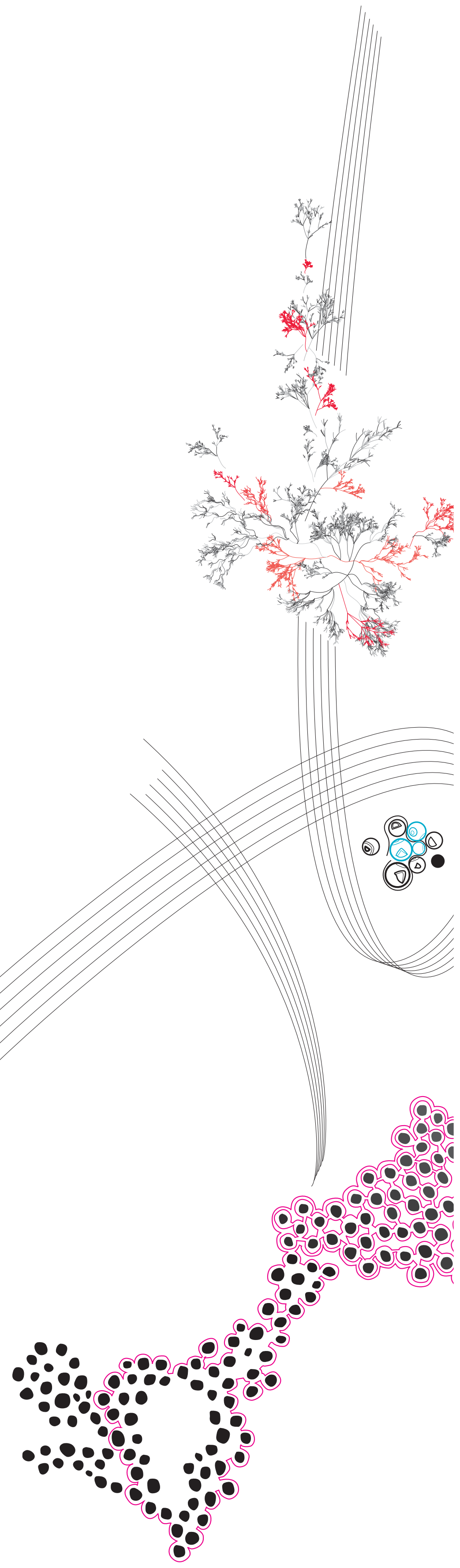
MSc Computer Science
Final Project



Ledger Category Classification in Bookkeeping: Applying Machine Learning on Dutch Invoice Data

Pepijn Mesie

UT Supervisors: dr. Faiza Bukhsh, dr. Shenghui Wang

Company Supervisors: Remco de Man, Jeroen Smienk

September, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Bookkeeping can be a complex and time-consuming process for small companies due to the many ledger codes available. Because of this, Moneybird has created a research assignment to investigate the possibility of implementing machine learning algorithms to speed up their customers' bookkeeping process. In this thesis, we investigate the possibility of implementing a machine learning model that predicts the correct ledger code for Dutch invoice data using textual descriptions of the invoice lines and contextual data such as the contact company name. First, we developed classifiers that were trained to classify invoice lines based only on the textual description. We implemented shallow classifiers trained on TF-IDF embeddings, a deep neural network trained on FastText embeddings, and a BERT-based model. The results of this experiment showed that the BERT-based model achieved the highest Macro F1 score (±60%). Continuing from these results, we investigated whether the performance could be improved by including contextual information such as the company name and the type of legal entity in the classifiers created in the first experiment. The results of this experiment showed that contextual information improves classification performance by up to ±14%. Finally, the classification speeds of the previously developed models are compared to determine which model is best suited for use in a practical setting. This experiment shows that the deep neural network that includes both FastText embedded textual descriptions, as well as contextual data in its input achieved the best results, achieving the highest macro F1 score of ±71% and a prediction speed of 0.0014s for 1000 samples. Using the best-performing model developed using the research questions, we implemented a prototype to show how machine learning models could be implemented to suggest ledger codes to customers of the bookkeeping software. With the results and prototype, we can conclude that it is possible to create a machine learning classifier that can predict ledger codes based on Dutch invoice data. Finally, we discuss the limitations of this research, such as the limited number of classes included in the data set and future research recommendations.

Keywords: Machine learning, Text classification, Bookkeeping, Invoice classification, Ledger account classification

Chapter 1

Introduction

Bookkeeping is the process of recording transactions. In the Netherlands, every company is required by law to perform bookkeeping. These transactions are often stored as invoices that contain one or more lines that describe the product or service the company provides or receives. A key part of the bookkeeping process is maintaining ledgers. A ledger is a collection of the income and expenses a company has made. Ledgers can be created by assigning ledger codes to these invoice lines. A ledger code is a number or a character combination that is used to group different incomes and expenses based on their use. The selection of the correct ledger code can be difficult as there are many codes available. For example, in the Dutch Reference Classification System of Financial Information (RCSFI)[1] ledger code system, there are different codes for *Office equipment* and *Small purchases of office inventory*. The subtle differences between ledger codes can make the bookkeeping process time consuming and error-prone, especially for smaller companies that do not have a dedicated bookkeeper. Furthermore, each line of every invoice needs to be classified individually, which results in companies spending a lot of time on their bookkeeping.

Since manually keeping track of all invoices can be difficult, some companies choose to use bookkeeping software. These programmes can provide solutions to easily send outgoing and process incoming invoices to their system, allowing the company to spend more time on its core business rather than on administrative work. These solutions can keep track and store invoices in an easily interpretable way and also allow for automation such as automatically sending reminders for unpaid invoices to simplify the bookkeeping process.

