured by F_1-score) and evaluation time should be considered, as well as how flexible the method is w.r.t. introducing new classes (as the concepts found within insurance-claim-related spreadsheets can change over time). At first, I found that the large number of comparisons in the k-NN lead to an unfeasible long evaluation time when using the cosine similarity as distance metric – as it includes relatively complex computations, and evaluation was estimated to be finished in 14 hours. However, I found that the dot product between normalized vectors yields the same result as the cosine similarity function, which caused the evaluation to be finished in roughly 2 hours and 50 minutes. To put in perspective, both the rule-based classifier and MNB finished the evaluation in under five seconds. The MI evaluated in around 20 seconds; however, I used the cosine similarity in the implementation, so given that the dot product made the k-NN at least five times faster, the evaluation time of the MI can be reduced to the same level as the rule-based classifier and MNB with a minor improvement to the implementation. The reason for why I had an emphasis on the k-NN method, in addition to being able to observe which method for obtaining sentence embeddings grouped similar concepts best in vector space, was that new classes can be added by simply adding new training samples for the k-NN. Therefore, as a possible future work, I would suggest to try methods for *approximate nearest neighbor* (where the training samples are organized in a hierarchy of clusters) in order to reduce the computational complexity of k-NN.

To conclude, I found that for the task of extracting claim costs from spreadsheets, the approach I used relies too much on the cost amounts being formatted as monetary values (e.g., by having a currency symbol or currency code in the textual value). As we cannot simply add such formatting retroactively, because there are some numeric values without a specific concept, concept determination of cell values is not a suitable approach for extracting information that affect the claim handling. However, the approach makes meta-information, such as organisation names, locations, etc., more accessible in the sense that we can retrieve values of specific concepts. For instance, I show that we can more efficiently extract organisation names, and then use those values to search externally and find additional information. As extracting information that do not affect the claim handling, it can be performed as a background process – limiting the need for a highly efficient classification method. Therefore, I suggest that the k-NN should be used to extract meta-information, as it achieves the best accuracy, and representing the cell values as the average embedding of words. Furthermore, I provide some suggestions for future work: how to improve the data set that I created, fine-tuning the word embeddings on unstructured insurance-related documents (to make the embeddings more suitable to the insurance domain), and investigate a method to make the k-NN classify more efficiently.

1.1 Thesis Outline

- **Chapter 2**: I start by giving a brief introduction to machine learning before I introduce some basic techniques for natural language processing (NLP). Then, I move onto more modern techniques for NLP, where I give an introduction to feature representation of text that are able to encode the semantics of words (embeddings). After I give an introduction to embeddings, I show how we can use cosine similarity to determine how similar two given texts are – based on semantic similarities. Finally, I provide an explanation of k-means

(an algorithm for unsupervised learning) and two supervised learning algorithms that can be used for text classification: k-nearest neighbors and multinomial naïve Bayes.

- **Chapter 3**: I provide a walk-through of how I created the annotated data set. First, I explain how I processed the raw spreadsheets to obtain the textual values, followed by an introduction to the concepts that I found in insurance-claim-related spreadsheets. Then, I describe the rule-based classifier that I used to initiate the annotation process and then I explain how I used a multinomial naïve Bayes classifier (trained on the accumulated data set from the rule-based approach) to gather more samples for the different concepts.

- **Chapter 4**: I describe some methods related to the classification of concepts. I begin by explaining the tools that I used for implementation and obtaining pre-trained embeddings, and then move to the implementation of some functions that I used to transform the raw text into normalized text. The last section includes the implementation of the multi index that I used in an effort to reduce the number of comparisons in a k-nearest neighbors classifier.

- **Chapter 5**: I explain some issues with the data set I created, and I provide some metrics of the different methods that I used for classification. The metrics include the accuracy of models (including accuracy on a concept level for each model with the best settings) and the time it took to train and evaluate the different methods.

- **Chapter 6**: I provide some suggestions on how to improve the data set (such as reviewing the data set and using more fine-grained concepts) and how to potentially improve the classification (w.r.t. to a nearest-neighbors algorithm).

- **Chapter 7**: Finally, I provide a conclusion and a summary of my thesis.

Chapter 2

Background

In this chapter, I give a brief introduction to machine learning (Section 2.1) and a method for representing text as numerical values (Section 2.2) so that text can be used as input to a machine learning algorithm. In Section 2.3, I gradually introduce techniques for processing natural language. In Section 2.4, I provide an explanation of a more modern method for representing text as numerical values, which is able to encode the semantics of a text, followed by an explanation of how we can compare two texts based on semantic similarities (Section 2.5). Lastly, in Section 2.6, I present different machine learning algorithms that are relevant for this thesis.

2.1 Machine Learning

Machine learning (ML) is concerned with making computers learn from data so that analytical tasks can be automated. Instead of having a programmer define logical rules of a function (i.e., what to return given an input), ML algorithms have the objective of finding out what to generally return based on observed data. ML has existed for a long time, with Rosenblatt introducing a simple neural network (*the Perceptron* [4]) in the 1950s. However, due to the number of mathematical operations required by most ML algorithms, they were unfeasible to use until machines became more powerful in the 1980s and 1990s. ML then gained more traction as a research field, but was later limited by lack of training data. In the 2010s, when computers became more powerful and more data suitable for ML training was accessible, deep learning (DL) became popular within the fields of computer vision and text processing. DL algorithms have been proved to be quite capable of learning good representation of images and words, so that with enough training data, they achieve a high accuracy in tasks such as optical character recognition (OCR), machine translation (MT), named entity recognition (NER), etc.

2.1.1 Optimization Problem

ML aims at emulating the human process of learning by finding a function, based on observed data, that can be generalized to all unseen data (i.e., most optimal function is the function that is as close to reality as possible). For a classification task, we assume that a function $f : D \longrightarrow C$ classifies each element in D to the correct class in C. Furthermore, we are given a training set which is a sample set from D, i.e., $T \subseteq D$. Then, a function f_t is trained on the training set T until it coincides with f on T: $f_t(x) = f(x) \, \forall \, x \in T$. How the function is trained depends on the algorithm that is used. The objective is to find a function $f' : D \longrightarrow C$, which is as close to f as possible for all data in D (i.e., based on a training set T, the function should be generalizable to all data in D). The motivation for using ML to automate analytical tasks, such as classification, is that humans can only process a limited amount of data, and while such tasks can often be automated using rules (e.g., defined by a programmer), analyzing data to detect edge cases is time consuming and maintaining the rules is costly. Therefore, ML can be used to infer the

mapping from a domain based on observed samples (i.e., subset of population – all data), where an important feature is that the resulting function (or model) is generalizable to unseen data. Possible use cases for ML can for instance be to detect insurance fraud (classification – main focus in this thesis) or calculating risk of loss for an insured object (regression – estimating the relations between variables, not covered in this thesis) and there are mainly two categories of ML algorithms; supervised learning and unsupervised learning.

2.1.2 Supervised Learning vs. Unsupervised Learning

In supervised learning, the training set consists of data with the expected output. Given a data set $\{\langle x_1, y_1\rangle \langle x_2, y_2\rangle ..., \langle x_n, y_n\rangle\}$, where y_i denotes the expected output of the datum x_i, the objective is to find a function f so that $f(x_i) = y_i$, where $1 \leq i \leq n$. In classification tasks, the expected output of a datum is a class (or label) which is a member of a pre-defined set of labels. For instance, if the task is to detect fraudulent insurance claims (classify a claim as either fraudulent or not), it is required to have a data set where each datum is annotated as either fraudulent or legit, i.e., $\{\langle x_1, y_1\rangle \langle x_2, y_2\rangle ..., \langle x_n, y_n\rangle\}$ where x_i is an insurance claim and $y_i \in \{FRAUD,\ LEGIT\}$.

There are algorithms that do not require an annotated data set and, therefore, the learning process is considered unsupervised. If the data set has no mapping between a datum and a label, i.e., data set is $\{x_1, x_2, ..., x_n\}$, then supervised algorithms cannot be used, but unsupervised algorithms can. The most common use of unsupervised learning is cluster analysis, where the output y indicates which cluster x belongs to (similar objective as with supervised classification, $f : D \longrightarrow C$, except C is unknown, but can be restricted by, e.g., number of possible clusters). As supervised algorithms depend on having data which is as close to reality as possible, using them for fraud detection may be problematic as not all insurance fraud is detected, thereby, causing some fraudulent claims to be annotated as legit. In this case, using cluster analysis to group claims may be a more suitable approach, where new claims would be processed by the learned function and assigned to a cluster. The clusters could then be analyzed to see if the majority of their members are fraudulent, thereby, providing a generalized method for analyzing incoming claims.

Annotating data to use supervised learning is a costly and difficult process. The process often requires manual annotation to ensure that the quality of the data is good enough so that the algorithm can detect distinction between, e.g., labels. It is difficulty as, depending on the domain, the data can be ambiguous – meaning that two different individuals could possibly annotate the same datum with two different labels. For instance, the authors of NorNE [5] (Norwegian corpus for named entity recognition) had two individuals manually annotate roughly 600,000 sentences with entity types (e.g., organisation, person, products). Their reason was that having two annotators would increase the quality (i.e., by minimizing ambiguity) but it also increases the cost of the process. As the cost can be a reason not to use supervised learning, it should be considered whether unsupervised learning can work for the task at hand.

Disregarding which approach to use, both require numerical representation of the data in order to compute the result with ML algorithms. These numerical representations are called feature vectors, where each value represents a single feature of the data. The process of defining what the features represent is called feature extraction.

2.1.3 Feature Extraction

Features are characteristics of raw data denoted by a numerical value (e.g., age of driver is a characteristic of an automobile insurance claim) so that it can be used for computation by ML algorithms (i.e., algebraic operations). These features are usually extracted by domain experts in the feature extraction phase. Each feature is represented as either a numerical, boolean (0 or 1), or categorical (integer representing a single category) value extracted from the raw data.

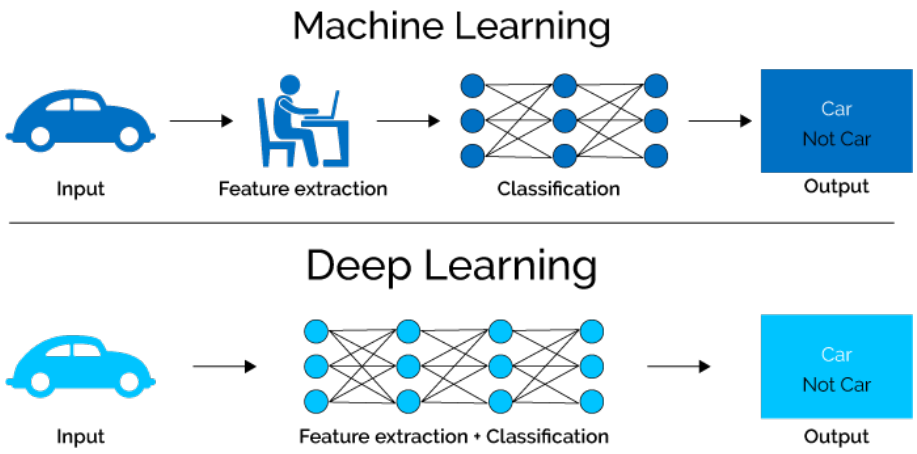


Figure 2.1: Feature extraction in machine learning vs. deep learning[1].

An insurance claim contains a lot of data which are not suitable for algebraic operations. To expand on the example given in Section 2.1.2 of fraud detection, the raw data of an insurance claim would need a numerical representation. For instance, an automobile insurance claim could have the following features; age of driver (numerical), has a driver's license (boolean) and type of damage (categorical; 0: fire, 1: theft, 2: collision, 3: self-accident), resulting in a feature vector with a dimension of three. For instance, a reported claim where a *19* year old driver with *no driver's license* who has *collided* with a bus is then represented by the vector $\langle 19, 0, 2 \rangle$.

However, in order to normalize the feature values to be in the range $[0, 1]$, the age of driver could instead be represented as a boolean feature (e.g., is driver under the age of 23) and each damage type could be represented as separate boolean features (i.e., is fire, is theft, etc.). This would cause the example claim above to be represented as a vector with a dimension of six instead of three, $\langle 1, 0, 0, 0, 1, 0 \rangle$, where the values are on the same scale. However, larger dimensions affects the computation complexity of ML algorithms, and in such cases, feature selection might be useful in order to drop features that are not important for finding an optimal function.

Modern machine learning algorithms have made it more common to automate the feature extraction process by allowing the ML model to learn good features itself. With deep learning (DL), which is a subset of ML, the feature extraction is merged into the learning process (see Figure 2.1). Models such as the GPT-3 [6] and BERT [7] are used as pre-training techniques of languages; i.e., they are trained to learn a representation of natural language (text). In natural language processing (NLP) tasks, using pre-trained representation comes with several benefits; the feature extraction process is not done manually, reduces the complexity of the pipeline (i.e., system between the raw data and output requires less processing of the raw data before it is processed by the model), high-performance models for specific tasks as the pre-trained representations are more understandable to machines in contrast to features determined by humans.

Although DL methods can improve the performance by learning representations, an initial numerical representation of raw data is always needed. There are a few different methods for how to represent text as numerical values, where the most basic is a bag-of-word representation.

2.2 Bag-of-words

Bag-of-words [8] (BOW) is a method for representing text as an un-ordered set (or multi-set – if the frequency is used) of words. The numerical values of a bag of words is represented by the occurrence of a word, either whether it occurs (boolean) or frequency (numeric), in a given

[1]Image from: https://laptrinhx.com/cnn-application-on-structured-data-automated-feature-extraction-4111608301/

text. The first step of BOW is to build a vocabulary consisting of unique words in a corpus (document collection). Each unique word is assigned an identifier which refers to the index in a feature vector. The result is that text can be represented by a vector x of size $|V|$, where V is the vocabulary of unique words and value at $x_i > 0$ if word i occurs in the text.

Example 1. Considering the example corpus shown in Table 2.1, each document can be represented as a bag-of-words. In the table, each token represents a word and the corpus consists of 16 unique tokens (i.e., $|V| = 16$). Using the vocabulary of the corpus (Table 2.2), each document Table 2.1 are represented by the following feature vectors with frequencies:

$$\begin{aligned} d_1 &= \langle 1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0\rangle \\ d_2 &= \langle 0,0,0,0,0,0,2,1,1,1,0,0,0,0,0,0\rangle \\ d_3 &= \langle 0,0,0,1,0,0,0,1,0,0,1,1,1,0,0,0\rangle \\ d_4 &= \langle 0,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1\rangle \end{aligned} \tag{2.1}$$

Instead of representing each word by the occurrence, causing every word to have the same importance, the values can be weighted by using *term frequency-inverse document frequency* (tf-idf) statistics. Considering Zipf's law, which states that the occurrence of a word in any given corpus is proportional to the rank in the frequency table (i.e., most frequent word occurs twice as often as the second, three times more than the third, etc.), using weighting allows for having a notion of the importance of each feature by assigning a smaller value to more common words. Tf-idf is the product of term frequency (frequency of a word in a document) and the inverse document frequency (frequency of a word in the corpus). Term frequency is defined in Definition 2.2.1, inverse document frequency in Definition 2.2.2, and tf-idf in Definition 2.2.3. Using tf-idf to apply weights to the features of the documents shown in Example 1, the documents are represented by the feature values shown in Table 2.2.

Definition 2.2.1 (Term frequency)

$$tf(t,d) = \frac{f_{t,d}}{\Sigma_{t' \in d} f_{t',d}} \tag{2.2}$$

where $f_{t,d}$ is the frequency of word t in document d.

Definition 2.2.2 (Inverse document frequency)

$$idf(t,D) = log\frac{N}{n_t} \tag{2.3}$$

where N is the number of documents ($|D|$) and n_t is the number of documents that word t occurs in.

Definition 2.2.3 (Term frequency-inverse document frequency)

$$tfidf(t,d) = tf \cdot idf \tag{2.4}$$

Document	**Content**	**Tokens**
d_1	"Limited offer: 5 % off!"	limit, offer, :, 5, %, off, !
d_2	"Reporting done right!!"	report, do, right, !, !
d_3	"Damages of 5 mNOK reported"	damage, of, 5, mnok, report
d_4	"Claim settlement offer accepted"	claim, settlement, offer, accept

Table 2.1: Example of an e-mail corpus.

Index	**Token**	**idf**	d_1	d_2	d_3	d_4
0	limit	0.60	**0.60**	0	0	0
1	offer	0.30	**0.30**	0	0	**0.30**
2	:	0.60	**0.60**	0	0	0
3	5	0.30	**0.30**	0	**0.30**	0
4	%	0.60	**0.60**	0	0	0
5	off	0.60	**0.60**	0	0	0
6	!	0.12	**0.60**	**0.24**	0	0
7	report	0.30	0	**0.30**	**0.30**	0
8	do	0.60	0	**0.60**	0	0
9	right	0.60	0	**0.60**	0	0
10	damage	0.60	0	0	**0.60**	0
11	of	0.60	0	0	**0.60**	0
12	mnok	0.60	0	0	**0.60**	0
13	claim	0.60	0	0	0	**0.60**
14	settlement	0.60	0	0	0	**0.60**
15	accept	0.60	0	0	0	**0.60**

Table 2.2: Feature representation of corpus (Table 2.1) using BOW with tf-idf weighted values.

There are, however, some issues with BOW representation of text. Consider the vocabulary in Table 2.2; vectorizing a new document d_5 with the content "Claim free money now!" will yield a vector where only $d_{5,6}$ and $d_{5,13}$ are non-zero values as "free", "money" and "now" does not exist in the vocabulary. These words are regarded as out-of-vocabulary words and if the corpus is small, out-of-vocabulary words can appear more frequently when trying to vectorize new documents. On the other hand, while increasing the number of documents in the corpus solves out-of-vocabulary words by having more words in the vocabulary, it will drastically increase the number of features per document. For instance, if each feature is represented by a four byte floating point, a corpus with 1,000,000 documents containing 30,000 unique words will require 120 GBs (120 kBs per feature vector) to store on disk or in memory. By applying some pre-processing techniques within natural language processing, the vocabulary size can be somewhat reduced even with a large corpus.

2.3 Natural Language Processing

Natural language processing (NLP) is a collective term for applying different techniques to process text written in natural language. NLP is a field that includes many research topics, such as named entity recognition (NER), speech recognition, part-of-speech (POS) tagging, sentiment analysis, etc. While these topics are mainly used as tasks to achieve an end-goal in a practical use case, NLP also includes research topics where the goal is to pre-process the text so that it is more suitable for further processing and makes the model for a specific task, e.g., NER, more accurate. A general term for pre-processing text at a word level, is normalization.