1.1 Problem definition

This research assignment was provided by Moneybird. Moneybird is a Dutch company that develops online bookkeeping software focused on simplifying the invoicing and bookkeeping process of small to medium-sized companies. Their goal is to ensure that entrepreneurs can focus on their business instead of their bookkeeping. Since the selection of ledger codes is still a complicated and time-consuming process, Moneybird wanted to explore the possibilities of making the selection process faster and more straightforward for its users. For their research assignment, Moneybird wants to explore an implementation of a machine learning system to predict ledger codes based on textual and contextual information from

[1] https://www.referentiegrootboekschema.nl/english

invoice lines.
Since smaller companies might not have a very large number of invoices that can be trained on, it is important to create a generalisable model that can classify ledger codes without being overfitted to a single company. This presents a significant challenge, as a car dealer's invoice for a car intended for sale might have the same description as another company's invoice for the same car used for transportation, yet the ledger codes would differ. Another aspect is that the textual data for the invoice lines are very short and contain little to no context, which can make it difficult for a model to distinguish between ledger accounts. Contextual data from the invoice could provide more information and improve the performance of the classifiers, something that has not been researched yet for Dutch invoice data. Finally, since the invoice creation in Moneybird is performed through a web form, the classification process mustn't decrease the responsiveness of the form. This requirement emphasises the importance of comparing not only classifiers based on their classification performance but also prediction speed.

1.2 Research questions

To create a system that can predict ledger codes based on textual and contextual data from Dutch invoices, a set of research questions has been developed. The main research question is *How can classification be applied to predict the correct ledger code for Dutch invoice lines?* This research question can be answered through answering the following subquestions:

SQ1: How do feature extraction methods for textual data such as word embeddings or TF-IDF affect the performance of ledger category classification models?

SQ2: What is the impact of the inclusion of contextual data, such as company information, on the performance of classification models designed to classify ledger codes for invoice data?

SQ3: How scaleable are the classification models developed and what considerations should be made before they are deployed?

The goal of the first research question is to research, implement and test different approaches to transform textual descriptions into an encoding that is better suited for machine learning. For the first research question, we will also develop deep learning network designs that can be expanded with contextual data in the second experiment.

The second research question will be answered by extending the approaches of the first research question by examining contextual data and their correlations to the ledger codes. The contextual data that showed correlation are then included in the classification models designed in the first research question, after which the performance of the classifiers is evaluated.

The goal of the third sub-question is to evaluate whether the classifiers developed in the first two research questions have practical value. An important consideration for this is the prediction speed, as a model that takes multiple seconds to generate a prediction will not be able to provide a good user experience when implemented into a web form. Therefore, it is essential to develop a model that can make fast predictions even when many users are creating invoices at the same time. Furthermore, it is important to consider factors like the requirement of a GPU for the implementation of a model, as these increase the costs of deployment significantly. The relation between the research questions and their results is described in Figure 1.1.

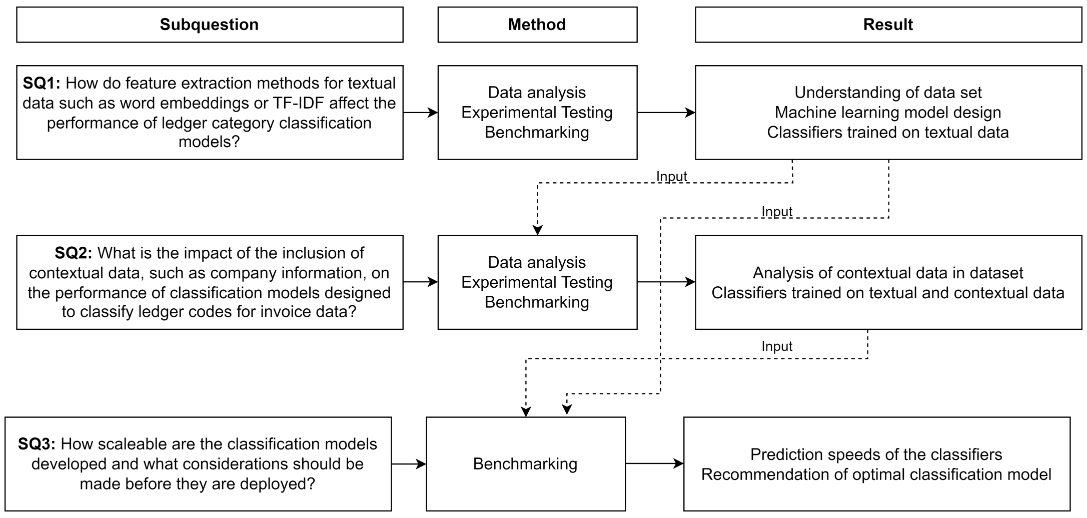

Subquestion
Method
Result
SQ1: How do feature extraction methods for textual data such as word embeddings or TF-IDF affect the performance of ledger category classification models?
Data analysis
Experimental Testing
Benchmarking
Understanding of data set
Machine learning model design
Classifiers trained on textual data
Input
Input
SQ2: What is the impact of the inclusion of contextual data, such as company information, on the performance of classification models designed to classify ledger codes for invoice data?
Data analysis
Experimental Testing
Benchmarking
Analysis of contextual data in dataset
Classifiers trained on textual and contextual data
Input
SQ3: How scaleable are the classification models developed and what considerations should be made before they are deployed?
Benchmarking
Prediction speeds of the classifiers
Recommendation of optimal classification model

Figure 1.1: A graphical representation of the subquestions and their relationships to each other

With the results of the research questions, the best-performing model is selected to use in the development of a prototype to test the practical value of the results. The prototype will be integrated into the existing bookkeeping software and evaluated by some of its developers.