2.3.1 Normalization

Normalizing a text on a word level is often done so that the corpus does not contain a high variation of different words that have similar semantics (meaning). As mentioned in Section 2.2, normalization can reduce the number of unique words in a vocabulary. A basic normalization technique is to transform all characters to lowercase letters, which is called *case folding*. By doing case folding when creating a vocabulary for a BOW model, the vocabulary will not contain duplicate words where the only difference is upper- and lowercase letters.

Example 1. Consider the following sentences:

$$\begin{aligned} s_1 = &\quad \textit{“University of Oslo”} \\ s_2 = &\quad \textit{“studying at the university”} \end{aligned} \tag{2.5}$$

Without case folding, the words “University” and “university” will be two unique features in the final BOW model, while with case folding, these are merged and represented as one feature.

Furthermore, each word can be reduced to a base form. *Lemmatization* is one technique for transforming a word to its base form. However, this technique is dependent on having a vocabulary in which a word can be looked up in and is, therefore, not useful when creating a new vocabulary. *Stemming* is another technique for reducing words to a base form, where rules are defined for how to cut suffix of certain words.

Example 2. The Porter algorithm [9] includes the following rules for stemming English words:

$$\begin{aligned} r_1: &\quad \textit{“ing”} \implies \textit{“”} \\ r_2: &\quad \textit{“ies”} \implies \textit{“i”} \\ r_3: &\quad \textit{“y”} \implies \textit{“i”} \end{aligned} \tag{2.6}$$

Using these rules, the word “studying” from s_2 in Example 1 is stemmed to a base form in two steps:

$$\begin{aligned} r_1: &\quad \textit{“studying”} \longrightarrow \textit{“study”} \\ r_2: &\quad \textit{“study”} \longrightarrow \textit{“studi”} \end{aligned} \tag{2.7}$$

The rules above entails that “studying”, “studies” and “study” are all reduced to the base form “studi”.

These are just some techniques for normalizing text in NLP. As the normalization is executed at a word level, and the input data in a NLP pipeline are larger texts that include several words, another required pre-processing step is needed in order to segment the words in a text.

2.3.2 Tokenization

Word segmentation (tokenization) is the process of splitting a larger text into smaller components (tokens), where the tokens corresponds to words in the text. Tokenization is a challenging process as there are many consideration to take into account, e.g., abbreviations. Consider the abbreviated word “hasn’t”; there are multiple different interpretations w.r.t. which token or tokens exist in that word, such as:

"hasn't"
"hasn", "t"
"hasn", "'", "t"
"has", "n't"

and so on. In the example of an e-mail corpus in Section 2.2, it is assumed that the content has been tokenized (Table 2.1) using a simple approach of separating tokens by whitespace and separate special, numeric and alphabetical characters from each other, so the word "hasn't" is interpreted as three different tokens ("hasn", "'", "t"). There is also the issue of compound words, which are multiple words merged together to form a new word, e.g., "southwest", that can cause similar features to be represented differently.

While words are mostly separated by whitespace in English, languages such as Norwegian contain a lot of compound words. In Norwegian, "damage report" is translated to "skaderapport", compounded by the words "skade" (English: "damage") and "rapport" (English: "report"). Finding tokens among compound words requires either a dictionary to look up in (similar to lemmatization) or complex rules which are expensive to write and maintain as written language changes over time. One approach for tokenization that can be useful for languages consisting of compound words is to use n-grams as tokens.

2.3.3 *N*-grams

N-grams are sequences that consist of n items. A bi-gram consists of two items, a tri-gram consists of three items, and so forth. In NLP, n-grams are used to represent either word or character sequences as tokens.

On a character level, a single n-gram consists of n number of characters concatenated together to represent a single token. The n-grams are found by scanning n characters of a given text using an offset of 1. As there are at most k^n possible combinations for a n-gram, where k is the number of unique characters (or words), n is often set as a low number, e.g., $3 \leq n \leq 6$.

Example 3. Let T be the set of n-grams for the string $s =$ *"skaderapport"*, where $n = 5$:

$$T = \{\textit{"skade"}, \textit{"kader"}, \textit{"adera"}, \textit{"derap"}, \textit{"erapp"}, \textit{"rappo"}, \textit{"appor"}, \textit{"pport"}\}$$

Using character n-grams for a corpus containing the words "skade" (plural: "skader"), "rapport" and the string s, the structural similarities between s and the other words are found by the intersection of n-grams:

$$T \cap \{\textit{"skade"}\} = \{\textit{"skade"}\}$$
$$T \cap \{\textit{"skade"}, \textit{"kader"}\} = \{\textit{"skade"}, \textit{"kader"}\}$$
$$T \cap \{\textit{"rappo"}, \textit{"appor"}, \textit{"pport"}\} = \{\textit{"rappo"}, \textit{"appor"}, \textit{"pport"}\}$$

In this example, words that are structurally similar to "skaderapport" will have overlapping features in a BOW representation. As such, character n-grams can be a useful tokenization method where compound words are frequently found or where words are not clearly separated (e.g., by whitespace).

N-grams can also be used on a word level. Although character n-grams are used to find structural similarities between features, n-grams of words are used to separate features by their context. This approach requires that the input text is already tokenized as it consist of concatenating n tokens to create a new token with some notion of context. For instance, the bi-grams "machine learning" and "artificial intelligence" are part of the sentence "machine learning is a sub field of artificial intelligence". Using bi-grams of words as features will enable to separate text by the context of their words., e.g., texts that are either about machines or intelligence will be separated from machine learning and artificial intelligence related texts. However, for capturing the context of the content, there are more efficient methods for feature representation than using a BOW model consisting of word n-grams, where the context of a token is encoded in an *embedding* (i.e., vector).

2.4 Embeddings

Embeddings are representation of data that has been compressed to a smaller dimension so that data have a dense representation with only non-zero values. Although some techniques within the field of NLP can be applied in order to reduce the number of unique tokens, a BOW-based feature representation is still very sparse, i.e., the majority of feature values are equal to zero. In Table 2.2, there are four different documents and 16 features where on average $\frac{11}{16}$ (68.75%) of a document's feature values are equal to zero. Adding new documents with roughly the same size but with new words will cause the documents to have an even more sparse representation.

Modern approaches for representing text are based on the distributional hypothesis [8], [10], which states that the meaning of a word is determined by its context (i.e., words that are surrounded by similar words have similar meaning). Papers such as Word2vec [11], GloVE [12], and FastText [1], [2] describe methods for training dense word representation using the context of words. The methods are beneficial, not only as they provide pre-trained word embeddings that can be used in different NLP tasks (e.g., semantic analysis, NER), but the learned representations have a relatively small dimension (typically 100-300) in contrast to BOW representations. The resulting feature representation encodes the semantics of its word, which comes with interesting opportunities, such as algebraic operations on words.[2]

$$\textit{“daughter”} - \textit{“mother”} + \textit{“father”} = \textit{“son”}$$
$$\textit{“king”} - \textit{“man”} + \textit{“woman”} = \textit{“queen”}$$

2.4.1 Word Embeddings

Word embeddings are distributed representation of words that are trained on word co-occurrences [12], [13] (statistics-based – see Table 2.3 for an example of a co-occurrence matrix) or context windows [11] (machine-learning-based – i.e., training set consists of words with their surrounding words and are found in unstructured texts), thereby encoding the semantics of words as, according to the distributional hypothesis, a words meaning should be known by its surrounding words (context) [8], [10]. The result is a trained model that takes a word (i.e., token) as input and maps it to a fixed-sized vector (e.g., vector with a dimension of 300), where the values are set based on the context of the given word (the context exists in the model's training data).

Word2vec presents two different models for efficiently learning word embeddings; continuous bag-of-words model (CBOW) and continous skip-gram (architecture shown in Figure 2.2). Both models use randomly initiated word vectors which are adjusted during training, i.e., the vectors are continuous distributed representations.[3]

The CBOW aims at predicting the current word based on the context, where the context is the four preceding and four trailing words [11]. All the context word vectors are averaged in the projection layer (Figure 2.2), therefore, the model is considered a BOW as the order of the context words is not taken into account in the model. On the other hand, the skip-gram model is given a word in which the aim is to classify the word (projection layer) and predict its context words (output layer).

GloVE [12] demonstrated a small increase in accuracy over the CBOW using a word-word co-occurrence matrix as initial representation (matrix X, where X_i is the vector of word i and X_{ij} is the frequency of word i occurring with the word j). Training a skip-gram on Wikipedia[4] data and representing words as a bag-of-character n-grams (experimented with different values for n), where the word is the sum of its n-gram vectors, achieved state-of-the-art on word similarity and analogy tasks compared to other methods for morphological word representations [1]. Further, extending the idea of using character n-grams, Grave et. al used Common Crawl[5] and Wikipedia

[2] Word vector calculator: http://vectors.nlpl.eu/explore/embeddings/en/calculator/#
[3] *InitNet* function: https://github.com/tmikolov/word2vec/blob/master/word2vec.c
[4] https://www.wikipedia.org/
[5] https://commoncrawl.org/

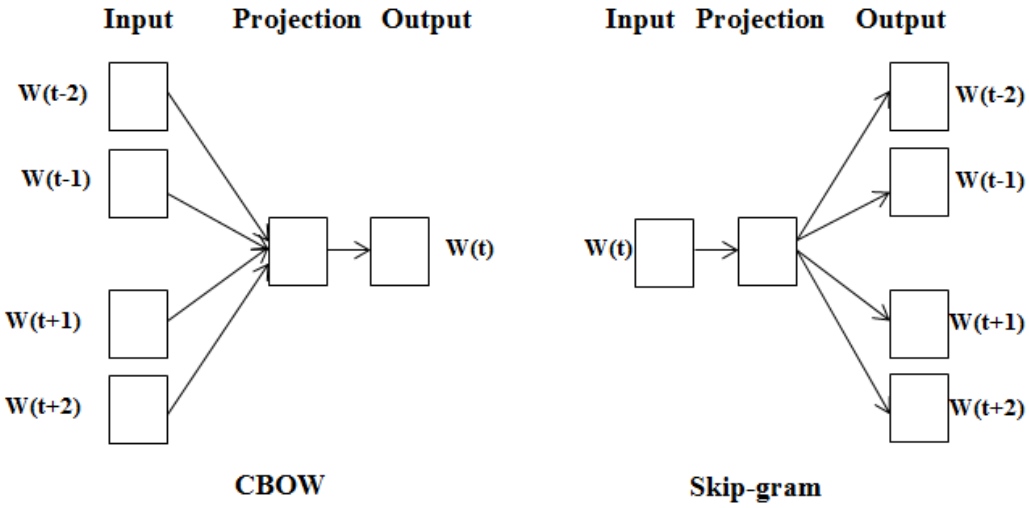


Figure 2.2: Architecture of CBOW and skip-gram [11].

data to learn word vectors for 157 languages [2]. They used the CBOW model instead of the skip-gram and set $n = 5$.

2.4.2 Sentence Embeddings

Sentence embeddings are similar to word embeddings except that the objective is to encode the meaning of whole sentences instead of their individual components. A common approach in NLP is to represent sentences as a $L \times D$ matrix, where L is the number of words in the sentence and D is the dimension of the word embeddings. However, sentence embeddings are represented as a single vector with dimension D, i.e., they have a fixed size rather than being a matrix of varied size. These embeddings are useful for different kinds of similarity tasks between sentences, e.g., natural language inference (determine if one sentence is an entailment, contradiction or neutral of another sentence), or question and answer retrieval. There are generally two approaches for obtaining sentence embeddings; the distribution method and the composition method.

The distribution method involves training embeddings based on a larger corpus, similar to the training process of word embeddings. Sentence-BERT [3] (SBERT), which extends the BERT model [14] (a neural network architecture for language modeling), reported state-of-the-art performance on sentence evaluation tasks. The motivation for SBERT was to reduce overhead in BERT, as BERT requires both sentences as input for pair-wise comparison. SBERT consists of a siamese network [15] (i.e., two networks with identical architecture and shared weights processing two different inputs at once) of BERT and triplets (Definition 2.4.1), as seen in Figure 2.3.

Definition 2.4.1 (Triplet loss function) *Triplet loss function is used for training similarity functions for vectors and is defined as:*

$$L(A, P, N) = max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0) \tag{2.8}$$

where A is an anchor vector, P is a vector similar to A and N is a vector dissimilar to A. The objective of the triplet loss function is to satisfy

$$||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 \leq 0 \tag{2.9}$$

so that the distance between the anchor and the positive sample, $||f(A) - f(P)||^2$, is smaller than the distance between the anchor and the negative sample, $||f(A) - f(N)||^2$. As a trivial solution to Equation 2.9 is that function f returns a vector with only zero values ($f(x) = \vec{0}$), α is used to indicate a margin between the positive and negative samples.

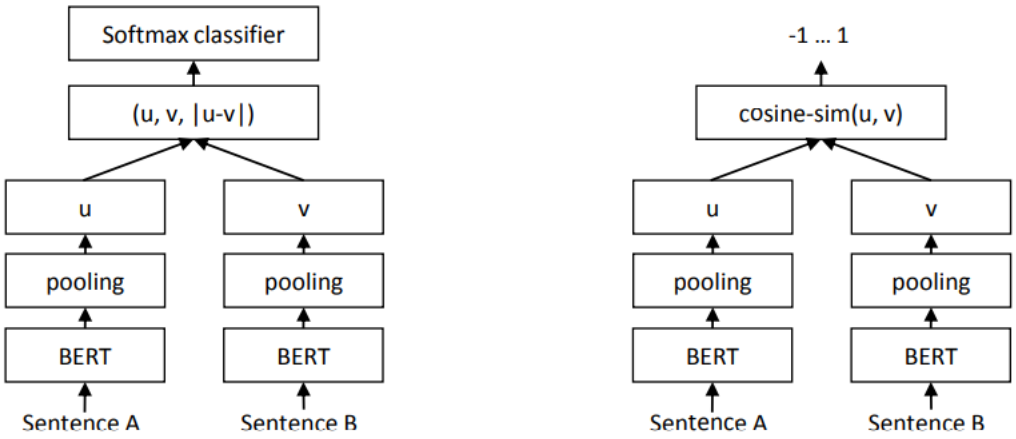


Figure 2.3: Architecture of SBERT; left) fine-tuning sentence embeddings, right) similarity inference of two sentence embeddings [3].

Another approach that is possible for obtaining sentence embeddings, is the composition method. Similar to how FastText trains word embeddings on an aggregation of character n-grams, the composition method involves aggregating lower level embeddings. More specifically, usually when the composition method is used, pre-trained word embeddings are aggregated according to specified strategy (pooling strategy, common strategies include sum, mean or max vector values) to form a representation of a sentence.

Both approaches yield a single vector for sentences that can be used for computing the similarity between two sentences. While early methods for text similarity was heavily based on the syntax, i.e., similarity was determined by similar characters and ordering, sentence embeddings yields representation in vector space and makes it possible to use algebra for computing similarity.

2.5 Measuring Text Similarity

Text similarity has historically been used in information retrieval (IR) systems for use cases such as spelling correction, retrieval ranking, and question and answer retrieval. There are several methods for comparing the similarity between strings. In this section, three different methods will be covered to get a basic understanding of how strings can be compared to each other and how a modern method allows for computing similarity based on semantic similarity. First, the edit distance and the Jaccard coefficient are presented, which are methods that only depend on the structure of each string. Then, the definition of cosine similarity is given, as this metric is more common to use now that text is represented in vector space.

2.5.1 Edit Distance

Edit distance is simply the number of different edit operations that has to be applied to string a so that it is equal to string b. The possible edit operations include insertion, deletion, substitution and transposition. There exist several algorithms for computing edit distance, where the main difference between them is which edit operations are allowed. For instance, the Levensthein distance [16] allows insertion, deletion and substitution, while the Damerau-Levensthein

distance [16], [17] extends the Levensthein distance by supporting transpositions.

Example 1. Levensthein distance

$$\begin{aligned} \textit{levensthein}(\text{“insure”, “insured”}) &= 1 && \text{(insert } d \text{ at the end)} \\ \textit{levensthein}(\text{“insuransed”, “insurance”}) &= 2 && \text{(substitute } s \longrightarrow c\text{, delete } d \text{ at the end)} \end{aligned} \tag{2.10}$$

Example 2. Damerau-Levensthein distance

$$\begin{aligned} \textit{levensthein}(\text{“insurde”, “insured”}) &= 2 && \text{(subst. } d \longrightarrow e \text{ and } e \longrightarrow d\text{)} \\ \textit{damerau}(\text{“insurde”, “insured”}) &= 1 && \text{(transpose: } de \longrightarrow ed\text{)} \end{aligned} \tag{2.11}$$

2.5.2 Jaccard Similarity Coefficient

The Jaccard similarity coefficient measures the similarity between two sets as a ratio between the number of equal set members and sum of set members in both sets. Generally, in NLP and IR, the Jaccard similarity is measured between two sets of tokens.

Definition 2.5.1 (Jaccard Similarity Coefficient)

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.12}$$

Example 3. Jaccard similarity of n-grams
Let $T = \{\text{“skade”, “kader”, “adera”, “derap”, “erapp”, “rappo”, “appor”, “pport”}\}$, $U = \{\text{“skade”}\}$ and $V = \{\text{“rappo”, “appor”, “pport”}\}$.

$$J(T, U) = \frac{|T \cap U|}{|T \cup U|} = \frac{1}{8} = 0.125 \tag{2.13}$$

$$J(T, V) = \frac{|T \cap V|}{|T \cup V|} = \frac{3}{8} = 0.375 \tag{2.14}$$

Both edit distance and Jaccard similarity are useful string metrics within IR, where the similarity between strings needs to be measured w.r.t. ranking and/or spelling correction, however, they have a limited number of use-cases within NLP (although they can be used in spelling correction in a production pipeline to limit out-of-vocabulary words caused by spelling errors). With the rise of word embeddings and increased use of vector space representation, cosine similarity has proven useful in a variety of NLP tasks.

2.5.3 Cosine Similarity

Cosine similarity is computed as the angle between two vectors. While the euclidean distance formula (Definition 2.5.2) is used to measure the distance between two points in vector space, it is not suitable for word embeddings as similar concepts tends to point in the same direction but vary in length (magnitude) – see Figure 2.4.

Definition 2.5.2 (Euclidean distance)

$$d(u, v) = \sqrt{(v_1 - u_1)^2 + (v_2 - u_2)^2 ... + (v_n - u_n)^2} \tag{2.15}$$

[7]Screenshot from: https://projector.tensorflow.org/

Figure 2.4: Word embeddings similar to "university" projected in 2D space; red line shows the euclidean distance and θ is the angle between "education" and "university".[7]

	I	drink	coffee	in	the	morning	tea
I	**1**	**1**	**1**	0	0	0	**1**
drink	**1**	**1**	**1**	**1**	0	0	**1**
coffee	**1**	**1**	**1**	**1**	**1**	0	0
in	0	**1**	**1**	**1**	**1**	**1**	**1**
the	0	0	**1**	**1**	**1**	**1**	**1**
morning	0	0	0	**1**	**1**	**1**	0
tea	**1**	**1**	0	**1**	**1**	0	**1**

Table 2.3: Simple co-occurrence matrix.

As illustrated in Figure 2.4, the distance between "education" and "university" is relatively large; however, the angle between these two vectors is small. The angle is computed using the cosine similarity function (Definition 2.5.5). The cosine similarity function is computed using the dot product between vectors (Definition 2.5.3) and magnitude of vectors (Definition 2.5.4).

Definition 2.5.3 (Euclidean dot product)

$$u \cdot v = u_1 v_1 + u_2 v_2 ... + u_n v_n \tag{2.16}$$

Definition 2.5.4 (Euclidean norm) *Definition for computing the Euclidean norm, i.e., magnitude (or length), of a vector in Euclidean space.*

$$||u|| = \sqrt{u \cdot u} = \sqrt{u_1^2 + u_2^2 ... + u_n^2} \tag{2.17}$$

Definition 2.5.5 (Cosine similarity)

$$similarity = cos(\theta) = \frac{u \cdot v}{||u|| \cdot ||v||} \tag{2.18}$$

Example 4. Computing cosine similarity As an example of cosine similarity, vectors of words can be used to illustrate their similarities. Let $n = 3$ denote the context size (i.e., preceding, current and trailing word) for counting co-occurring words and the count is a boolean value (0 or 1). Given a corpus of the two sentences:

$$\begin{aligned} s_1 &= \textit{"I drink coffee in the morning"} \\ s_2 &= \textit{"I drink tea in the morning"} \end{aligned}$$

the co-occurring words in the corpus are represented as bag-of-words:

{"I", "drink", "coffee"}
{"drink", "coffee", "in"}
{"coffee", "in", "the"}
{"in", "the", "morning"}
{"I", "drink", "tea"}
{"drink", "tea", "in"}
{"tea", "in", "the"}

The co-occurrences can be represented in a word-word co-occurrence matrix (Table 2.3) denoted as X, where X_{ij} is the count of word i occurring with word j and X_i is the word vector for i.

The cosine similarity between "coffee" and "tea" can be computed in a few simple steps: 1) the vector representations are found by looking up in the co-occurrence matrix (Equation 2.19),

	I	drink	coffee	in	the	morning	tea
I	1.00	**0.89**	0.67	0.61	0.45	0.00	0.67
drink	**0.89**	1.00	0.80	0.73	0.40	0.00	0.60
coffee	0.67	**0.80**	1.00	0.73	0.60	0.52	**0.80**
in	0.61	0.73	0.73	1.00	**0.91**	0.71	0.73
the	0.45	0.40	0.60	**0.91**	1.00	0.77	0.60
morning	0.00	0.00	0.52	0.71	**0.77**	1.00	0.52
tea	0.67	0.60	**0.80**	0.73	0.60	0.52	1.00

Table 2.4: Affinity matrix based on cosine similarity of vectors in Table 2.3. Most similar word (excluding the itself) in **bold**. Computational complexity is $O(n^2)$.

2) the dot product between the vectors is computed (Equation 2.20), 3) the magnitude of each vector is computed (Equation 2.21), and 4) the similarity between the vectors is computed by using the dot product and magnitude (Equation 2.22).

$$\begin{aligned} vec(\text{“}coffee\text{”}) &= x_2 = \langle 1,1,1,1,1,0,0\rangle \\ vec(\text{“}tea\text{”}) &= x_6 = \langle 1,1,0,1,1,0,1\rangle \end{aligned} \tag{2.19}$$

$$\begin{aligned} x_2 \cdot x_6 &= \langle 1,1,1,1,1,0,0\rangle \cdot \langle 1,1,0,1,1,0,1\rangle \\ &= 1\cdot 1 + 1\cdot 1 + 1\cdot 0 + 1\cdot 1 + 1\cdot 1 + 0\cdot 0 + 0\cdot 1 \\ &= 1+1+0+1+1+0+0 \\ &= 4 \end{aligned} \tag{2.20}$$

$$\begin{aligned} ||x_2|| &= \sqrt{1^2+1^2+1^2+1^2+1^2+0^2+0^2} = \sqrt{5} \\ ||x_6|| &= \sqrt{1^2+1^2+0^2+1^2+1^2+0^2+1^2} = \sqrt{5} \end{aligned} \tag{2.21}$$

$$similarity(\text{“}coffee\text{”}, \text{“}tea\text{”}) = \frac{x_2 \cdot x_6}{||x_2|| \cdot ||x_6||} = \frac{4}{\sqrt{5}\cdot\sqrt{5}} = 0.80 \tag{2.22}$$

Following these steps, the cosine similarity can be computed between each pair of words in the co-occurrence matrix, which can be inserted in an affinity matrix (X_{ij} is distance between i and j) such as Table 2.4.

Definition 2.5.6 (Unit vector)

$$\hat{u} = \frac{u}{||u||} \tag{2.23}$$

The definition for unit vector (Definition 2.5.6) can be used to normalize a vector so that the direction is kept but the magnitude is equal to 1. The dot product between normalized vectors yields the same results as the cosine similarity function. Knowing this, the computational cost can be reduced by pre-calculating the unit vector of each vector, thereby, removing the need for computing the magnitude for each vector in a pair when finding the similarity.

Example 5. Computing cosine similarity (continued)

$$\begin{aligned} \hat{x}_2 &= \frac{\langle 1,1,1,1,1,0,0\rangle}{\sqrt{5}} = \langle 0.4472, 0.4472, 0.4472, 0.4472, 0.4472, 0, 0\rangle \\ \hat{x}_6 &= \frac{\langle 1,1,0,1,1,0,1\rangle}{\sqrt{5}} = \langle 0.4472, 0.4472, 0, 0.4472, 0.4472, 0, 0.4472\rangle \end{aligned} \tag{2.24}$$

$$similarity(x_2, x_6) = \hat{x}_2 \cdot \hat{x}_6 = 0.80 \tag{2.25}$$

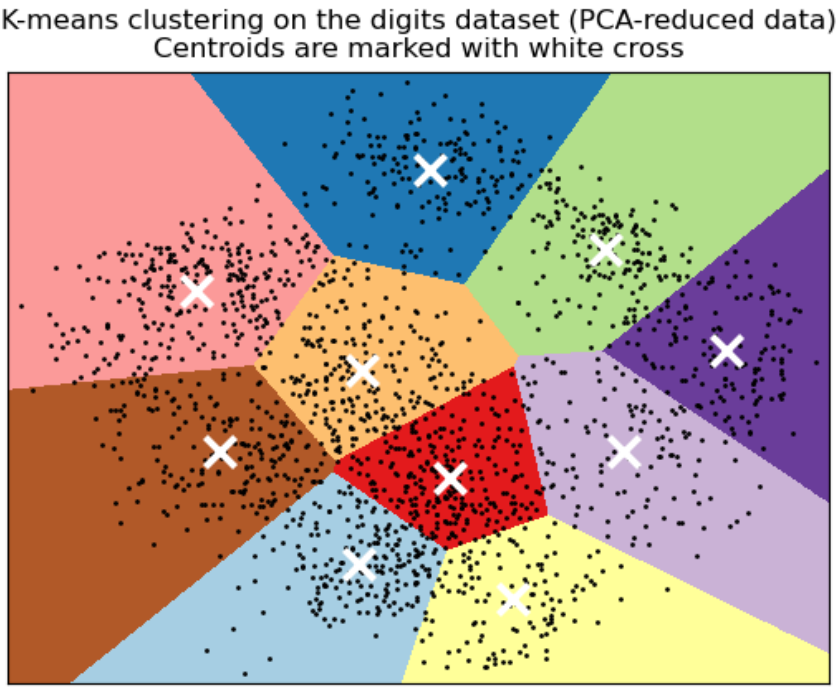


Figure 2.5: Voronoi diagram of k-means clustering.[8]

2.6 Algorithms

In this section, I present three machine learning algorithms that are relevant for this thesis. Two algorithms that depend on a distance metric are presented first; k-means (Section 2.6.1) and k-nearest neighbors (Section 2.6.2). The final algorithm that is introduced can be used for text classification and is popular to use as a baseline for text classification of a new data set (Section 2.6.3).

2.6.1 k-Means

The k-means clustering algorithm is an unsupervised learning algorithm that finds k number of clusters (i.e., sub spaces in a larger vector space). Each k^{th} cluster is represented by a centroid, which is the averaged vector of all the cluster members.

The algorithm is initiated by selecting k random data points from the training set as initial centroids. Then, each other data point is assigned to the closest centroid using a defined distance metric, e.g., euclidean distance (Definition 2.5.2) or cosine similarity (Definition 2.5.5). The centroid is then adjusted as the average of all its members. The training process consists of repeating centroid assignment and adjustment until either current iteration reaches the maximum number of iterations allowed, centroids are no longer adjusted, or the adjustment is below a certain threshold.

The result of k-means is the function $f(x) = y$, where y is the centroid x is closest to. For 2-dimensional vectors, the resulting function can be illustrated as a Voronoi diagram (Figure 2.5). Each region in the diagram represents a cluster and is given a unique number, i.e., index of centroid in the collection of all centroids. When x is within the i^{th} region (closest to centroid i), then $f(x) = i$.

[8]Image from: https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html

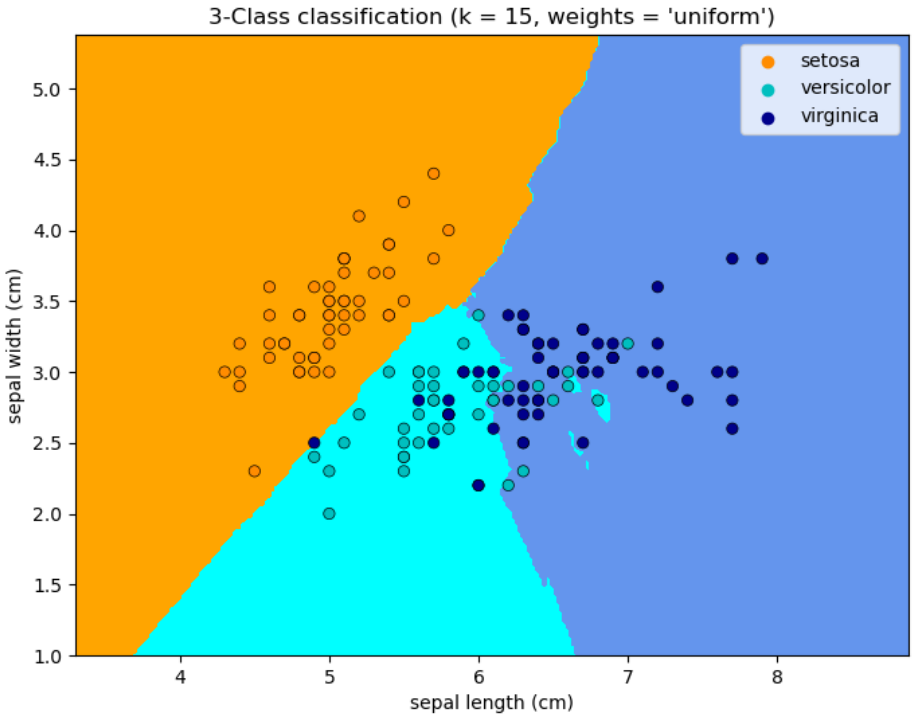


Figure 2.6: Voronoi diagram of k-NN classification.[9]

The algorithm is used in areas such as cluster analysis and feature learning. As mentioned in Section 2.1.2, a cluster analysis task can be to identify potentially fraudulent claims by analysing the members of each cluster. With some modifications, k-means can be used for feature learning [18].

2.6.2 k-Nearest Neighbors

The k-nearest neighbors (k-NN) algorithm is a supervised, lazy learning algorithm that can be used for classification. As k-NN is supervised, to use this algorithm it is needed to have an annotated dataset, i.e., $\{\langle x_1, y_1\rangle \langle x_2, y_2\rangle ..., \langle x_n, y_n\rangle\}$, where y_i is in a set Y of possible classes. Similar to k-means, it uses a distance metric to compute the distance between data points.

For classification, when the k-NN receives an input x, it computes the distance to the data points in the training set and selects the top k nearest data points. The output is the predicted class, which is the class with most occurrences among the k nearest of x (Figure 2.6 illustrates regions of different classes, including regions of outlier data points, with k-NN). Because the majority class is selected as prediction, k needs to be an odd integer.

The algorithm has some major limitations regarding complexity at prediction time. It is called lazy learning (also, instance-based learning) as there is no actual training involved, i.e., it use all data points (instances) in the training set for prediction. While algorithms such as k-means require some training time, they are relatively fast at prediction, but with k-NN this is reversed.

However, as k-NN is instance-based, introducing new classes does not require architectural modifications to the model or even re-training of the model. This is advantageous when the set of possible classes evolves over time. In addition, the computational limitation can be maintained to

[9]Image from: https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html

some extent as k-NN is easily parallelizable (computing distances can be done either by multiple threads or across different machines in a distributed system).

2.6.3 Multinomial Naïve Bayes Classifier

The multinomial Naïve Bayes classifier (MNB) is a supervised machine learning algorithm that can be used for, e.g., text classification. The MNB is a *generative* model, as it models the distribution of classes based on training data, in contrast to *discriminative* models which learns the boundaries between classes (e.g., k-means, k-NN).

Definition 2.6.1 (Bayes' theorem) *The Bayes' theorem is defined as:*

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)} \tag{2.26}$$

where $P(A)$ is the prior (observed) probability of event A, and $P(A \mid B)$ is the conditional probability of A and B, i.e., probability of A given B.

Using the *Bayes' theorem* (Definition 2.6.1), the MNB computes the probability of a class y given some input x, $P(y \mid x_1, x_2 \dots x_n)$, by using the learned prior probability of y and the learned conditional probability of x_i given y. For text classification, the prior probability is learned using Definition 2.6.2 and the conditional probability is learned using Definition 2.6.3. While the prior probability is computed for a class, the conditional probability is computed for a term (i.e., token). Therefore, the first step in the training is to create a vocabulary of unique terms in the training set. Having the learned probabilities, the class of an input text can be classified by selecting the class that has the highest probability (Definition 2.6.4) based on terms in the given text.

Definition 2.6.2 (Prior probability in multinomial Naïve Bayes classifier) *Prior probability of a class c is defined as:*

$$P(c) = \frac{N_c}{N} \tag{2.27}$$

where N_c is the number of documents belonging to class c and N is the total number of documents.

Definition 2.6.3 (Conditional probability in multinomial Naïve Bayes classifier) *The probability of a term t given a class c is defined as:*

$$P(t \mid c) = \frac{T_{ct} + 1}{(\Sigma_{t' \in V} T_{ct'}) + |V|} \tag{2.28}$$

where T_{ct} is the count of term t in class c and $|V|$ is the size of the vocabulary (i.e., number of unique terms in training data). Each count T_{ct} is added by 1 (called add-one, or Laplace smoothing) to eliminate zero probabilities.

Definition 2.6.4 (Multinomial Naïve Bayes classification) *The classification function f selects the class c with the highest probability given the features of x:*

$$f(x) = argmax_{c \in C} P(c) \prod_{i=0}^{n} P(x_i \mid c) \tag{2.29}$$

If the features of x has the same probability for more than one class, then the class with highest prior probability, $P(c)$, is selected. Note that the function f uses the Bayes' theorem, however, the denominator $P(x_i)$ is removed as it is the same for each class c and does not affect the maximum probability.

Document	**Content**	**Tokens**	**Class**
d_1	"Limited offer: 5 % off!"	limit, offer, :, 5, %, off, !	SPAM
d_2	"Reporting done right!!"	report, do, right, !, !	SPAM
d_3	"Damages of 5 mNOK reported"	damage, of, 5, mnok, report	CLAIM
d_4	"Claim settlement offer accepted"	claim, settlement, offer, accept	CLAIM

Table 2.5: Emails annotated as *spam* or *claim related*.

i	t	$P(t \mid \text{SPAM})$	$P(t \mid \text{CLAIM})$
0	limit	0.07	0.04
1	offer	0.07	0.08
2	:	0.07	0.04
3	5	0.07	0.08
4	%	0.07	0.04
5	off	0.07	0.04
6	!	0.14	0.04
7	report	0.07	0.08
8	do	0.07	0.04
9	right	0.07	0.04
10	damage	0.04	0.08
11	of	0.04	0.08
12	mnok	0.04	0.08
13	claim	0.04	0.08
14	settlement	0.04	0.08
15	accept	0.04	0.08

Table 2.6: Vocabulary of corpus in Table 2.5.

Example 1. Text classification Multinomial Naïve Bayes can be used for text classification, e.g., such as detecting spam among a corpus of emails. Let $C = \{\text{SPAM, CLAIM}\}$ denote the possible classes, where the task is to classify an email as either spam or claim related. By annotating the emails shown in Table 2.1, we get the training set

$$D = \{\langle d_1, \text{SPAM}\rangle, \langle d_2, \text{SPAM}\rangle, \langle d_3, \text{CLAIM}\rangle, \langle d_4, \text{CLAIM}\rangle\} \tag{2.30}$$

as illustrated in Table 2.5. The prior probability for each class is computed using Definition 2.6.2:

$$\begin{aligned} P(\text{SPAM}) &= \tfrac{2}{4} = 0.5 \\ P(\text{CLAIM}) &= \tfrac{2}{4} = 0.5 \end{aligned} \tag{2.31}$$

and the conditional probabilities are computed using Definition 2.6.3:

$$\begin{aligned} P(\text{limit} \mid \text{SPAM}) &= \tfrac{1+1}{12+16} = 0.07 \\ P(\text{limit} \mid \text{CLAIM}) &= \tfrac{0+1}{9+16} = 0.04 \\ P(! \mid \text{SPAM}) &= \tfrac{3+1}{12+16} = 0.14 \\ &\dots \end{aligned} \tag{2.32}$$

See Table 2.6 for all conditional probabilities. Given a new document d_5 = "claim!", with the tokens "claim" and "!", it can be classified using the function in Definition 2.6.4:

$$\begin{aligned} P(\text{SPAM} \mid d_5) &= P(\text{SPAM}) \cdot P(\text{claim} \mid \text{SPAM}) \cdot P(! \mid \text{SPAM}) \\ &= 0.50 \cdot 0.04 \cdot 0.14 \\ &= 0.0028 \\ P(\text{CLAIM} \mid d_5) &= P(\text{CLAIM}) \cdot P(\text{claim} \mid \text{CLAIM}) \cdot P(! \mid \text{CLAIM}) \\ &= 0.50 \cdot 0.08 \cdot 0.04 \\ &= 0.0016 \\ f(d_5) &= \max(P(\text{SPAM} \mid d_5), P(\text{CLAIM} \mid d_5)) \\ &= \text{SPAM} \end{aligned} \tag{2.33}$$

As such, the text "claim!" is classified as spam.