1.3 Document structure

The remainder of the thesis is structured into 5 chapters. In Chapter 2, background information on bookkeeping and machine learning is defined, as well as related works. Chapter 3 describes the data set, investigating features and data quality. This chapter also describes data preparation. Chapter 4 describes the results of the experiment, which are used in the development of the prototype, described in Chapter 5. Finally, chapter 6 summarises the work, describes limitations, and recommends future work.

Chapter 2

Background

The purpose of this chapter is to provide a general overview of the methods used to carry out the research, as well as some background knowledge of the RCSFI standard for bookkeeping in the Netherlands. Finally, the section will describe related works that investigated the classification of ledger codes for invoice data, investigating their similarities and differences to this research.

2.1 CRISP-DM

In our research, we use the CRoss-Industry Standard Process for Data Mining (CRISP-DM) guidelines described by Wirth et al.[23] to carry out the experiments. Their paper describes a cycle of six processes that can be applied to perform a data mining project:

1. **Business understanding:** This phase examines the objectives and requirements that must be met for the project to be successful.
2. **Data understanding:** Once data has been collected, the data understanding phase starts, where the goal is to identify potential data quality issues and discover insights into the data set.
3. **Data preparation:** The goal of the data preparation phase is to develop the final data set that will be used for modelling from the collected data set. This phase includes feature selection, as well as data preprocessing.
4. **Modelling:** In the modelling phase, the prepared data is applied to (several) models, and the results are collected.
5. **Evaluation:** After results have been generated, the results need to be evaluated to determine whether the results achieve the requirements stipulated in the first phase of the experiment.
6. **Deployment:** Once the evaluation has determined the approach to match all requirements and objectives, the model can be deployed to the customer.

Since new insights are often made during the different phases of a project, the CRISP-DM guidelines describe these phases in a cycle where (several) phases can be repeated to update and improve previous steps (described in Figure 2.1).

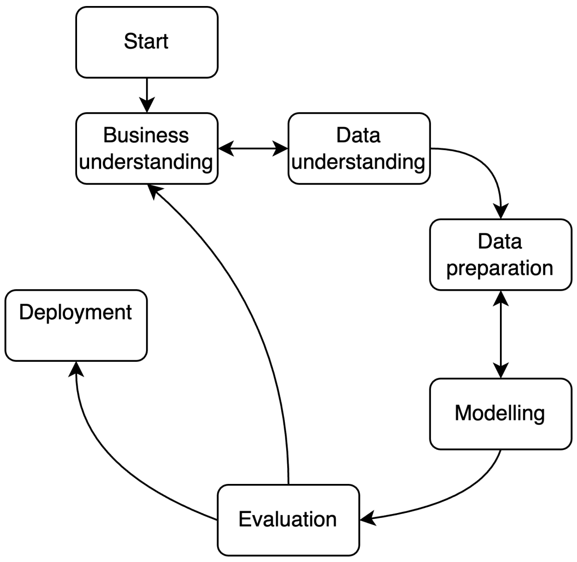


FIGURE 2.1: The CRISP-DM project cycle, as described in Wirth et al.[23]

2.2 Bookkeeping

The process in which this research is conducted is focused on the selection of ledger codes. Whenever a company sends or receives invoices, the company has to perform bookkeeping to ensure that all incomes and expenditures are taxed correctly. Bookkeeping is generally performed through a system of ledger accounts, where each transaction is recorded, stored, and grouped by type. These ledger accounts can be combined to generate ledgers, which can be used to visualise the financial situation of a company over time. These ledgers are also used when companies file for their taxes.

2.2.1 RCSFI

A ledger is a summary of all revenue and expenses a company makes, including its debts and assets. To facilitate the generation of ledgers, online bookkeeping software can make use of a general ledger scheme. These schemes contain standardised codes that are used to group expenses and income streams into different categories according to the types of revenue used for tax returns. The standard used in The Netherlands is the RCSFI which was created by the Dutch government and uses a tree structure to categorise the ledger codes, an example of which can be found in Table 2.1.

Ref-code	**Description**
WOmz	Net turnover
WOmzNoh	Net turnover from the sale of trade goods
WOmzNohOlh	Net turnover from the sale of trade goods taxed at a high rate
WOmzNohOlv	Net turnover from the sale of trade goods taxed at a low rate

TABLE 2.1: An example of the tree structure of the RCSFI[1]

2.3 Machine learning

The goal of machine learning is to use computers to create predictions on unseen data using a model based on previously seen data. Using a large set of data that represents the data that needs to be predicted, a large number of variables are tuned to calculate the correct output given the input.

Machine learning can be divided into two types, namely supervised and unsupervised approaches. For supervised approaches, each training sample contains a target label, and the goal of a machine learning model is to predict the label based on the sample. For unsupervised machine learning, no such labels exist. The goal of the model is to be able to distinguish between samples and/or group samples together.

2.3.1 Loss function

To be able to train machine learning algorithms, a loss function is used to determine the difference between the correct output (y) and the predicted class ($\hat{y}$). This difference can be used to determine the direction in which the model needs to be updated so that there is a smaller difference between the expected and predicted outputs, resulting in a model that can correctly classify samples. This section describes some common classification losses.