Chapter 3

Annotation of Spreadsheet Cells

To find a suitable feature representation of textual cell values in insurance-claim-related spreadsheets – to use for determining the concepts of cell values, I gathered 1,216 Excel spreadsheets from actual Norwegian property claims at Protector Forsikring.

The first issue I encountered with the data was to retrieve the formatted values (i.e., the value of a cell displayed by Excel) in the spreadsheets. The packages for processing Excel files with Python only seemed to provide functionality for retrieving the actual values and not textual values (e.g., the formatted value "100 %" has a value of "1.00"). I tried implementing the extraction of formatted values using an API for the C# programming language, however, the performance was poor as it required an instance of Excel running. I ended up using the Apache POI library for Java, which provided a class for formatting the cell values (see Section 3.2). I processed the spreadsheets by converting the raw files to structured JSON-objects, following a standardized structure, where the attributes of each cell were easily accessible for large-scale use. Although my initial plan was to use unsupervised learning to learn a feature representation for the textual cell values, where keeping the table structure within spreadsheets would enable to utilize the relations between cells, personal data had to be removed from the collected data due to privacy regulations (i.e., GDPR).

The privacy issue was, therefore, an incentive to create an annotated data set – which can be used for concept determination of cells in semi-structured spreadsheets related to insurance claims, as data such as, person names, phone numbers, email addresses, etc., had to removed. Given how the spreadsheet files were stored in the system, the collection contained an unknown number of duplicates, but as only the textual cell values were needed to create an annotated data set, the values were transformed into a set of values (i.e., the JSON-objects were not needed so the issue of duplicates is not present in the final data set). As there were over 200,000 unique values in the raw data, I implemented different approaches to streamline the annotation process. In order to collect a data set of values annotated with concept labels (described in Table 3.1), the first phase consisted of using a rule-based classifier to gather some initial samples of the different concepts. By using the rule-based classifier, common values (i.e., numeric values with common formats, such as dates, time, percent, money, etc.), which were expected to make up a large part of the unprocessed values, were annotated with decent accuracy by using strict rules, while concepts with less strict rules (e.g., organisations and locations) required frequent intervention of manually correcting the annotated concepts. I explain the rule-based classifier in Section 3.4. Having obtained samples for most concepts using the rule-based classifier, I trained a multinomial naïve Bayes classifier to further annotate more samples with the objective of having more accurate annotations based on the accumulated data set instead of relying on hand-written rules – decreasing the need for manual correction of annotations (see Section 3.5). As both classifiers were not trusted to annotate the correct concept of a value, each annotation was manually supervised and corrected when needed. The resulting data set contains 119,963 unique values with an annotated concept (Table 3.2 shows the distribution per concept).

Label	Concept	Description
TIME	Time	E.g., “12:00:00”, “kl. 12:00”
CARDINAL	Cardinal	General number without a concept (e.g., “10,000”)
PROD	Product	Named product, e.g., “Grandiosa Pizza pepperoni”
MONEY	Money	Number with a currency code/symbol (“kr 10”, “10 NOK”)
DATETIME	Date and time	E.g., “01.04.2021 12:00”, “1. april kl. 12”
DATE	Date	E.g., “1. april”, “1/04/2021”, “01.04.2021”
TEXT	Text	General text not captured by any of these concepts
ORG	Organisation	A company, municipality (if suffix is “kommune”), etc.
LOC	Location	A location, e.g., address, administrative division, etc.
PERCENT	Percent	A percentage (e.g., “10 %”)
FAC	Facility	Named facilities, such as schools, locatable stores, etc.
QNT	Quantity	A number with a measurement symbol (e.g., “16 kg”, “1 stk”)
EVT	Event	Text with several concepts (i.e., who, what, where, when)

Table 3.1: Data set concepts.

	Training Set	Test Set	Sum	Distribution
TIME	33,707	8,427	42,134	35.12%
CARDINAL	28,098	7,025	35,123	29.28%
PROD	9,602	2,401	12,003	10.01%
MONEY	5,813	1,454	7,267	6.06%
DATETIME	5,578	1,395	6,973	5.81%
DATE	4,680	1,170	5,850	4.88%
TEXT	3,371	843	4,214	3.51%
ORG	1,494	374	1,868	1.56%
LOC	1,428	358	1,786	1.49%
PERCENT	972	243	1,215	1.01%
FAC	868	218	1,086	0.91%
QNT	274	69	343	0.29%
EVT	80	21	101	0.08%
Sum	95,965	23,998	119,963	100.00%

Table 3.2: Data set distribution.

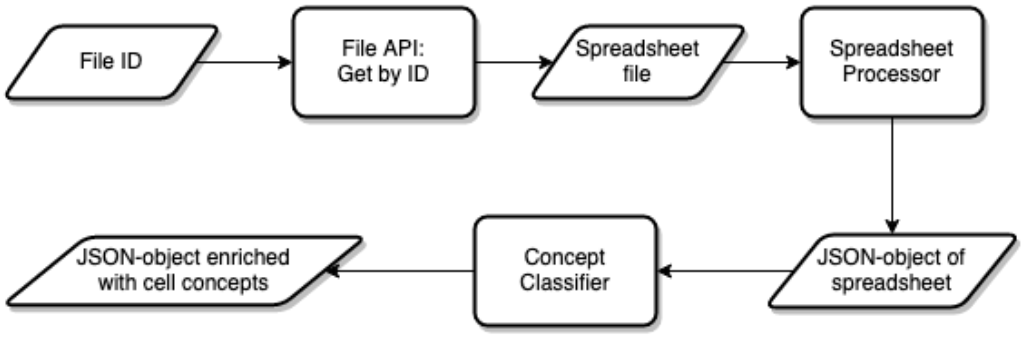


Figure 3.1: Pipeline for enriching a spreadsheet with concept information of cells.

3.1 Spreadsheet Enrichment Pipeline

In this chapter, I explain how I annotated cell values in spreadsheets to obtain a data set that can be used for supervised classification of concepts. The pipeline illustrated in Figure 3.1 shows how a system would process a raw spreadsheet file and enrich the spreadsheet with concepts of cells. At Protector Forsikring, a file is stored in a file storage and is attached to claim using a unique identifier, and can be accessed through an API. In the pipeline, the first step is to retrieve a spreadsheet file from the API. The spreadsheet file is then processed by structuring the attributes of the cells in a standardized file (JSON-object, i.e., mapping from keys to values), where the cell values have also been formatted according to how Excel displays the values (the spreadsheet processing is described in Section 3.2). To enrich the spreadsheet with concept information, each cell's formatted value is passed into a text classifier (trained on annotated data) which outputs a concept. The concept of a cell is then added as an attribute in the standardized file representation of a spreadsheet. In Sections 3.4-3.5, I present two different text classifiers that I used to streamline the annotation process (i.e., so that I could efficiently collect samples for each concept), and I also used the multinomial naïve Bayes (Section 3.5) as a baseline for concept classification.

3.2 Processing Excel Spreadsheets

I collected the spreadsheets from Norwegian property claims, i.e., the files are attachments in actual claims, where the spreadsheets have been created by an involved party (e.g., claimant or surveyor) of the insurance event that the claim concerns. Property claims can be related to, for instance, damage of a building, inventory or equipment, so the content and structure of the spreadsheets vary between the different claims. The file type of the spreadsheets indicated that they had been created using Excel, which stores the cell values but have functionality for displaying the values differently in the application. For instance, when setting the format of a cell to percent in Excel, the value in the spreadsheet is stored as a fraction of 100 (e.g., "1.00") while the value displayed in the cell is the value multiplied by 100 with the percent symbol as suffix (e.g., "100 %"). Excel uses rules to define the these cell formats (see Example 1).

Example 1. Excel formats are split by ";" into four components, where 1) defines how to format positive numbers, 2) negative numbers, 3) zero, and 4) text.[1] For instance, the following format defines how to format and display numeric values as money with the currency symbol for Norwegian kroner ("kr"):

$$\text{"kr"}\backslash\#,\#\#0.00;[\text{Red}]\backslash-\text{"kr"}\backslash\#,\#\#0.00;;$$

[1]Excel number formatting:
https://support.microsoft.com/en-us/office/number-format-codes-5026bbd6-04bc-48cd-bf33-80f18b4eae68

Key	**Value**
address	A1
type	STRING
format	General
formula	
value	Maskinforsikring:
formattedValue	Maskinforsikring:
concept	*TEXT*

Table 3.3: JSON-object of a cell with string value (*concept* is set by a concept classifier in the full pipeline).

Using this format, Excel will display "kr 10,000.00" if the value is "10000.0" and "- kr 10,000.00" (in addition to being colored red) if the value is "-10000.0". As the components for zero and text are empty (no specified format), zeroes and texts are displayed without any formatting. These rules can be far more complex, which make them difficult to parse.

In order to apply natural language processing techniques on the cell values in spreadsheets, I needed to have the formatted value and not the actual value. While the formatted values are used for displaying numeric values in a more human-readable format, they provide some context of numerical concepts. At first, I investigated different packages for processing Excel files with the Python programming language. However, none of the packages seemed to provide functionality for extracting formatted values from Excel files.[2] A promising solution was to use an API available for the C# programming language, but the API relied on communicating with a running instance of the Excel application.[3] As the API for C# directly depends on Excel, it had performance issues which caused the spreadsheet processing phase to take several days to complete. In the C# program that I wrote for processing the spreadsheets, a new instance of Excel had to be started for each file to process, and I observed that the program only exited some of the instances successfully after the processing was finished. Due to the small amount of running instances that it took before running out of memory, the process had to be closely monitored – leading to long processing time as I had limited time to spend staring at the program running. I later found a library for the Java programming language, Apache POI[4], that provides functionality for formatting stored values in Excel spreadsheets. As I have more experience with Java than C#, in combination with a more usable library than the C# API, the processing program written in Java was more efficient. The processing program was written to a) download an insurance claim spreadsheet from the internal systems at Protector Forsikring – based on a predefined list of files to collect, b) load the file into a Workbook-object (class for spreadsheets in Apache POI), c) map the spreadsheet to a structured JSON-object, and d) repeat the steps from (a) until there were no files left to process. After a few hours, the spreadsheet files were stored as separate JSON-objects and ready for further processing in large-scale.

Each JSON-object of a spreadsheet were structured as a list of sheets (an Excel file can contain one or more spreadsheets, i.e., sheets), where each sheet was a list of cells structured as JSON-objects. A cell had the following attributes: cell address (i.e., column letter concatenated

[2]Overview of packages for working with Excel in Python: http://www.python-excel.org/

[3]Microsoft.Office.Interop.Excel documentation:
https://docs.microsoft.com/en-us/dotnet/api/microsoft.office.interop.excel?view=excel-pia

[4]Java API for Excel:
https://poi.apache.org/components/spreadsheet/

Key	**Value**
address	I4
type	NUMERIC
format	m/d/yy
formula	
value	44090.0
formattedValue	9/16/20
concept	*DATE*

Table 3.4: JSON-object of cell with numeric value and specified string format (*concept* is set by a concept classifier in the full pipeline).

with row number), type (i.e., string, numeric or formula), format, formula, value, and formatted value (extracted using the DataFormatter-class in Apache POI). In Table 3.3-3.4, examples of the JSON-object of cells are illustrated. The reason for structuring the spreadsheets as JSON-objects was that I could use the attributes of a cell in further processing. However, due to privacy regulations (i.e., GDPR), values such as person names, phone numbers and email addresses had to be removed, and I made the decision of creating a data set where values are annotated with concepts. Therefore, I only needed the formatted value from the JSON-objects, which were extracted and added to a set – resulting in a collection of more than 200,000 unique textual values, before annotating a subset of the values with concepts.

3.3 Concepts in Insurance-Claim-Related Spreadsheets

In the context of this thesis, concepts are an abstraction of what the facts (i.e., values) in the spreadsheets represent in the real world; i.e., they are more narrow and descriptive data types than, e.g., "string" and "numeric". As inspiration, I used some of the entity types that are commonly used for named entity recognition (NER) together with concepts that I discovered when looking at the data. For instance, NorNE [5] used common annotations such as organisation and location; however, as they annotated text from a news corpus (i.e., a collection of sentences gathered from news articles), they used a separate entity type for geo-political entities (GPE) to describe entities such as "Lier kommune" (English: "Lier municipality"). While news articles tend to be more politically oriented, in an insurance perspective, GPEs are regarded as either organisations (customers or third parties) or locations (e.g., to describe location of damage or insured objects). With that in mind, the value "Lier kommune" is annotated as an organisation (i.e., ORG) in the data set. Although the most interesting use is to be able to determine concepts of strings (i.e., organisation, location, etc.), the spreadsheets consists largely of numeric values. Therefore, I defined labels for numeric values, such as, quantity, money, percent, and date/time (see Table 3.1 for description of all concepts present in the data set).

Ideally, the concepts should be separable so that the values can be structured in, e.g., a table column with no ambiguity (i.e., a column only contains values of a specific concept which can be utilized using a method which applies to all the column values), e.g., "Europris AS" can only be interpreted as an organisation name; however, there are some overlap between the used labels. For instance, in order for a value to be annotated as *DATETIME*, both date and time must be present (illustrated in Figure 3.2). Further, the *EVT* label is assigned to values that describe an insurance event, but as the present concepts varies between the different values annotated as *EVT*, it is the concept that caused most uncertainty during the annotation process. It was

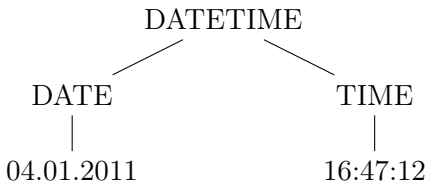


Figure 3.2: Concepts present in the string “04.01.2011 16:47:12”.

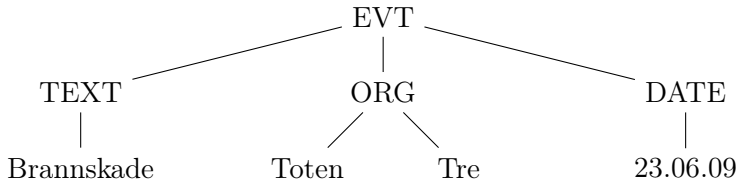


Figure 3.3: Concepts present in the string “Brannskade Toten Tre 23.06.09”.

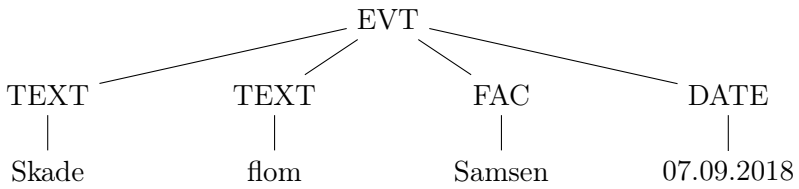


Figure 3.4: Concepts present in the string “Skade flom Samsen 07.09.2018”.

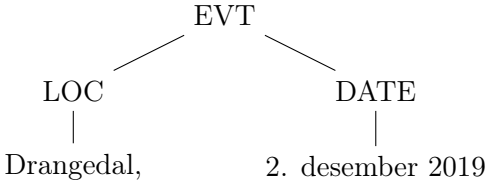


Figure 3.5: Concepts present in the string “Drangedal, 2. desember 2019”.

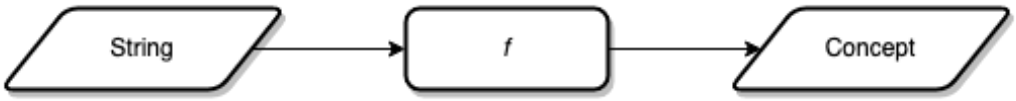


Figure 3.6: Flow of the rule-based classifier.

during the annotation process that I fine-tuned my own guideline for annotating values as *EVT*. The guideline I eventually used was to annotate a value as *EVT* if two or more of these are present in the value: what, who, where, and when. Figures 3.3-3.5 illustrates different concepts contained under the *EVT* label:

- Figure 3.3: Toten Tre had a fire damage on 23.06.09.
- Figure 3.4: Samsen (cultural facility in Kristiansand) was affected by a flood on 07.09.2018.
- Figure 3.5: an unknown event happened in Drangedal on December 2. 2019.

When no specific concepts were applicable to a value, the value was annotated as either *TEXT* (string without a specific concept) or *CARDINAL* (numeric without a specific concept).

As I annotated values from a set of values, and not directly from spreadsheets, I had no context for the different values. As a result, I think this improved the process as I could not use any other information than what was present in the value in order to determine its concept. In an attempt to streamline the annotation process, I tried two different approaches for automatically assigning labels, where I used a rule-based classifier to initiate the annotation process.

3.4 Rule-Based Annotation

I started annotating values in the set by having a rule-based classifier that I used to get suggestions for values (samples) that fit a specific concept – based on the syntax. Figure 3.6 illustrates the flow of the rule-based classifier: a string is given as input, the classification function f computes the match ratio between the string and a concept (based on a syntactic rule), and returns the concept with the highest ratio. The flowchart in Figure 3.6 can be seen as the internal process of the *Concept Classifier* in Figure 3.1. The idea for implementing the rule-based classifier was so that I was able to gather samples for each concept that I had defined – as manually iterating through 200,000 values would require a lot of time and would not guarantee finding samples for each concept, so that I would have enough samples to use machine learning for further improving the accuracy of each annotation. I defined a pattern for each concept, except for *PROD*, *FAC* and *EVT* as they are difficult to capture using hand-written patterns (see Table 3.5). In addition, I defined patterns for concepts that are not included in the final data set but had to be removed from the data set due to privacy regulations (i.e., personal data such as phone numbers, bank account information, personal identifiers, vehicle registration numbers, email addresses, and person names). This method helped me obtain samples for each concept as I specified a concept that the classifier should extract from the unprocessed samples (i.e., I used the classifier to show me samples that had a highest match with a given pattern, e.g., organisation), where I chose a concept with low samples in the accumulated data set of annotated samples. Although I did not define patterns for *PROD*, *FAC* or *EVT*, the classifier returned samples that were annotated as a concept with a weak pattern, but rather matched one of the three patterns (e.g., a lot of samples matched the defined pattern for person names, but were mostly products). As a consequence, I obtained samples of every concept as I knew about the limitations of having weak patterns, I manually supervised each annotation and corrected the annotations when needed.

Label	**Regular Expression**
TIME	$\backslash s*\backslash d\{1,2\}:\backslash d\{2\}:\{2\}\backslash s*$
CARDINAL	$\backslash s*[\text{-}\|-]?\backslash s*(\backslash d+[\backslash s.,]*)+[,.]?(\backslash d+)*\backslash s*$
MONEY	$\backslash s*[\text{-}\|-]?\backslash s*((kr\|KR)\backslash s*[\text{-}\|-]?\backslash s*(\backslash d+[\backslash s.,]*)+[,.]?(\backslash d+)*$ $\|(\backslash d+[\backslash s.,]*)+[,.]?(\backslash d+)*\backslash s*(nok\|NOK\|euro\|EURO\|USD))\{1\}\backslash s*$
DATETIME	$\backslash s*\backslash d\{1,2\}[/.-]\backslash d\{1,2\}[/.-]\backslash d\{2,4\}\backslash s+\backslash d\{1,2\}:\backslash d\{2\}:\backslash d\{2\}\backslash s*$
DATE	$\backslash s*\backslash d\{1,2\}[/.-]\backslash d\{1,2\}[/.-]\backslash d\{2,4\}*$
ORG	$\backslash s*((\backslash w-]+\backslash s+)\{1,2\}(as\|AS\|A/S\|ab\|AB\|KF\|kf)$ $\|([A\text{-}Z\text{ÆØÅ}a\text{-}z\text{æøå}]+\backslash s+kommune))\{1\}\backslash s*$
LOC	$\backslash s*((\backslash w-]+\backslash s*)\{1,3\}\backslash s*(vei\|veien\|veg\|vegen\|gate\|gata$ $\|gaten\|vn\|vg\|gt)[,.-]?\backslash s*\backslash d*\backslash s*[A\text{-}Z\text{ÆØÅ}a\text{-}z\text{æøå}]?)?$
PERCENT	$\backslash s*[\text{-}\|-]?([\backslash s.,]*\backslash d+)+([,.]\{1\}\backslash d+)?\backslash s*\%\backslash s*$
QNT	$\backslash s*[\text{-}\|-]?\backslash s*(\backslash d+[\backslash s.,]*)+[,.]?(\backslash d+)*\backslash s*$ $(kg\|g\|stk\|stykk\|m\|cm\|dm\|mm\|l\|dl\|cl\|ml\|r\|dager)\backslash s*$
PHONE	$\backslash s*(\backslash +\backslash s*47)?(\backslash s*\backslash d\{1\})\{8\}\backslash s*$
BANK	$\backslash s*(\backslash d\{4\}[.\backslash s]*\backslash d\{2\}[.\backslash s]*\backslash d\{5\}\|(NO\|no)\{1\}(\backslash s*\backslash d\{1\})\{13\})*$
PERSON_ID	$\backslash s*[0-3]\{1\}[\backslash s.]*\backslash d\{1\}[\backslash s.]*[0-1]\{1\}(\backslash d\{1\}[\backslash s.]*)\{8\}\backslash s*$
VEHICLE_ID	$\backslash s*[A-Z]\{2\}\backslash s*[0-9]\{4,5\}\backslash s*$
EMAIL	$\backslash s*([\backslash w.-]+)\{1\}@\{1\}(\backslash w+.*-*)1,\backslash.\backslash w\{2,\}\backslash s*$
PERSON	$\backslash s*([A\text{-}Z\text{ÆØÅ}]\{1\}[A\text{-}Z\text{ÆØÅ}a\text{-}z\text{æøå}]+\backslash s*)\{2,\}\backslash s*$

Table 3.5: Rules in the form of regular expression (patterns) for concepts (labels). Labels in **bold** were removed from final data set.

Definition 3.4.1 (Pattern scoring function) *The score of a pattern is defined as the ratio between the length of the sub-string of s that matches pattern p and the total length of s:*

$$score(s, p) = \frac{end(s, p) - start(s, p) + 1}{length(s)} = \frac{j - i + 1}{length(s)} \tag{3.1}$$

where j is the last index and i is the first index of the sub-string $s_{i,j}$ of s that matches the pattern p.

Example 1. In the string in Figure 3.2, s = *"04.01.2011 16:47:12"*. Using the patterns defined in Table 3.5, the *DATE*, *TIME* and *DATETIME* patterns have the following scores:

$$\begin{aligned} score(s, DATE) &= \tfrac{10-0+1}{19} = \tfrac{11}{19} = 0.58 \\ score(s, TIME) &= \tfrac{18-10+1}{19} = \tfrac{9}{19} = 0.47 \\ score(s, DATETIME) &= \tfrac{18-0+1}{19} = \tfrac{19}{19} = 1.00 \end{aligned} \tag{3.2}$$

The scoring function enables us to set a probability for each pattern based on how much of the string matches a given pattern. By having different patterns, we can use the computed score to determine a label based on rules, e.g., in this example, s will be annotated as *DATETIME*.

While the rules for numeric-based concepts are strict (i.e., few syntactic differences between values of same concept makes it feasible to write rules that capture most of the values), the rules for string-based concepts (e.g., organisation, location, person name) are difficult to capture using hand-written rules. Therefore, some of the patterns that I defined will overlap, i.e., the score is equal for two or more patterns on the same input. For instance, the rule for organisation (defined in Table 3.5) will match any string that contains one or two words followed by an abbreviation of a company form (e.g., AS, KF) or a single word followed by "kommune", while the rule for person name requires two or more words that starts with an uppercase letter. As a consequence, the scores of *ORG* and *PERSON* will be the same given the input "Organisation AS".

$$score(\text{"Organisation AS"}, ORG) = score(\text{"Organisation AS"}, PERSON) \tag{3.3}$$

In order to overcome this issue, I used the weighted scoring function (Definition 3.4.2) to give the pattern for *ORG* higher precedence (as the rule for *ORG* is stricter than the rule for *PERSON*). To give *ORG* higher precedence than *PERSON*, I set the static weight of *PERSON* to 0.9. The default static weight that I assigned to the different patterns was 1.0, however, the static weight of *CARDINAL* was also lowered to 0.9 as this pattern could overlap with *PHONE*, *DATE*, *BANK* or *PERSON_ID*.

Definition 3.4.2 (Weighted pattern scoring function) *The weighted scoring functions multiplies the pattern score with a static weight:*

$$weighted(s, p, w) = score(s, p) \cdot w \tag{3.4}$$

Definition 3.4.3 (Text scoring function) *The TEXT label is scored by the ratio between the number of characters that do not occur in any patterns and the total length of s:*

$$score_{text}(s, P) = \frac{length(s) - |\{i \mid p \in P \wedge i \in [start(s, p) \dots end(s, p)]\}|}{length(s)} \tag{3.5}$$

where P is a set of patterns.

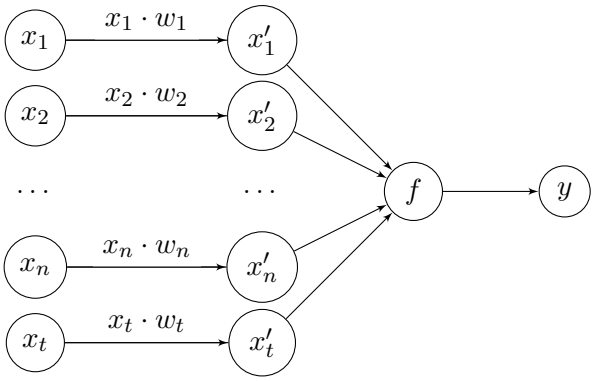


Figure 3.7: Architecture of the rule-based classifier.

Example 2. To continue Example 1, let string $s =$ *"04.01.2011 16:47:12"*. Given the set of patterns $P = \{DATE, TIME, DATETIME\}$, then the score of the *TEXT* label is 0.0 for s:

$$score_{text}(s, P) = \frac{19 - |\{0, 1 \dots 18\}|}{19} = \frac{19 - 19}{19} = \frac{0}{19} = 0.00 \tag{3.6}$$

as all the characters of s are captured by a pattern in P.

The rule-based classifier scores a label based on the label's defined pattern using the weighted scoring function (Definition 3.4.2), where the weight is a static weight assigned to the label, and classifies the input as the label with highest score. First, the input string is featurized to a vector x of size $n + 1$, where n is the number of patterns, x_i is the score between the input and pattern referenced by i, and x_{n+1} is the text score of the input string. Then, each feature x_i is weighted by a static weight assigned to feature at index i, resulting in a new vector x'. Finally, the vector x' is passed into a function f which returns the index of the feature with the largest value, i.e., if x'_i is the largest value in x', then $f(x') = i$. The classifier's architecture resembles the architecture of a feed-forward neural network – although, not fully connected, as seen in Figure 3.7; however, the classifier does not learn the weights based on sample data. As I had collected enough annotated samples using the rule-based classifier, I advanced the annotation process by using a classification algorithm that has the ability to learn to classify based on sample data, with the objective of having more accurate annotation suggestions. I chose the Naïve Bayes classifier, as the algorithm does not require a lot of fine-tuning of parameters in the same scale as neural networks.

3.5 Multinomial Naïve Bayes Annotation

I trained a multinomial naïve Bayes classifier (MNB) to classify text as one of the predefined concepts using data gathered with the rule-based classifier (RB). The flow of the classification with the MNB is similar to the flow in the RB, where the difference (besides the function itself) is that the string is converted to a vector with numerical values and used as input in the classification function (see Figure 3.8). As one objective of this thesis is to have a new annotated data set and see how different feature representations for a text classification task affects the results, I decided to use a MNB as it is a good baseline for text classification (e.g., as it models the distribution of the data, see Section 2.6.3, it is deterministic given that the data does not change). For the RB, the input was not preprocessed as I used some of the raw syntax in the patterns to determine matches, e.g., I did not use case-folding as uppercase letters are included in the defined patterns, and the raw samples were used as input. However, in order to use the MNB, I needed to tokenize the samples so that they would have a numerical representation.

The standard approach for feature representation for MNB text classification, is to represent each feature by a unique token (also called term), where the MNB has a vocabulary which

Figure 3.8: Flow of the multinomial naïve Bayes classifier.

contains each unique token that exists in the training data (i.e., bag-of-words representation, see Section 2.2). The MNB then learns the conditional probability for each token (see Section 2.6.3), where the probability is the likelihood of a token given a class (label). In order to tokenize the samples, I used a character n-gram tokenizer (Algorithm 1), where I set $n = 4$.

Algorithm 1: Character n-gram tokenizer

Input: String s to tokenize and max character sequence length n
Precondition: length(s) > 0
L $\longleftarrow$ empty list; i $\longleftarrow$ 0;
do
  L.append(s[i:min(i + n, length(s)]);
  i $\longleftarrow$ i + 1;
while *length(s)* $<= i + n$;
Output: L

The reason for why I used character n-grams as tokens was mostly due to the data. As the data have been created by humans, some samples in the data set contain spelling errors. For instance, the sample "Protektor" (annotated as *ORG* in the data set) is most likely "Protector" with wrong spelling. By using character n-grams, the features are sub-word information and, therefore, both "Protektor" and "Protector" will share features (depending on n) – instead of being individual features. However, the value of n needed to be set so that there was some balance between capturing similar features in text and avoid that the majority of numerical samples were individual features. By setting $n = 4$, the classifier was able to find similarities in text and numerical samples of similar concept, i.e., numeric values with shared suffix or prefix (see Equation 3.7).

$$\begin{aligned} ngrams(\text{“Protektor”}) \cap ngrams(\text{“Protector”}) &= \{\text{“Prot”, “rote”}\} \\ ngrams(\text{“235,20 NOK”}) \cap ngrams(\text{“121,50 NOK”}) &= \{\text{“0 NO”, “ NOK”}\} \\ ngrams(\text{“100 cm”}) \cap ngrams(\text{“120 cm”}) &= \{\text{“0 cm”}\} \end{aligned} \tag{3.7}$$

Although the character n-grams capture similarities between similar concepts (i.e., classes), there was an issue where there were similarities across different concepts. Specifically, some samples annotated as *PROD* overlap with, e.g., samples that are annotated as *QNT*. For instance, both the samples "100 cm" (*QNT*) and "3 år" (*QNT*) are found as part of other samples that are annotated as *PROD*:

"HP 3 år neste virkedag Modular Smart Array 2000-array HW-stø"
"HILTON speil, 60x100 cm. Sølv"

and as a result, I observed that the MNB eventually did not annotate any unprocessed samples as *QNT*. The MNB provided more accurate annotations – compared to the RB (including for the concepts *PROD*, *EVT* and *FAC*, which are not covered by defined patterns), however, the number of samples annotated with rare concepts (e.g., *EVT*, *QNT*, *FAC*) was reduced to zero after a few iterations. Therefore, it seems that the distribution of the data (i.e., 35.12 % of the samples are annotated as *TIME* while 0.08 % are *EVT*) and overlap of features across concepts are a major limitation of the MNB for this data set.

Chapter 4

Classification of Concepts

In this chapter, I present the different methods that I used for classifying the concept of the annotated samples that I gathered (see Chapter 3). First, I provide an overview of the tools that I used, where two different approaches for feature representation of textual values are presented (Section 4.1). In order to test the two different approaches for feature representation, I used a k-nearest neighbors (k-NN) classifier (see Section 2.6.2 for explanation of the algorithm), where the class of a given input is selected based on the classes of its nearest neighbors. This allowed me to see whether the feature representations grouped similar concepts close to each other in vector space. I also included a preprocessing stage where the raw samples were turned into normalized strings (explained in Section 4.2), where the aim was to be able to see how preprocessing the samples affected the different feature representations. The *tqdm* package (package for tracking progress of iterations in Python) indicated that classifying each test sample (23,998 number of samples) using k-NN would take more than 14 hours. Therefore, by sacrificing accuracy for efficiency, I implemented a naïve approach for representing each individual concept by a single instance (see Section 4.3), thereby reducing the number of comparisons per test sample down from 95,965 (samples in training set) to 13 (number of concepts). However, as I later found out that the dot product between normalized vectors gives the same result as the cosine similarity (used as distance metric between the vectors, see Section 2.5.3 for definition), I was able to reduce the runtime of classifying the test samples using k-NN down to roughly 2 hours and 40 minutes. The results of both k-NN on full test set and on single instance per concept (with the two different methods for feature representation) is given later in Chapter 5, where they are compared to the multinomial naïve Bayes baseline classifier from Chapter 3.

4.1 Tools

4.1.1 Programming Language

I used the Python programming language (version 3.7) for the experiments regarding classification of concepts. Although I used Java for processing the spreadsheet (Section 3.2), I decided to use Python in the experiments where I utilized the data as there exist several packages for machine learning (ML). Popular packages for ML, such as PyTorch[1], are implemented in the C programming language (due to performance) and are provided as wrappers for Python (having a more readable code syntax than C). Using such packages decreases the development and setup time without compromising the performance to a large extent. The main dependencies for the experiments are (1) *tqdm* (for tracking the progress of iterations), (2) *regex* (package for using regular expressions to process strings; used in the rule-based classifier and for preprocessing raw samples), (3) *FastText* (see Section 4.1.2), (4) *PyTorch* (see Section 4.1.4), and (5) *Sentence*

[1] PyTorch: https://pytorch.org/

Skip-gram	**CBOW**
“gåten”	“Megi”
“gåte”	“EUR”
“gåtene”	“1.757”
“gåter”	“SEK”
“gåtefull”	“85,50”
“gåte»”	“USD”
“gåtefulle”	“MNOK”
“gåtefullt”	“JPY”
“mordmysterium”	“Likkavæl”
“mysteriefortelling”	“435,00”

Table 4.1: Differences between the skip-gram and CBOW model by FastText: top 10 nearest words of the word “NOK” (found by the *get nearest neighbors*-function).

Transformers (see Section 4.1.3). The packages were installed using the *pip* command from Python's package installer.[2]

4.1.2 FastText: Pre-Trained Word Embeddings

```
/path/to/project> pip install fasttext==0.9.2
```

Listing 4.1: Installation of FastText dependency.

I used the FastText package to obtain pre-trained word embeddings for the Norwegian language. FastText provides two different word embedding models: (a) continuous skip-gram trained on character n-grams from Wikipedia[3] data [1], and (b) continuous bag-of-words (CBOW) trained on character n-grams from both Wikipedia and Common Crawl[4] (CC) data [2]. Both models embed words as a 300-dimensional vector, constructed using the trained embeddings for character n-grams (i.e., a word is considered as a bag-of-character n-grams). In contrast to other word embedding models, FastText uses n-grams to obtain sub-word information, causing the models to be more resilient against spelling errors and causes words to rarely be out-of-vocabulary (i.e., if a word is out-of-vocabulary in a word embedding model, then: embedding(word) $= \vec{0}$).

As the CBOW model uses CC data in addition to Wikipedia data, there are naturally some differences compared to the skip gram model. Table 4.1 shows the top ten nearest neighbors of the word “NOK”, which is the currency symbol for Norwegian kroner. As the concept of “NOK” is money, ideally, the top ten nearest should also be related to the concept of money. As seen in Table 4.1, the CBOW model returns other currency symbols (i.e., “EUR”, “SEK”, “USD”, “JPY”) while the skip-gram returns different forms of the word Norwegian “gåte” (English: “riddle”). Although the CBOW model might seem better for the task of classifying concepts, I included both models in the experiment in order to compare the different models with the composition method for obtaining numerical representation of text containing several words (i.e., sentence embeddings – see Section 2.4.2).

[2]Python package installer: https://pypi.org/project/pip/

[3]https://www.wikipedia.org/

[4]https://commoncrawl.org/

I used FastText to obtain sentence embeddings by the composition method. FastText provide the function *get sentence vector*, which returns a sentence embedding based on the embeddings of the sentence's words. The function finds the sentence embedding by; (1) tokenizing the sentence by whitespace (e.g., "Brannskade Toten Tre 23.06.09" is tokenized to a list of the words "Brannskade", "Toten", "Tre" and "23.06.09"), (2) finding the word embedding for each token, and (3) computing the average vector of the normalized word embeddings. This results in composed embeddings (using mean as pooling strategy) for sentences. For the distributed method, I used sentence embeddings that have been pre-trained by Sentence-BERT.

4.1.3 Sentence Transformers

```
/path/to/project> pip install sentence-transformers==1.1.0
```

Listing 4.2: Installation of Sentence Transformers dependency.