The 0/1-loss is a very simple loss function that is either 0 when a sample is correctly classified, and 1 if the sample is misclassified.

$$L_{0/1} = \begin{cases} 0 & \text{if } \hat{y} = y, \\ 1 & \text{otherwise,} \end{cases}$$

Where $\hat{y}$ is the predicted class and y is the actual class

The 0/1-loss function minimises misclassification, but a downside is that it is not differentiable. Differentiable loss functions are preferred because the slope (which is calculated using the derivative) can be applied to determine how extreme the model updates need to be. A model with 0/1 loss cannot be trained very effectively, as all weight updates would be equally large. Alternatives that are be differentiable are the *Squared hinge loss*[11] and *binary cross-entropy loss*[3, 11]:

$$L_{Hinge^2} = max(1 - y \cdot p, 0)^2$$
$$L_{BCE} = -(y \cdot log(p) + (1 - y) \cdot log(1 - p))$$

Where y is the expected class and p the probability of class 1

2.3.2 Multi-class classification

Supervised classification algorithms can again be divided into two groups. Binary classification is a classification problem where the model divides the samples into two classes, for example, $True$ or $False$. In multiclass classification problems, the model has to choose from more classes when predicting, such as *Dog*, *Cat*, or *Snake*.

Because all the loss functions described above only work with the class being either $True$ or $False$, these cannot be used for multiclass classifiers. The *binary cross-entropy loss* can be extended to work with more than two classes by summing the loss of all classes[3]. Given a sample labled x, the loss function sums log_2 of $(1 - p_i)$ (if the label i is not correct) or (p_i) (if the label i is correct). if the probability of a class is close to 1, $log_2(p_i)$ is low and $log_2(1-p_i)$ becomes a larger negative value. By applying the activation function for every label and negating the sum of the results, a number can be calculated that is high if incorrect classes have a high probability and low if only the correct class has a high probability. The *cross-entropy loss* uses the following function:

$$L_{CE} = -\sum_{i}^{C} (y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i))$$

Where C is the number of classes, y_i is the correct output for class i, and p_i is the probability of class i

2.3.3 Hierarchical classifiers

For data sets where the labels contain a hierarchical structure, it can be beneficial to reflect the hierarchical structure in a classification model. Hierarchical classifiers divide the larger classification problem into smaller classification problems in each of the layers of the hierarchical structure. Miranda et al.[16] created the Hiclass library, implementing three approaches to hierarchical learning (see Figure 2.2).

The first approach is to train a classifier for each possible label of every level of the hierarchy (see Figure 2.2.D). This model has the largest number of classifiers, each of which is trained in a *one vs rest* approach. To classify using the *local classifier per node* approach, the top-level labels are queried, after which the label with the highest probability is selected. Afterwards, only the classifiers that are children of the label selected in the previous hierarchy are queried. The process is repeated until the final hierarchical layer has been queried and the prediction is complete.

The second approach creates a classifier for all nodes in the hierarchical tree that have child nodes (see Figure 2.2.C). This approach trains the individual classifiers to distinguish the labels with the same parent node from each other. For the classification, only one classifier is queried on every layer of the hierarchy.

The final approach implemented by the library trains a single classifier for every layer of the hierarchy (see Figure 2.2.B). Similarly to the previous implementation, the classification is also performed top-down; for every layer, only the output with the label with the highest probability in the parent layer is enabled.

2.4 Classification algorithms

2.4.1 Naive-Bayes

Naive-Bayes is a probabilistic classification algorithm that functions under the assumption that each input variable is independent (which is also why it is called *Naive*, as this is an unrealistic assumption in real-world data)[3]. The algorithm relies on the following function:

$$P(c|x) = \frac{P(x|c) \cdot P(c)}{P(x)}$$

where $P(c|x)$ is the probability of input x belongs to class c, $P(x|c)$ the probability of input x given that it belongs to class c, $P(c)$ being the probability of class c and $P(x)$ being the probability of the input x.

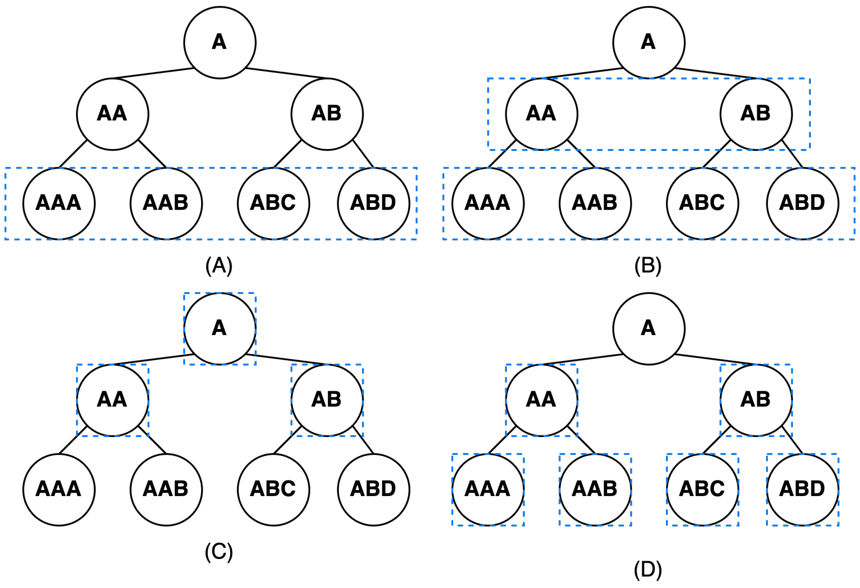


FIGURE 2.2: The different types of hierarchical classifiers are visualised, where each blue square represents a classifier. (A) is a non-hierarchical classifier, (B) is a local classifier per level, (C) is a local classifier per parent node, and (D) is a local classifier per node.

If the formula is calculated for all classes $c \in C$, the classifier can predict the class with the highest probability given the input x.