The *Sentence Transformers* (ST) package for Python includes pre-trained sentence embeddings, i.e., it provides sentence embeddings using the distributed method. While the composition method aggregates the embeddings of smaller components to represent larger data (e.g., aggregating word embeddings to sentence embeddings), the distributed method has the objective of training embeddings (see Section 2.4.2). The model from ST that I used (*paraphrase-xlm-r-multilingual-v1*) is trained for multiple language, including Norwegian.

For classification using the sentence embeddings from ST, I used the *semantic search* function (from the *util* module of ST) to select the top five nearest samples (using cosine similarity as distance metric) from the training set. The predicted class is then the class with most samples among the top five nearest, i.e., the classification is k-nearest neighbors, where $k = 5$. I set $k = 5$ because the *EVT* concept only has 80 training samples (see Table 3.2) and I expect this concept to have the most spread samples in vector space, so that requiring too many samples to be close would lead to no samples being classified as *EVT* during test runs. Also, k needed to be large enough to tolerate that some of the nearest neighbors were annotated as a different concept than the expected concept.

4.1.4 PyTorch: Open-Source Machine Learning Framework

```
/path/to/project> pip install torch==1.7.1+cpu
```

Listing 4.3: Installation of PyTorch dependency.

In addition to functionality for building machine learning models, PyTorch provides functions for different vector (i.e., tensor) operations. As PyTorch is implemented in C, the performance is expected to be significantly better than a pure Python implementation. Therefore, I used PyTorch's functions for computing the cosine similarities and dot product between vectors. Although packages such as *numpy* also provide functionality for vector operations and are implemented in C, I used PyTorch as it has the ability switch between CPU and GPU computation. However, I was not able to conduct the experiments on a GPU (all experiments was done locally with a Intel Core i7-8565U CPU) but there is a possibility to reproduce the experiments using a GPU instead without a need for making significant changes to the implementations.

4.2 Preprocessing

I wanted to see how preprocessing the cell values of insurance-related-claim spreadsheets affected the resulting accuracy of concept classification. The functions that I used were functions that I had implemented in the early stages of this thesis, where the functions that were used are for

canonicalizing strings (used during the creation of the data set), tokenization, normalizing (on a token level), and a preprocessing function used to normalize raw strings. As the objective was to see the effect of preprocessing the strings before getting the embedding of the string, I implemented them rather simple but so that the effect would be more visible.

The first function, *canonicalize*, is used to convert a raw string into a string where consecutive whitespace have been reduced to a single whitespace, and preceding and trailing whitespace have been removed (see Listing 4.4). As the function only removes unnecessary whitespace (i.e., preceding and trailing whitespace, and consecutive whitespace does not affect the semantics), the function was first used in the data collection phase in order to remove duplicated strings that only differed by whitespace. Therefore, all samples have been processed by this function.

```
def canonicalize(string: str) -> str:
    return regex.sub(r"\s+", " ", string).strip()


def tokenize(string: str) -> List[str]:
    s = regex.sub(r"(\d+)", r" \1 ", string)
    s = "".join([c if c.isalpha() or c.isdigit()
                 else f" {c} "
                 for c in s])
    return regex.findall(r"(\S+)", s)


def normalize(token: str) -> str:
    return token.lower()


def preprocess(string: str) -> str:
    return " ".join([normalize(t)
                     for t in tokenize(string)])
```

Listing 4.4: Functions used for preprocessing strings.

To find the effect of preprocessing the raw strings, I used the *tokenize* and *normalize* functions to preprocess the raw strings (using the *preprocess* function). First, the *tokenize* function separates digits from special characters and alphabetical characters by surrounding (consecutive) digits by whitespace, e.g., "12/5/13" is converted to the string " 12 / 5 / 13 ". Next, the function separates special characters from digits and alphabetical characters, e.g., "VEDR.:" is converted to "VEDR . : ". Although the temporary string can contain consecutive, preceding and/or trailing whitespace after the two first steps, it does not affect the result, as the function returns a list of non-whitespace strings, e.g., tokenize("12/5/13") = ["12", "/", "5", "/", "13"] and tokenize("VEDR.:") = ["VEDR", ".", ":"]. The *preprocess* function is used to normalize a raw string, where it joins normalized (case-folded) tokens by a single whitespace (used whitespace as both FastText and ST tokenize by whitespace). This function adds a new step in the concept classification flow (see Figure 4.1). For instance, the string "12/5/13" is normalized to "12 / 5 / 13". However, because the preprocessing applies case folding, currency codes will be normalized to lowercase, e.g., preprocess("* 1.000 NOK") = "* 1 . 000 nok". While Table 4.1 showed that the FastText CBOW model is able to group currency codes, Table 4.2 shows that the meaning of "nok" is completely different from "NOK" in both models. The Norwegian word "nok" is translated to "enough" in English, and as Table 4.2 shows, both models group the word "nok" with other adverb words. As such, I expect that preprocessing the raw strings will lead to lower accuracy for concepts that have few, but meaningful, keywords that are ambiguous – when using k-nearest neighbors classification on sentence embeddings.

Skip-gram	**CBOW**
“så”	“likevel”
“såpass”	“ikke”
“tilstrekkelig”	“allikevel”
“neppe”	“heller”
“ikke”	“neppe”
“likevel»”	“vel”
“men”	“men”
“alikevel”	“kanskje”
“tilstrekkelig»”	“bare”
“riktignok”	“sikkert”

Table 4.2: Differences between the skip-gram and CBOW model by FastText: top 10 nearest words of the Norwegian word “nok” (found by the *get nearest neighbors*-function).

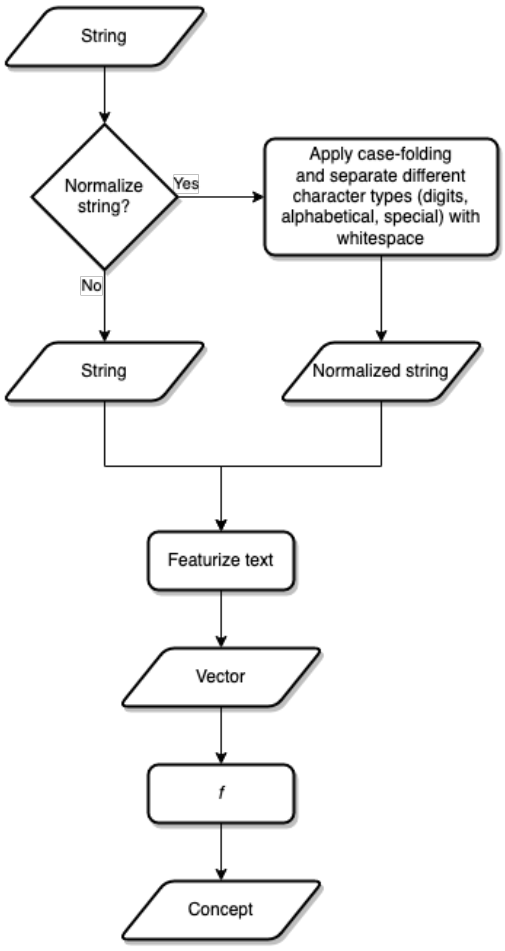


Figure 4.1: Concept classification flow with option to normalize strings.

4.3 Implementation of Indexed Concepts

As I mentioned earlier, the *tqdm* package indicated that running all 23,998 test samples through a k-nearest neighbors (k-NN) classifier would take over 14 hours to complete. Therefore, I tried to implement a method where I index the concept, so that each concept is represented by a single instance – effectively reducing the number of instances to compare against down from 95,965 to 13 (number of concepts). The *ConceptIndex*-class is initiated with a FastText model (vocabulary used to obtain word embeddings) and samples (i.e., list of strings that belong to the index's concept) – see Listing 4.5. The index value is set by aggregating each sentence embedding of the samples – using a pooling strategy (current implementation only supports using the mean embedding), which is then used by the similarity-function (returns the cosine similarity between the input and index).

```
class ConceptIndex:
    def __init__(self, vocabulary: FastText, samples: List[str]):
        self.__vocabulary = vocabulary
        self.__index = self.__build(samples)

    def __build(self, strings: List[str]) -> torch.Tensor:
        return torch.mean(torch.stack([self.string2vec(s)
                                       for s in strings]), dim=0)

    def string2vec(self, string: str) -> torch.Tensor:
        return torch.tensor(self.__vocabulary
                            .get_sentence_vector(string))

    def similarity(self, string: str) -> float:
        return torch.cosine_similarity(self.string2vec(string),
                                       self.__index, dim=0).item()
```

Listing 4.5: Concept-index class – constructed by the embeddings of its members.

In order to use k-NN with the indices, I implemented a class that holds one index per concept (see Listing 4.6). The *MultiIndex* class builds an index per concept (samples are given by a dictionary, where the key is the label, i.e., concept, and the values are lists of samples) and has a function for computing the similarities between an input and each concept index. The *predict* function is used for classification as it returns the concept of the index with highest cosine similarity to the input string (i.e., 1-NN classification).

```
class MultiIndex:
    def __init__(self, vocabulary: FastText,
                 samples: Dict[str, List[str]]):
        self.__vocabulary = vocabulary
        self.__indices = {k: Index(self.__vocabulary,
                                   samples[k])
                          for k in samples}

    def similarities(self, string: str) -> Dict[str, float]:
        return {k: self.__indices[k].similarity(string)
                for k in self.__indices}

    def predict(self, string: str) -> str:
```

```
s = self.similarities(string)
return max(s, key=lambda x: s[x])
```

Listing 4.6: Class for constructing a multi-index.

There are some options that the MultiIndex supports, such as which FastText model to use and whether to preprocess the strings (done as a separate stage before using the MultiIndex). In my experiments, I tested both the skip-gram and CBOW model from FastText (using the embeddings for Norwegian), and I also tested to preprocess the strings before creating the indices and predicting test samples. Although the current implementation does not support toggling the pooling strategy (i.e., only the mean strategy is supported), a future improvement can be to add support for, e.g., sum and max pooling strategy (I would expect sum pooling strategy to have a poor performance, as the number of samples varies between the concepts, but max pooling strategy might separate the concepts more clearly). The results of the experiments that I conducted are provided in Chapter 5.

Chapter 5

Results

In this chapter, I present the results of the data set and the experiments where I used the data set to classify concepts. In Section 5.1, I show some observations of the data set, where the main issue was data ambiguity and not having clearly defined the available concepts before starting to annotate samples – causing some lower quality of the annotations. After showing some issues with the data, I present the results of using the annotations in a classification task, where I used two different supervised learning algorithms: multinomial naïve Bayes classifier (baseline) and k-nearest neighbors classifier (Section 5.2). In addition, I provide results for using a single averaged embedding to represent a single concept, where I also tried the method with few training samples (number of samples ranging from 1 to 10) and compare it to the baseline classifier. Lastly, I provide an example of an annotated spreadsheet in Section 5.3 and discuss how a spreadsheet enrichment pipeline (consisting of the methods that I provide) can be used in an insurance perspective.

I measured the accuracy of the classification models using F_1-score (see Definition 5.0.3), where the total accuracy of a model is given as the macro average F_1-score of each concept. I used the F_1-score because using precision (Definition 5.0.1) as accuracy alone gives a skewed view of a model's performance. For instance, I found that the rule-based classifier had a precision of 100 % for the concept *DATETIME*, however, the F_1-score was 0.29 %. This was due to the model classifying only two samples as *DATETIME*, which were correctly classified (i.e., true positive = 2, false positive = 0), while it classified 15 samples as *DATE* and 1,378 as *CARDINAL* – giving a high number of false negatives (low recall).

Definition 5.0.1 (Precision) *Ratio between the number of correct classifications and total number of classifications with that class:*

$$precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{5.1}$$

Definition 5.0.2 (Recall) *Ratio between the number of correct classifications and total number of samples belonging to the class:*

$$recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{5.2}$$

Definition 5.0.3 (F_1-score) *Harmonic mean of precision (Definition 5.0.1) and recall (Definition 5.0.2):*

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{5.3}$$

5.1 Data

The data set contains samples that are cell strings from insurance-claim-related spreadsheets and are annotated with a concept, i.e., what the cell represent in the real world. The objective for

creating such a data set was that I thought having a method for classifying the concept would possibly improve the process of extracting information from spreadsheets (which are used in the insurance domain in day-to-day business operations). However, using classification of concepts for information extraction (IE) would require to know more concepts.

With only 13 concepts, the data set is quite limited for the insurance domain. Although six additional concepts were dropped from the data set, due to privacy issues, there are more concepts that should have their own annotation. Concepts that I have noticed in several samples are *invoice* and *action*.

For instance, both (a) "SYKEHUSPARTNER HF" and (b) "Fakt: 10177079 Lev: SYKEHUSPARTNER HF L-Beskr:" are annotated as organisation (*ORG*). While (a) is a legal company name (i.e., there are no other information than the official company name), (b) contains information regarding a specific invoice. I annotated (b) as *ORG* because a company name can be extracted from the text; however, it would be more accurate to annotate it as invoice, where multiple concepts are contained (e.g., invoice number, "Fakt: 10177079", and supplier name, "Lev: SYKEHUSPARTNER HF"). Other samples that should have been annotated as invoice, e.g., "Faktura Polygon (fakturanr. 1008367)" and "Faktura BBE Takst - fakturanr. 117185", were also annotated as *ORG*. As I tried to annotate each sample with a single concept, I encountered ambiguous samples frequently – which made the annotation process more challenging.

Data ambiguity occurred frequently during the annotation process. As shown above, when the samples actually were invoices, I annotated them as *ORG*. The reason for why I annotated them as *ORG* is that I prioritized annotating with a concept that indicates what the sample could be searched against in order to enrich the data (given that this method is implemented to extract information from spreadsheets). As there are company registers that offer company name search, I annotated most invoices as *ORG* (except invoices that only contained a reference number, e.g., "Deres faktura 11447 (vår ref 43228)", which is annotated as *TEXT*) as the company name can be looked up in external registers. I encountered the same issue with samples that described an event (*EVT*), where I did not introduce the concept until I had seen multiple samples that needed its own concept and, eventually, I defined *EVT* to contain information such as, what, where, who and/or when. However, as I had annotated some samples before creating the *EVT* label and had clear guidelines for the label, there are some annotated samples that should be corrected. For instance, (a) "Asfaltering v/Høgmo barnehage - utføres sommer 2010" and (b) "Rælingen kommune - skade Rud skole den 17. mars 2010" are both annotated as facilities (*FAC*: (a) "Høgmo barnehage", (b) "Rud skole"). I assume these two samples were annotated before I started to use *EVT* – as both samples contain a facility, date, and *what*. Additionally, (b) contains an *ORG* ("Rælingen kommune" – municipality in Norway) but was annotated as *FAC* because I prioritized locatable text (i.e., *FAC* and *LOC*) over party-specific text, such as organisation names. However, despite the concepts either of those contained, they both should be annotated as *EVT*. The ambiguity of the *EVT* label is best illustrated by some samples that seem to have been annotated before I had defined guidelines.

The sample "Flaum Sikring av verdiar" is annotated as *EVT* while a similar sample, "Flaum Sikring av teknisk utstyr", is annotated as *TEXT*. Both samples describe an *action* related to inventory (valuables: "verdiar", and technical equipment: "teknisk utstyr"). However, there is a subtle ambiguity which leads to both samples having two possible interpretations: (1) protection against further damage of inventory after a flood ("Flaum"), or (2) spelling error where "Flaum Sikring" should be one word and describe protection specifically against flood damage (i.e., as a preemptive measure). Given that the data is from insurance claims, interpretation (1) is probably the correct, in which case both samples should have been annotated as *action* (or something that indicates that the samples are applicable to the insurance event and affects the total damage). As the guideline I used was to annotate as *EVT* if two or more of what, where, who or when was present in the sample, having separate concepts for damage types and actions would have made it more easy to distinguish samples that are *EVT*.

Method	**Macro avg.** F_1	**Numeric**	**String**
Rule-based	52.90%	73.54%	28.83%
Naïve Bayes (character 3-gram, raw)	79.08%	86.11%	70.88%
Naïve Bayes (character 3-gram, normalized)	79.11%	87.31%	69.55%
Naïve Bayes (character 4-gram, raw)	78.08%	83.67%	71.55%
Naïve Bayes (character 4-gram, normalized)	75.76%	80.07%	70.72%
Naïve Bayes (character 5-gram, raw)	71.34%	70.07%	72.83%
Naïve Bayes (character 5-gram, normalized)	74.77%	77.14%	72.01%
5-NN (FastText CBOW, raw)	85.19%	93.77%	75.18%
5-NN (FastText CBOW, normalized)	**87.09%**	**96.97%**	**75.57%**
5-NN (FastText Skip-gram, raw)	79.57%	90.26%	67.11%
5-NN (FastText Skip-gram, normalized)	82.55%	91.74%	71.83%
5-NN (Sentence-BERT, raw)	82.26%	95.13%	67.25%
5-NN (Sentence-BERT, normalized)	80.76%	93.68%	65.67%
Multi-Index (FastText CBOW, raw)	53.37%	58.61%	47.27%
Multi-Index (FastText CBOW, normalized)	52.92%	61.17%	43.30%
Multi-Index (FastText Skip-gram, raw)	34.83%	39.01%	29.94%
Multi-Index (FastText Skip-gram, normalized)	64.78%	77.19%	50.30%

Table 5.1: Results of different methods; best score in **bold**, **Numeric** shows the accuracy of numeric concepts (*QNT, DATE, DATETIME, PERCENT, MONEY, TIME, CARDINAL*) and **String** shows the accuracy of string concepts (*ORG, TEXT, PROD, EVT, FAC, LOC*).

5.2 Classification

Despite a somewhat low quality in the annotated data, the k-nearest neighbors classifier (k-NN) – using the full training set during classification, was able to achieve a high accuracy and had a higher accuracy than the multinomial naïve Bayes classifier (MNB), which I used as a baseline. In Table 5.1, the results of the different methods are shown. I have used four different models for the classification task: rule-based classifier (Section 5.2.1), multinomial naïve Bayes classifier (Section 5.2.2), k-nearest neighbors (Section 5.2.3), and a multi index (Section 5.2.4). As the multi index achieved a surprisingly high accuracy (considering that each concept was represented by the mean embedding of all its samples), I experimented by using very few samples for training (Section 5.2.5).

The accuracy on a concept level is provided in Table 5.2, where each model is represented its best parameter setting (i.e., parameters that led to highest overall accuracy). Further, Figure 5.1 shows the confusion matrices of the models in Table 5.2. A confusion matrix illustrates the precision and recall of a model, where in Figure 5.1, the predicted concept is shown on the x-axis and the actual (gold) concept is shown on the y-axis. If the accuracy is 100 %, a confusion matrix will show a diagonal line across the matrix.