2.4.2 Support Vector Classifier

A Support Vector Machine (SVM) can be used for binary classification[3]. The theory behind the SVM classification approach is to separate multidimensional data using a maximum-margin hyperplane. This hyperplane, which is a plane that is 1 dimension lower than the input data, splits the training data into two subsets, each representing one class. The hyperplane is created to have the largest distance between the different classes (see Figure 2.3).

In practice, the distribution of the classes in a data set might have noise and show some overlap, which would make an exact separation of the data points impractical as the noise could cause a model to find a hyperplane with very low margins, which can lead to bad performance on unseen data. A solution that can be applied to ensure that the model finds a hyperplane with a larger margin is through the implementation of a penalty for misclassified samples. The penalty enables the model to maximise the margins of the hyperplane while still allowing the misclassification of some potential outliers, allowing the model to be more generalisable and preventing overfitting to the training data.

2.4.3 Perceptron

A perceptron is another algorithm to divide data into classes. Initially described by Rosenblatt[20], perceptrons are sometimes described as artificial neurons because they function similarly.

An example of a perceptron with three inputs is visualised in Figure 2.4. The input of the model, as well as the bias, is multiplied by the individual weights, after which the

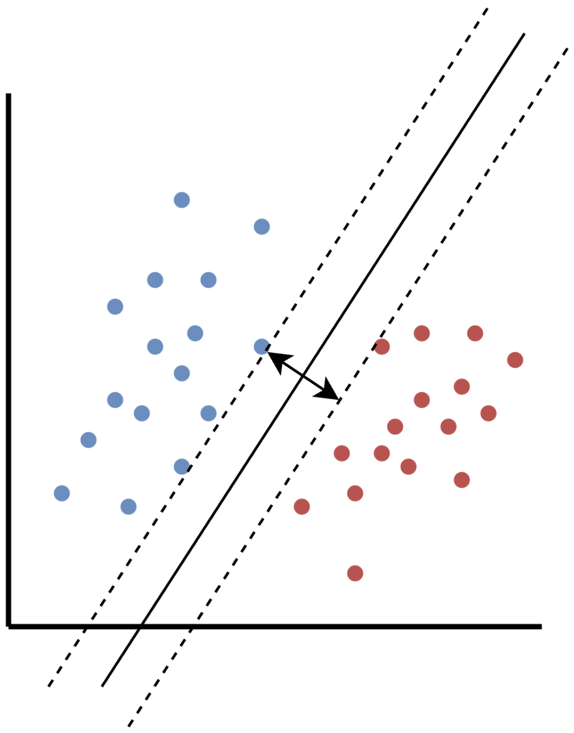


FIGURE 2.3: Example of an SVC fit to 2-dimensional data

resulting values are summed up and used as the input to the activation function.

For single perceptrons, the activation function is used to map the output value to be in the range $[0, 1]$, which is done through the *sigmoid* function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The output of the activation function can then be used for binary classification. All inputs where the output is greater than 0.5 are considered true, and the rest are labelled false.

Perceptrons can also be used for multi-class classification problems, which is achieved through the use of multiple perceptrons alongside each other, where the output of each perceptron represents the likelihood of a single class. The probability of an input being part of a class can be calculated using a softmax function[3]. The softmax function scales the output of all the perceptrons to values that sum up to 1 using the following formula:

$$\sigma(x) = \frac{output_i}{\sum_{j=1}^{K} output_j} \quad \texttt{for} \ i = 1, ..., K$$

Where K is the number of outputs.

The perceptron is trained by iterating over a training data set through a process of weight updates based on the output of the model. The weight update process is called backpropagation, which works by comparing the model output to the expected output and updating the weights linked to the incorrect output values in the direction of the expected result[3]. The process of iterating the training set and performing weight updates is called an *epoch* and is usually repeated multiple times before the model is fully tuned to the data.

When a perceptron model is trained, a model can overfit the data on which it is trained.

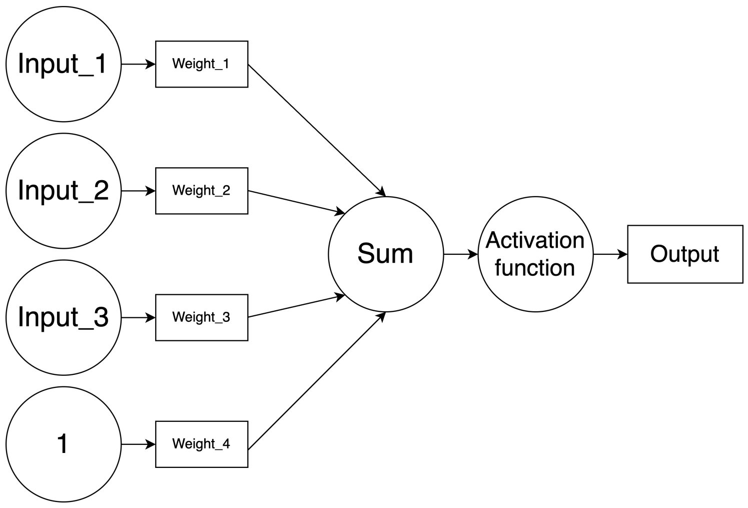


Figure 2.4: A perceptron with 3 inputs

Overfitting is caused by the model aligning itself too closely to the training samples, causing the model to be sensitive to features that are not specific to the entire class, but only to the training samples. Overfitting will negatively affect the general performance of the model, which should be prevented. Multiple approaches can be applied during the training process to prevent overfitting.