	RB	MNB(n=3, norm)	5-NN(CBOW, norm)	MI(skip-gram, norm)
TIME	99.85%	99.85%	**99.99%**	98.11%
CARDINAL	80.11%	99.03%	**99.15%**	84.61%
PROD	0.00%	90.73%	**94.04%**	80.92%
MONEY	97.61%	**98.51%**	97.93%	86.93%
DATETIME	0.29%	93.49%	**99.89%**	83.40%
DATE	92.45%	91.05%	**98.52%**	69.52%
TEXT	48.20%	72.51%	**82.64%**	48.38%
ORG	66.88%	77.29%	**82.32%**	57.79%
LOC	57.89%	83.71%	**86.02%**	60.09%
PERCENT	**98.38%**	86.65%	98.35%	84.23%
FAC	0.00%	74.30%	**83.40%**	49.77%
QNT	46.09%	42.59%	**84.89%**	34.06%
EVT	0.00%	18.75%	**25.00%**	4.84%
Macro Avg.	52.90%	79.11%	**87.09%**	64.78%

Table 5.2: F_1-scores on a concept-level of different models with optimal parameters (best score in **bold**): rule-based (RB), multinomial naïve Bayes (MNB), 5-nearest neighbors (5-NN), multi-index (MI).

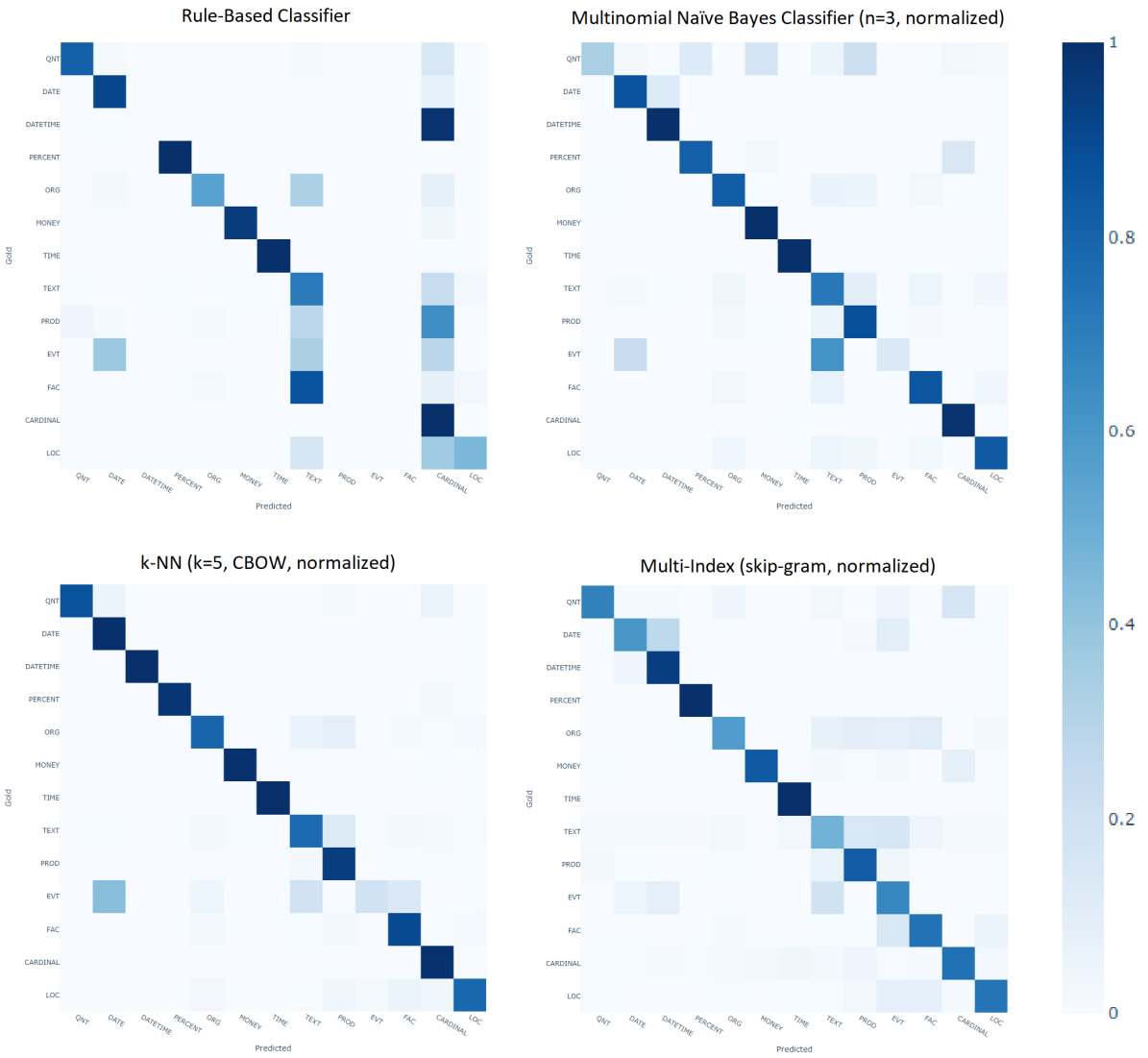


Figure 5.1: Confusion matrices of models with optimal parameters.

5.2.1 Rule-Based Classifier

I implemented the rule-based classifier (RB) early in the experiment to gather samples for most of the concepts in order to have enough training data to use machine learning for further progressing the annotation process (see Section 3.4). As I did not have any parameters to adjust (other than the rules themselves or the static weights – which depended on the rules), the RB has only one measured accuracy.

The RB achieved an accuracy of 52.90%, where the numeric concepts have an accuracy of 73.54 % and string concepts have an accuracy of 28.83%. It was able to barely achieve the highest accuracy for the concept *PERCENT* (98.38%), but both *DATETIME* (0.29%; Figure 5.1 shows that most *DATETIME* samples are classified as *CARDINAL*) and *QNT* (46.09%) seem to have decreased the accuracy of numeric concepts. The RB generally performs bad on string concepts, however, if we exclude the concepts that the RB does not support (i.e., *PROD*, *FAC*, *EVT*) the accuracy of string concepts is 57.66% (macro average of *TEXT*, *ORG*, *LOC*) – which might not seem that bad. Although the RB performs better than just selecting the most common concept (i.e., selecting *TIME* for all samples given as input would yield an accuracy of 35.12%), other machine-learning-based methods does provide a significant increase in accuracy.

5.2.2 Multinomial Naïve Bayes Classifier

In Section 3.5, I explain how I used the multinomial naïve Bayes classifier (MNB) in an attempt to streamline the annotation process. For the MNB that I used in the annotation proccess, I used character n-grams, where $n = 4$, as features and used the raw input (i.e., the input was not normalized). In Table 5.1, we see that the MNB with those parameters achieved an accuracy of 78.08%. However, for the purpose of actual classification, I experimented with increasing and decreasing n by one, and also using normalized samples (see Section 4.2 for details on the normalizing) instead of raw samples.

Again, looking at Table 5.1, we see that the MNB performs best with character 3-grams and normalized samples (marginal difference between raw and normalized samples). However, quite interestingly, we see that when n increases, the accuracy for numeric concepts decreases while the accuracy for string concepts increases.

As the MNB models the data distribution (in combination with a bag-of-words feature representation), it has some obvious issues regarding the less-likely concepts *EVT* (0.08% of the data set) and *QNT* (0.29%). Looking at the confusion matrix in Figure 5.1, it is visible that *QNT* samples are often classified as *PERCENT*, *MONEY* or *PROD*. For *PERCENT* and *MONEY*, the reason is most likely due to the digits in the test samples for *QNT* having a higher probability in the aforementioned concepts, while also being more probable concepts with prior probabilities of 1.01% and 6.06%, respectively. For *PROD*, it is possible that a *QNT* sample completely overlaps as some *PROD* samples may contain measurements for the product, e.g., "HILTON speil, 60x100 cm. Sølv" (*PROD*) contains "100 cm" (*QNT*). As the *PROD* concept has a prior probabilitiy of 10.01% (see Table 3.2), it is more likely that "100 cm" belongs to *PROD* (according to the data distribution). Further, we can see that *EVT* samples are classified as either *DATE* or *TEXT* more often than they are classified correctly. However, as I had some issues annotating samples as *EVT* (see Section 5.1), I assume that the incorrect classification of *EVT* samples are more due to low quality of the annotated samples – caused by human-error, which led to poor data distribution of samples related to *EVT*.

Although the issue highlights the need for reviewing the data set (to ensure a higher quality), I also used k-nearest neighbors for classification, where I used different feature representations of text that have some notion of the semantics (in contrast to bag-of-words which only considers the syntax).

5.2.3 *k*-Nearest Neighbors

I used k-nearest neighbors (k-NN) for classifying concepts based on sentence embeddings. As k-NN is cluster-based, in the sense that it uses data points in vector space to determine some distance between points, my idea was that I would be able to determine which method for obtaining sentence embedding (i.e., either composition or distributed method – see Section 2.4.2) provided the best encoded semantics w.r.t. text related to insurance claims. Considering that EVT only has 80 training samples, I set k relatively low ($k = 5$) in order to have a balance between required number of samples and tolerated number of incorrect samples in the proximity of the input. Table 5.1 shows that k-NN achieves better results than the baseline classifier (MNB).

Based on the accuracy reported in Table 5.1, sentence embeddings obtained by the composition method (specifically, using word embeddings trained with a continuous bag-of-words by FastText) provide more semantic similarities than sentence embeddings obtained using the distributed method (embeddings trained with Sentence-BERT) when used on cell values from spreadsheets. I think the reason for why the composition method performs better for this particular classification task, is that cell values in spreadsheets can be considered as bag-of-keywords, i.e., very few samples are grammatically correct sentences and they are more dependant on having a sound representation of each individual word (as one word can change what concept the sample belongs to).

In addition to choosing k-NN for its capability of grouping similar samples, I chose this algorithm as it provides great support for adding and removing classes. I only used a limited number of concepts, however, the possible concepts present in insurance data is vast – meaning that the possible classes for classification of concepts will increase over time (e.g., upon discovery of a new concept or when replacing a concept for several more fine-grained concepts). For instance, a neural network would require both architectural modification (adding a new neuron/node in the output layer) and re-training the model, while with k-NN, adding a new class is as simple as adding a new annotated sample. However, this simplicity of k-NN comes at the expense of complexity during classification of a given datum.

The k-NN has a classification complexity of $O(n)$, where n is the number of training samples. As I mentioned in Chapter 4, using the cosine similarity function as distance metric, the estimated runtime was over 14 hours. While using the dot product of normalized vectors instead of cosine similarity reduced the runtime significantly, it took a long time to iterate through all 23,998 test samples (roughly 2 hours and 50 minutes to iterate all test samples using FastText embeddings, while Sentence-BERT had a build time of 20 minutes and iterated all test samples in nearly 1 hour and 30 minutes). To test how the k-NN would perform when having $k = 1$ and where each training sample is an aggregation of a concept's training samples, I implemented the multi index (see Section 4.3).

5.2.4 Multi Index

Interestingly, the multi index (MI; described in Section 4.3) achieved an accuracy of 64.78%, using word embeddings trained with skip-gram and normalizing the samples, which is significantly higher than the RB. I implemented the MI in order to reduce the number of samples to compare during classification and the evaluation time of the test set was done in just above 20 seconds (see Figure 5.2). However, the MI computes the cosine similarity and not the dot product between normalized vectors, and given that computing the dot product made the k-NN at least five times faster, the implementation of MI can potentially be improved to complete the evaluation as quick as the MNB.

Although the accuracy of MI is not nearly as high as when using all available training samples with k-NN, it does seem to have made the RB unnecessary to develop. In contrast to the RB, the development time for the MI was low as it is based on mathematical functions and not

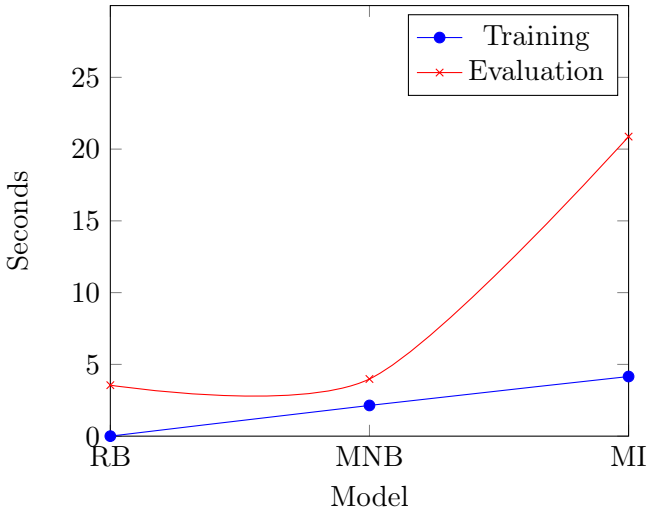


Figure 5.2: Training time and evaluation time for the rule-based classifier (RB), multinomial naïve Bayes classifer (MNB; n=4, raw), and multi index (MI; skip-gram, normalized).

hand-written rules. Based on Occam's razor, the MI is a better solution as the RB has a lot of assumptions (for instance, assumes that text ending with the keyword "AS" is an organisation name) while MI only has the assumption that the samples are correctly annotated. As the MI reached 64.78% using 95,965 training samples, I investigated how it would perform given only one training sample per concept – because if the required number of training samples is high, then the RB may be more efficient to use.

5.2.5 Single-Instance Training

In order to test how well the MI performs with few training samples, I created ten training sets that only contained one randomly selected sample per concept. As the RB is rather static and not trained, I compared the MI trained with single instances with the MNB trained on the same data. Both models were tested on the full test set, where the MI had an average accuracy of 31.70% across the training sets and the MNB achieved 16.03% on average (see Table 5.3).

Further, Table 5.4 shows the accuracy of the models given a training set consisting of 2, 4 and 8 (from Table 5.3) combined. Giving two more samples per concept increases the accuracy of MI to 40.71% and MNB to 23.75% – giving a higher increase to the MI model. However, as 40.71% is lower than the RB, I tried to combine all ten training set, so that each concept had ten samples. As the results in Table 5.5 shows, having ten samples per concept leads to the MI having an accuracy on par with the RB, which are 51.30 % and 52.90%, respectively. Given that the MI only requires ten samples per concept, it would have been easier to use the MI – instead of the RB, early in the annotation process. Another observation from Table 5.5 is that the increase in accuracy is higher for the MNB than for the MI, which indicates that it might not require many samples until the accuracy of the MNB converges with the accuracy of the MI.

I did not do any further experiments on how many samples would be required before the MNB converges with the MI, but the findings of how many samples are required before the MI converges with the RB are quite interesting. It shows that a concept can be represented numerically by the average word embedding of just ten samples and produce results on par with rules that aim at capturing distinct features. In Chapter 6, I provide some ideas for how to further improve the data set, word embeddings and usability of a k-NN for classification.

Training Set	MI(skip-gram, norm)	MNB(n=4, raw)	Difference (MI - MNB)
1	37.20%	19.79%	17.42%
2	23.19%	18.92%	**5.28%**
3	34.80%	15.71%	19.09%
4	33.23%	10.93%	**22.30%**
5	30.55%	13.13%	17.41%
6	36.00%	17.74%	18.25%
7	35.50%	15.44%	20.06%
8	30.55%	14.63%	**15.92%**
9	25.47%	16.85%	8.61%
10	29.77%	17.42%	12.35%
Macro Avg.	31.70%	16.03%	15.67%

Table 5.3: F_1-score of multi index (MI) and multinomial naïve Bayes (MNB) trained with ten different training sets containing one randomly selected sample per concept; tested on full test set; training sets with lowest, highest and average difference in **bold**.

	MI	MNB	Difference (MI - MNB)
Avg. (2, 4, 8)	29.23%	14.73%	14.50%
Combined (2, 4, 8)	40.71%	23.75%	16.96%
Increase (comb. - avg.)	11.49%	9.02%	2.46%

Table 5.4: F_1-score of multi index (MI) and multinomial naïve Bayes (MNB) trained on combined set consisting of training set 2, 4 and 8 from Table 5.3.

	MI	MNB	Difference (MI - MNB)
Avg. (1-10)	31.70%	16.03%	15.67%
Combined (1-10)	51.30%	38.03%	13.27%
Increase (comb. - avg.)	19.60%	22.00%	-2.40%

Table 5.5: F_1-score of multi index (MI) and multinomial naïve Bayes (MNB) trained on combined set consisting of all sample sets from Table 5.3.

	B	C	D	E	F	G	H	I
2	**GLASSBYGG AS**							Dato: 30.01.2012
3	**Stormskade**							
4								
5	**Post**	**Beskrivelse**		**Pris**	**Antall**	**Sum**	**Vedlegg**	**Kommentar**
6	1	Glass fra Pilkington				32 289	1	
7	2	Glass fra Pilkington				34 453	2	
8	3	Glass fra Pilkington				2 143	3	
9	4	Glass fra Pilkington				6 105	4	
10	5	Arbeid, faktura fra P. J. Hinsværk				1 700	5	
11	6	Arbeidstimer egne ansatte		246	13.5	3 321	6	Lønnskostnader vedl. 13
12	7	Arbeidstimer egne ansatte		280	13	3 640	7	Lønnskostnader vedl. 13
13	8	Arbeidstimer egne ansatte		246	5.5	1 353	8	Lønnskostnader vedl. 13
14	9	Arbeidstimer egne ansatte		221	5	1 105	9	Lønnskostnader vedl. 13
15	10	Arbeidstimer egne ansatte		329	12.5	4 113	10	Lønnskostnader vedl. 13
16	11	Servicebil	km	3.60	100	360		Ikke dokumentert
17	12	Egen kranbil	timer	500	13	6 500		Ikke dokumentert
18	13	Fasadeglass Grilstad	m^2	325	15	4 875	11	Ikke bestilt pr. dato, samlebestilling senere, m^2-pris dokumentert
19	14	Dører eget lager	stk	10 131	3	30 393	12	Egenproduksjon, dokumentert med kalkyle
20	**Sum eks. mva.**					**132 350**		
21								
22	GLASSBYGG as							
23	Ingv. Ystgaards v 9							
24	7047 Trondheim							

Figure 5.3: Damage overview in a Norwegian property claim (person names have been removed). Source: Protector Forsikring.