- Dropout disables a percentage of input variables which forces the model to learn to differentiate classes with different input data features, increasing the model's generalisability[22].
- Early stopping is another approach that prevents the model from overfitting the training data. In early stopping, a comparison is made between the loss of the validation set to the same loss of the previous epoch. If during the training, the validation loss does not improve over a predefined number of epochs, the training process is stopped, as further training will only cause the model to overfit on the training data[17].
- Weight decay, as described by Krogh et al.[14], implements a method to ensure that the weights of a model remain small. This helps to prevent overfitting, as large model weights can cause models to overfit on a single feature as it weighs heavily. Weight decay is implemented by including a fixed parameter during the weight update stage of the model training stage that penalises large weights.

2.4.4 Multi-layer perceptron

A limitation of perceptrons is that they can only be used to train data that can be linearly separated[3]. One way for a model to learn non-linear data is to apply multiple layers of perceptrons in series to create a multi-layer perceptron (see Figure 2.5). A multi-layer perceptron consists of an input layer, followed by one or more hidden layers, after which a final output layer is used to determine the output. By inserting all the outputs of intermediate layers into an activation function, a nonlinearity is inserted into the model (see Figure 2.6). The nonlinear activation function allows the network of perceptrons to learn nonlinear correlations in the training data so that more complicated distributions can

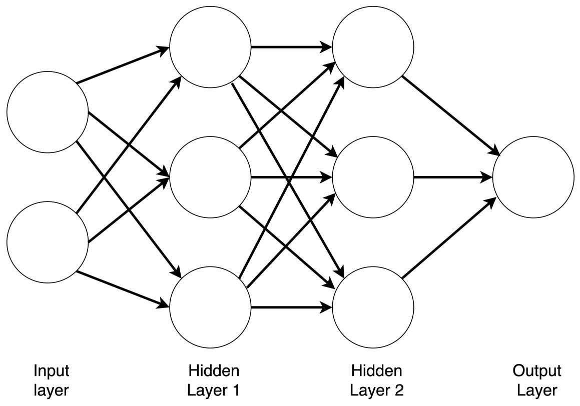


FIGURE 2.5: Example of a multi-layer perceptron

also be learned[9]. The multilayer perceptron is also often called a Deep Neural Network (DNN). The activation functions most commonly used are the Hyperbolic Tangent (Tanh) and Rectified Linear Unit (ReLU).

$$Tanh(x) = \frac{2}{1+e^{-2x}} - 1$$
$$ReLU(x) = max(0, x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise,} \end{cases}$$

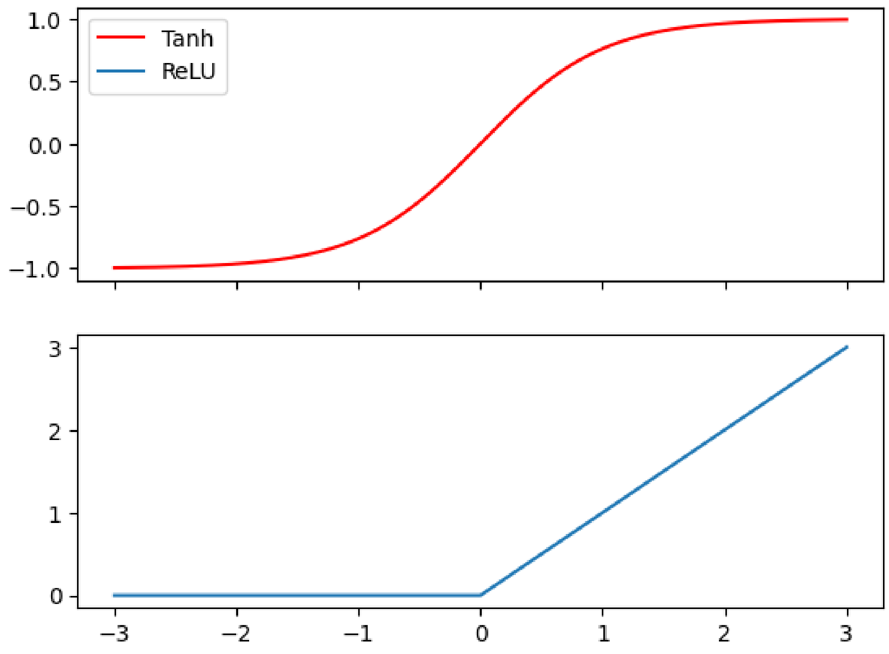


FIGURE 2.6: The input plotted against the output of the ReLU and Tanh activation functions

The networks are trained through the process of backpropagation, where the gradient of a loss function is calculated for every neuron, starting at the final layer and moving towards the first layer using the chain rule. Because applying gradient calculations on every sample is computationally expensive, most practical implementations of neural networks group the samples into batches, performing the backpropagation with the average gradients of all the samples in the batch. Batching speeds up the training process but is generally limited by memory usage, as all samples in a batch need to be loaded into memory at the same time.

An issue with the Tanh activation function is that it suffers from the problem of vanishing gradients, especially in networks with a large number of layers. With the vanishing gradient problem, the gradient of the Tanh activation function can become very small with very large positive or negative inputs. Since the gradients are multiplied by each other using the chain rule, the gradients shrink exponentially to the point that the layers at the start of the model are not updated effectively, which can lead to models not performing well or training extremely slowly.

The ReLU activation function does not suffer from vanishing gradients, as the gradient of an activation function is 1 or 0. The simplicity of the gradient also makes the backward pass of the training process very fast, as the gradient calculation can be performed very fast. One downside of the ReLU activation function is that it can suffer from the dying ReLU problem, where a neuron will not be taught anything since all its inputs are 0, effectively disabling the neuron. This problem is solved by changing the ReLU function to output a very small negative value when the input is negative. The most commonly used activation function that does this is the $LeakyReLU$:

$$LeakyReLU(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha \cdot x & \text{otherwise,} \end{cases}$$

Where α is set to a very small value such as 0.01

2.4.5 Neural network variations

Multi-layer perceptrons can be very effective for the classification of data. Sometimes, however, the shape of the input data does not (realistically) allow for the implementation of standard neural network implementations. In, for example, image classification, the input size of the model would become extremely large (a 128x128 rgb image would require 49.152 inputs). A convolutional layer can be used to reduce the number of trainable parameters. Compared to a fully connected layer (where every input is connected to each of the neurons in the next layer), a convolutional layer only connects inputs near each other to the neuron in the next layer (visualised in Figure 2.7). A neural network that uses one or more convolutional layers is called a Convolutional Neural Network (CNN)[3]. These networks can also be used for one-dimensional data, such as text, where the convolutional element can be useful for giving context to an input word by the words that surround it.

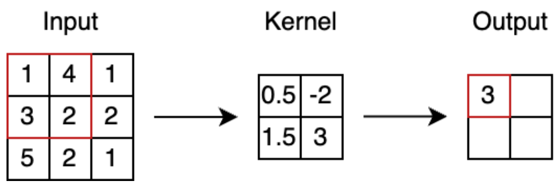


FIGURE 2.7: An example of a convolution

2.5 Text processing

Due to the unstructured nature of textual data, it is often necessary to process the raw text in a numerical form that is more suitable for computer processing. There are a large number of approaches to text processing, which can generally be split into contextual and non-contextual text processors. In contextual algorithms, the words surrounding the word being processed are taken into account when processing words. This can improve the quality of the algorithm output since words such as *right* can mean different things depending on the context (it can mean a direction, as well as something being correct). A downside of contextual approaches is that they can be computationally expensive compared to non-contextual approaches, as the surrounding words must be considered for every word in a sentence. Non-contextual algorithms do not take the surrounding words into account when processing sentences, which generally allows non-contextual models to be faster in their processing.
In the remainder of this section, several text-processing algorithms are described.

2.5.1 TF-IDF

Term Frequency Inverse Document Frequency (TF-IDF)[21] is a non-contextual approach that counts the occurrences of words in documents and multiplies that by the inverse of the number of documents in which the word occurs. The formula consists of two parts:

$$\begin{aligned} TF(word, sentence) &= \frac{frequency(word, sentence)}{length(sentence)} \\ IDF(word) &= log_e(1 + \frac{count(sentences)}{count(word \subset sentences)}) \\ TF\text{–}IDF(word, sentence) &= TF(word, sentence) * IDF(word) \end{aligned}$$

For the invoice lines, each invoice line is seen as a document. To calculate the values, a sparse bag-of-words matrix of size $num\_words \times num\_documents$ is created, where each column represents a word in the data set, and the rows contain all the documents. For each document, the values of the columns represent the number of times the word occurs in that document (see Table 2.2). The values of each column representing the words are then multiplied by the inverse document frequency. The inverse document frequency describes the number of documents in which a term occurs, making words that occur frequently weigh less, causing commonly found words to have less impact in the classification.

Sentence	I	He	They	like	likes	dogs	cats
I like dogs	1	0	0	1	0	1	0
He likes cats	0	1	0	0	1	0	1
They like cats	0	0	1	1	0	0	1

Table 2.2: Example of a term frequency table.

Since the bag-of-words approach uses the exact spelling of words as unique words, it will see variations of words through inflection as completely different words (as visualised with the words *like* and *likes* in Table 2.2). Inflection will not only harm the performance of classification algorithms, but will also increase the size of the matrix and therefore increase computational complexity. To limit the vocabulary and combine the inflections of words, it is common to apply preprocessing to the textual data before performing the vectorisation to reduce the vocabulary of the data set.

Text regularisation methods

The most simple approach to reduce the number of unique words in a data set is to remove special characters and to change all characters to lowercase. This will ensure that, for example, the first word of a sentence is no longer considered different from the same word occurring later in a sentence.

Words such as *"the"* and *"that"* are very commonly used in the English language; they occur so often that they cannot be linked to a specific class when applying text classification. Therefore, stop words are commonly removed in text preprocessing when applying TF-IDF.

With lemmatisation, the words are changed to remove inflections. Lemmatisation converts words such as *"improving"* and *"improvements"* to *"improve"*, effectively grouping them. This is done using a dictionary-based approach because the lemma of a word could be completely different from the original word (for example, the lemma of *"better"* is *"good"*).

An alternative to lemmatisation is to apply stemming. In stemming, the last few characters of words are removed to get the stem in a rule-based approach. The stem does not have to be a grammatically correct word. For example, both *history* and *historical* would stem to the word *histori*. The application of rules is generally faster compared to the dictionary-based approach applied with lemmatisation, but it is generally less consistent and more prone to errors.

2.5.2 FastText

FastText is an open source text embedding library developed by Facebook[4]. It is designed to be a lightweight but performant embedding algorithm.

FastText is trained using a continuous skip-gram model, which calculates the words that are most likely found in the context of an input word. This is achieved by maximising the following function:

$$\sum_{t=1}^{T}\sum_{c\in C_t} log(p(w_c|w_t))$$

Where C_t is the set of words surrounding the word w_t. The optimisation of this formula is achieved through several binary classifications of a set of words containing the actual context words, as well as a set of negative samples.