	B	C	D	E	F	G	H	I
2	**ORG**							DATE
3	**TEXT**							
4								
5	**TEXT**	**TEXT**		**TEXT**	**TEXT**	**TEXT**	**TEXT**	**TEXT**
6	CARDINAL	TEXT				CARDINAL	CARDINAL	
7	QNT	TEXT				CARDINAL	QNT	
8	QNT	TEXT				CARDINAL	QNT	
9	CARDINAL	TEXT				CARDINAL	CARDINAL	
10	CARDINAL	TEXT				MONEY	CARDINAL	
11	CARDINAL	TEXT		CARDINAL	CARDINAL	CARDINAL	CARDINAL	DATE
12	CARDINAL	TEXT		MONEY	CARDINAL	CARDINAL	CARDINAL	DATE
13	CARDINAL	TEXT		CARDINAL	CARDINAL	CARDINAL	CARDINAL	DATE
14	CARDINAL	TEXT		CARDINAL	CARDINAL	CARDINAL	CARDINAL	DATE
15	CARDINAL	TEXT		CARDINAL	CARDINAL	CARDINAL	CARDINAL	DATE
16	CARDINAL	TEXT	TEXT	CARDINAL	PERCENT	CARDINAL		TEXT
17	CARDINAL	TEXT	TEXT	MONEY	CARDINAL	MONEY		TEXT
18	CARDINAL	TEXT	QNT	MONEY	CARDINAL	CARDINAL	CARDINAL	TEXT
19	CARDINAL	TEXT	QNT	CARDINAL	QNT	CARDINAL	CARDINAL	TEXT
20	**TEXT**					**CARDINAL**		
21								
22	ORG							
23	PROD							
24	LOC							

Figure 5.4: Annotated concepts of spreadsheet shown in Figure 5.3 (using k-NN classifier).

5.3 Result of Spreadsheet Enrichment Pipeline

Considering that my motivation was to find the concepts of cell values in spreadsheets to make data in spreadsheets more accessible w.r.t. information extraction, I ran the spreadsheet shown in Figure 5.3 through the pipeline illustrated in Figure 3.1. Using the k-NN to classify concepts, the cells in the spreadsheet in Figure 5.3 are annotated with the concepts in Figure 5.4. The data structure in Listing 5.1 is the output from the pipeline, where the structure is created during the spreadsheet processing and the *concept* attribute in each cell is set by the classifier.


```
{
  "sheets": [
    {
      "name": "Ark1",
      "cells": [
        {
          "address": "B2",
          "type": "STRING",
          "format": "General",
          "formula": "",
          "value": "GLASSBYGG AS",
          "formattedValue": "GLASSBYGG AS",
          "concept": "ORG"
        },
        ...
        {
          "address": "B24",
          "type": "STRING",
          "format": "General",
          "formula": "",
          "value": "7047 Trondheim",
          "formattedValue": "7047 Trondheim",
          "concept": "LOC"
        }
      ]
    }
  ]
}
```


Listing 5.1: Spreadsheet as a JSON-object after processing.

As we can see in Figures 5.3-5.4, the costs (column G) do not have any special formatting and are mostly classified as cardinals. By looking at the spreadsheet, we see that column C is the sum of each cost related to the damage, as the header is "Sum". For instance, the price in cell E11 multiplied by the amount in the cell F11 equals to the sum in the cell G11 ($246 \times 13.5 = 3121$). Because there is no money format (i.e., currency symbol or currency code) in the column G (the value 3121 is displayed as "3 121"), there are no features besides the digits themselves that can capture the similarity to samples annotated as money. As many numeric concepts can be found in insurance-claim-related spreadsheets, the format of numeric cell values are important to distinguish the numeric concepts. To illustrate the importance of the formats, we see that the cell D18 (symbol for square meters) and the cell D19 (symbol for pieces) are classified as quantities – even though they do not contain any digits. Although two cells in column G (G10 and G17) are classified as money, they are wrongly classified, as there is nothing in the cells that

indicates that they represent monetary values. The two cells are classified as money because the majority of the five ($k = 5$) nearest samples are annotated as money. Assuming that the column G contains the costs because of the two values classified as money would not work in practice, as then we could assume that the column B contains quantities (see the cell B7 and the cell B8). Therefore, I would not rely on this pipeline for extracting insurance claim costs and use them in an automated decision, as it is too prone to errors that can cause wrong claim handling (e.g., incorrect costs would cause an incorrect estimate for claim payout). However, meta-information, such as organisation names and locations, are more accessible by having an annotation which describes what concept a cell value belongs to.

For instance, if we want to link the spreadsheet in Figure 5.3 to profiles of organisations that are present in the file – without manual labour, then a standard approach would be to lookup each cell value in an official organisation registry, e.g., via an API. For this particular spreadsheet, that would lead to 100 requests against an API, and if the API is expensive to use, this approach would not be cost efficient. In the spreadsheet, only two cells are annotated as organisations ("GLASSBYGG AS" and "GLASSBYGG as"). Using the data structure in Listing 5.1, we can form a query, such as, *give me all the values that represent an ORG*, and we only get two values to lookup externally. Searching for "GLASSBYGG AS" in Norway's official organisation registry, yields a match and the public identification for the company GLASSBYGG AS can be used as a key to link GLASSBYGG AS to the spreadsheet.[1] Although linking an organisation that is already linked in a structured system (e.g., the insured) does not provide additional value, this method would enable to link other parties that are not entered into the system (e.g., organisations that have been hired to provide a service related to the claim).

Furthermore, an issue that I encountered was that personal data had to be excluded from the data set, but I did not know which files (or values in the files) contained personal data. By extending the set of concepts to include person names, phone numbers, etc., the pipeline can be used as a method to identify personal data, making it possible to increase the control of sensitive files. Although the classifiers I used does classify some values wrong, discovering some personal data would be better than having no knowledge of which files may contain personal data. In addition, long processing time would not be an immediate issue, as the process of identifying personal data can be seen as a background process that does not affect the claim handling. As the k-NN provides the highest accuracy, I would use this classifier in the pipeline if the objective is to identify personal data. Even though the runtime of the pipeline would be relatively long with the k-NN (compared to the MNB), it would be able to identify the most values that are at risk of being personal data. In addition, using the k-NN and sentence embeddings as features, we would have the opportunity to only store the annotated samples as vectors with numeric values – therefore, not increasing the number of locations where personal data are stored. For instance, instead of storing the mapping "Vemund Justnes" $\mapsto$ PERSON, we could convert the sample "Vemund Justnes" to a vector v (using an embedding function), and then store the mapping $v \mapsto$ PERSON (providing anonymization to some extent). However, storing the vector only would require the annotation to be highly certain, because reverting an embedded vector to the original sample is more challenging than reverting a bag-of-words feature vector to the original sample (i.e., a bag-of-words representation stores a vocabulary of words present in the training samples). Due to the challenge of reverting an embedding, the embedding function can not be changed after some samples have been converted, and I would suggest to use the CBOW trained by FastText, as those embeddings in combination with the k-NN achieved the best accuracy.

[1] Official registration for Norwegian organisations:
https://w2.brreg.no/enhet/sok/treffliste.jsp?navn=GLASSBYGG+AS&orgform=0&fylke=0&kommune=0

Chapter 6

Future Work

6.1 Fine-Grained Concepts

As I mentioned several times in Chapter 5, the data set should be reviewed by another individual in order to improve the quality of the annotations. To exploit a potential review process, I suggest that some new concept labels are introduced to the data set. For instance, in Section 5.1, I observed that there should have been labels for the concepts *invoice* and *action* to separate these concepts more clearly from concepts, such as organisation names or plain text. Furthermore, I would suggest to split some of the concepts that I used in to more fine-grained concepts, with an emphasis on products and locations. In the current data set, I annotated text describing, e.g., books, food, electronics, clothing, jewelry, vehicles, etc., as products. Considering that vehicles are expensive and can be supplemented with external information that may affect the premium (e.g., engine power), samples such as "AUDI Q7" and "Hitachi ZX190WT, hjulgravemaskin" should have been annotated as vehicles instead of products. Locations such as "Støperigata 2, Pb 1351 0113 OSLO" should have been annotated as address in order to separate them from more unspecific locations such as "OSLO - RYEN". Having a slightly more detailed list of concepts would enable us to extract data from insurance-related spreadsheets, either related to claims or underwriting, and possibly enriching the data before entering it into a structured system.

6.2 Fine-Tuned Word Embeddings for the Insurance Domain

In Section 5.2, I observed that using the composition method, where sentence embeddings were composed of word embeddings, provided the best result in classification of the concept. Therefore, I think it would be interesting to fine-tune the word embeddings specifically for the insurance domain. Considering that the word embeddings are trained using unsupervised learning algorithms, and that the data are unstructured texts, fine-tuning for the insurance domain would not require a lot of development time. As the insurance industry has an abundance of unstructured text, e.g., settlement letters, correspondence, claim reports, insurance policies, etc., we have access to a large amount of insurance-domain-specific text. For the purpose of fine-tuning word embeddings, the data set that I have created can be used to measure whether word embeddings trained on insurance data capture the semantics better than word embeddings trained on data from, e.g., Wikipedia or Common Crawl.

6.3 *k*-Means for Multi Index

The indices in my multi index (MI) implementation are essentially created by a k-means (i.e., each concept has a data set containing embeddings which are then clustered by finding the mean embedding) and the evaluation is done by using k-nearest neighbors, where $k = 1$ for both indexing and evaluation. As a future improvement, support for setting k in the indexing, i.e.,

setting the number of indices to include per concept, could be implemented. Then, we would be able to set k, e.g., by:

$$k = \frac{\text{max number of samples}}{|C|} \tag{6.1}$$

where max number of samples is set according to runtime during evaluation and C is the set of concepts. To consider the fact that some concepts are more sparse than others in vector space, the implementation should have some functionality to decrease k in more dense concepts and increase k of sparse concepts.

6.4 Approximate Nearest Neighbors

Although k-nearest neighbors is useful when the number of classes can change frequently, I encountered the issue that it is highly inefficient during evaluation as it compares the input to each training sample. As an alternative, there is some research on approximate nearest neighbors which, according to my knowledge, generally works by creating a hierarchy for the samples. It seems that the state-of-the-art for approximate nearest neighbors is to represent the hierarchy as a graph (hierarchical navigable small world graphs [19]) and I think that this method should be compared to both brute-force k-nearest neighbors and the multi index for the data set that I created.

6.5 Learning a Similarity Function

As an alternative to k-nearest neighbors or approximate nearest neighbors, a suggestion is to investigate learning a similarity function using the triplet loss function (see Definition 2.4.1). While a neural network for classification often requires that the size of the output layer is equal to the number of classes, a neural network with the triplet loss function could have an output layer of size one, where the value is the similarity score. An approach could be to use the training samples as anchor points and indices (i.e., such as the indices that I used in the multi index) as positive points and negative points, thereby, learning the similarity between input data and a vector representation of each class. The classification process would then consist of computing the similarity to each class and select the highest scoring class as output. In contrast to using a similarity metric such as cosine similarity, the neural network would be able adjust weights of each feature value and possibly identify distinctions on a feature level (i.e., the similarity is decided based on the feature values instead of a whole feature vector). Given that such an approach is practical, my hypothesis is that adding a new class would not require any architectural modifications, but only re-training of the weights in the neural network.

Chapter 7

Conclusion

In this thesis, I explain how I created a new data set annotated with concepts, that can be used for supervised classification. I provide some results of using the data set for classification, where I used a multinomial naïve Bayes classifier (MNB) as a baseline and a k-nearest neighbors (k-NN) in order to compare different feature representation of text gathered from insurance-claim-related spreadsheets. The MNB used a simple bag-of-character n-gram representation, which achieved an accuracy of 79.11% with $n = 3$ on normalized text. The k-NN used sentence embeddings to represent the text, where I tried both the composition method and distributed method for obtaining sentence embeddings, and achieved an accuracy of 87.09% when using the composition method on normalized text. However, while the MNB evaluated 23,998 samples in less than five seconds, the k-NN needed roughly 2 hours and 50 minutes.

Therefore, I provide an implementation of a method where I cluster the samples of a concept in order to have a single vector per concept to compare to. The implementation used the cosine similarity, and not the dot product between normalized vectors, and evaluated the test set in around 20 seconds. As I found that using the dot product was at least five times faster than using the cosine similarity in the k-NN, I think that the evaluation time can be reduced to the same level as the MNB with a minor modification to the implementation. However, this simplified indexing method did cause a significant drop in accuracy (64.78% using word embeddings trained with skip-gram and normalized text) and made the MNB seem like a better classification method. Although the accuracy was lower than the MNB, it was significantly higher than the accuracy of the rule-based classifier (RB) that I present in Section 3.4. I used the RB to initiate the annotation process and it achieved an accuracy of 52.90% on the test set. An observation that I did was that the indexing method performed better than the RB, which lead me to investigate how many samples were needed in order to create indices so that the accuracy was on par with the RB – as finding examples is a lot less time consuming than writing rules. I found that with just ten samples per concept, the accuracy of the indexing (51.30%) was on par with the RB.

To conclude, considering the motivation of knowing the concept of cell values in spreadsheets to make information extraction from spreadsheets more practical, I find that the method I describe is not sufficient enough for extracting data that affect the claim handling, as there is too much error in the classification of concepts. As I illustrated in Figures 5.3-5.4, the classification relies on having a format for values of a specific concept. For instance, monetary values require either a currency symbol or currency code in the formatted value, but not all monetary values have a format. Therefore, the method is not suitable to extract information such as costs related to a claim – which was the information that lead me to investigate this approach for extracting information from spreadsheets, so that costs could automatically be mapped to an insurance coverage. However, I did find the method useful to extract meta-information, such as organisation names or locations, where the information can be further enriched using external data providers.

Most notable is that the method can be used to identify personal data in spreadsheets,

thereby limiting the issue that I encountered, where I did not know which data values needed to be anonymized due to privacy regulations. The accuracy that I achieved is not an issue in the task of identifying personal data, as knowing which files *might* contain such data is better than not knowing at all. Considering that the process of identifying personal data does not affect the claim handling, it can be executed as a background process, where the evaluation time of the classifier is not a big issue. Therefore, I would suggest using the k-NN with average word embeddings as feature representation, as this classifier achieved the best accuracy. In order to identify personal data, samples for concepts such as person names, phone numbers, etc., have to be added to the data set. As I found that the indexing method I implemented achieved a higher accuracy than the RB with only ten samples per concept, I would suggest to use the indexing method instead of writing rules to obtain samples for concepts regarding personal data.

Although I have not developed any new state-of-the-art methods, I have identified future work which can take advantage of, and improve, the data set that I have created. I underestimated how ambiguous data can be: even when the data only contains a limited amount of information, it can be difficult to determine what the information actually concerns. Therefore, I suggest that the data set should be reviewed, but while doing so, new and improved concepts can be introduced in order to make the data even more useful. Furthermore, as the composition method was proven to yield the best results when classifying the concepts, I think that some effort should be put into fine-tuning the word embeddings that I have used, by training them on unstructured insurance data. I believe this would encode the semantics of words more suitably to the insurance domain – causing better grouping of similar concepts, rather than relying on semantics that is found in text from Wikipedia or Common Crawl. The created data set can then be used to measure whether insurance-specific word embeddings provide a better grouping of similar concepts. Finally, as the k-NN has a significantly higher accuracy than the MNB, some effort should be put into trying the state-of-the-art algorithm for approximate nearest neighbors.

Bibliography

[1] P. Bojanowski, E. Grave, A. Joulin and T. Mikolov, 'Enriching word vectors with subword information', *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017, ISSN: 2307-387X.

[2] E. Grave, P. Bojanowski, P. Gupta, A. Joulin and T. Mikolov, 'Learning word vectors for 157 languages', in *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[3] N. Reimers and I. Gurevych, 'Sentence-bert: Sentence embeddings using siamese bert-networks', *CoRR*, vol. abs/1908.10084, 2019. arXiv: 1908.10084. [Online]. Available: http://arxiv.org/abs/1908.10084.

[4] F. Rosenblatt, 'The perceptron – a perceiving and recognizing automaton', 1957. [Online]. Available: https://blogs.umass.edu/brain-wars/files/2016/03/rosenblatt-1957.pdf.

[5] F. Jørgensen, T. Aasmoe, A.-S. R. Husevåg, L. Øvrelid and E. Velldal, 'Norne: Annotating named entities for norwegian', in *Proceedings of the 12th Edition of the Language Resources and Evaluation Conference*, 2020. [Online]. Available: https://arxiv.org/abs/1911.12146.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever and D. Amodei, *Language models are few-shot learners*, 2020. arXiv: 2005.14165 [cs.CL].

[7] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: 1810.04805 [cs.CL].

[8] Z. Harris, 'Distributional structure', *Word*, vol. 10, no. 2-3, pp. 146–162, 1954. DOI: 10.1007/978-94-009-8467-7_1. [Online]. Available: https://link.springer.com/chapter/10.1007/978-94-009-8467-7_1.

[9] M. F. Porter, 'An algorithm for suffix stripping', *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[10] J. R. Firth, 'A synopsis of linguistic theory 1930-55.', vol. 1952-59, pp. 1–32, 1957.

[11] T. Mikolov, K. Chen, G. Corrado and J. Dean, *Efficient estimation of word representations in vector space*, 2013. arXiv: 1301.3781 [cs.CL].

[12] J. Pennington, R. Socher and C. D. Manning, 'Glove: Global vectors for word representation', in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162.

[13] T. K. Landauer, P. W. Foltz and D. Laham, 'An introduction to latent semantic analysis', *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[14] J. Devlin, M. Chang, K. Lee and K. Toutanova, 'BERT: pre-training of deep bidirectional transformers for language understanding', *CoRR*, vol. abs/1810.04805, 2018. arXiv: 1810.04805. [Online]. Available: http://arxiv.org/abs/1810.04805.

[15] G. Koch, R. Zemel and R. Salakhutdinov, 'Siamese neural networks for one-shot image recognition', 2015.

[16] V. I. Levenshtein, 'Binary codes capable of correcting deletions, insertions and reversals.', *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966, Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[17] F. Damerau, 'A technique for computer detection and correction of spelling errors.', *Commun. ACM*, vol. 7, no. 3, pp. 171–176, 1964. [Online]. Available: http://dblp.uni-trier.de/db/journals/cacm/cacm7.html#Damerau64.

[18] A. Coates and A. Y. Ng, 'Learning feature representations with k-means.', in *Neural Networks: Tricks of the Trade (2nd ed.)* Ser. Lecture Notes in Computer Science, G. Montavon, G. B. Orr and K.-R. Müller, Eds., vol. 7700, Springer, 2012, pp. 561–580, ISBN: 978-3-642-35288-1. [Online]. Available: http://dblp.uni-trier.de/db/series/lncs/lncs7700.html#CoatesN12.

[19] Y. A. Malkov and D. A. Yashunin, 'Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs', *CoRR*, vol. abs/1603.09320, 2016. arXiv: 1603.09320. [Online]. Available: http://arxiv.org/abs/1603.09320.