Furthermore, FastText implements a character n-gram model, which splits words w into several substrings. For example, taking $n = 2$ and the word *cats*, the word is represented by the following n-grams:

```
<c, ca, at, ts, s>
```

Where < and > are tokens used to represent the start and end of words. Character n-grams allow the model to learn information about sub-words, which can give information about words outside of a model's vocabulary, as well as reduce the effect of spelling errors and inflections.

2.5.3 BERT

In 2018, Devlin et al.[8] proposed the Bidirectional Encoder Representation Transformers (BERT) model. BERT is a state-of-the-art ML framework that can be used for many NLP tasks. The ability for the model to be applied to a large number of tasks is achieved by splitting the training into two phases, namely, a pre-train phase and a fine-tuning phase. In the pre-training phase, the model is trained through two different tasks using unlabelled data. The first pre-training task is to predict masked words in sentences. During training with this task, 15% of the word tokens are chosen to be replaced by a [MASK] token (80%), a random token (10%) or the unchanged token (10%). The model is tasked with predicting the masked token
The second step of the pre-training process is *next sentence prediction*, which is done through a binary classification task in which two sentences are placed in the model; the model is tasked with predicting whether the sentences follow each other or not.

After the pre-training, the model can be used for a variety of NLP tasks through the application of fine-tuning. The pre-trained model can be adapted to return word vectors of all the words through the removal of the final layer used for pre-training. With the final layer removed, any model can be attached to the BERT output, which can then be fine-tuned on task-specific data. Appending a new *head* to the pre-trained model allows the model to keep the original model weights and, with that, knowledge of the language on which it was trained, making the fine-tuning step relatively quick and simple.

Similarly to FastText, BERT models can also create embeddings for out-of-vocabulary words. It achieves this through the application of sub-words. If a word does not appear in the model's vocabulary, the model looks for sub-words that are in the vocabulary and splits the word into multiple sub-words. For example:

```
"That went smoothly"
["That", "went", "smooth", "##ly"]
```

In the example, the word smoothly does not appear in the vocabulary, but *smooth* and the subword *ly* do appear, allowing for a better representation of the input over the exclusion of out-of-vocabulary words. (The *##* represents that the token is a continuation of the last word.)
The tokenizer is trained through the process of Byte-Pair encoding (BPE). For a BERT tokenizer, the words are divided into characters, after which the most common pair of consecutive characters are grouped. The grouping of characters is repeated until the desired vocabulary size is reached. The process is visualised by the following example, where the numbers represent the occurrences of words in the vocabulary.

```
[("deck", 3), ("buck", 1),
("stuck", 5), ("stun", 3)]

[(["d","e","c","k"], 3), (["b","u","c","k"], 1),
(["s","t","u","c","k"], 5), (["s","t","u","n"], 3)]
```

```
[(["d","e","ck"], 3), (["b","u","ck"], 1),
(["s","t","u","ck"], 5), (["s","t","u","n"], 3)]

[(["d","e","ck"], 3), (["b","u","ck"], 1),
(["st","u","ck"], 5), (["st","u","n"], 3)]

[(["d","e","ck"], 3), (["b","u","ck"], 1),
(["stu","ck"], 5), (["stu","n"], 3)]

...
```

After the publication of BERT, researchers have worked on further optimisation of the design. In 2019, Liu et al.[15] proposed the Robustly Optimised BERT Approach (RoBERTa). They propose changes such as a dynamic implementation of the data masking during pre-training, a larger batch size, and a byte-level character encoding. These changes increased performance on the SQuAD data set by a few percent compared to BERT.

In the original BERT implementation, the masking process was applied only once at the start of the pre-training, causing the model to be trained on identical masks for every epoch. For RoBERTa, the researchers dynamically applied the masking for every epoch, resulting in a small performance improvement over BERT.

The researchers also determined that removing the next-sentence prediction task used in the pre-training of BERT does not decrease the performance of the model on downstream tasks. Therefore, they decided to eliminate this step.

The byte-level character encoding for the input tokenisation differs from that of BERT in that BERT uses a character-based approach. This approach has the downside that Unicode characters take up a large amount of the vocabulary since there exist 149,186[1] different Unicode characters. To increase the number of useful character combinations in the vocabulary, the researchers developed RoBERTa with a byte-level BPE and increased the vocabulary size over that of BERT (50K vs. 30K). This change ensures that there are no “unknown” tokens, as all possible bytes are included in the data set.

Finally, RoBERTa was trained with a larger data set with larger mini-batches. The batch size was increased to 8K sequences per batch (256 in BERT). The increase in batch size allowed their model to achieve better performance with 500K steps compared to 1M steps of BERT training.

Delobelle et al.[6] used the RoBERTa architecture to develop RobBERT, a pre-trained language model trained on the Dutch subset of the OSCAR[1] data set, a large web-crawled set of unlabelled documents. Their implementation outperforms other Dutch and multilingual models such as BERTje[5] in die / dat disambiguation, as well as sentiment analysis.

[1]https://www.unicode.org/faq

2.6 Preprocessing

For some data types, it is important to preprocess the input data in a way that a computer can interpret. For example, most machine learning algorithms cannot interpret categorical values, for which one-hot encoding needs to be applied.

In one-hot encoding, a column of categorical variables is converted into multiple true/-false columns (visualised in Table 2.3) where the number of resulting columns is equal to the number of unique values for the categorical variable. Intuitively, one might think that assigning a number ($Dog = 1, Lion = 2, Cat = 3$) would be a more efficient way to generate these encodings. However, machine learning algorithms consider the input values as continuous, which means that the model would consider the class *Dog* closer to *Lion* than to *Cat*.
A similar approach to one-hot encoding can also be applied to situations where multiple values are true for each sample. This is called multi-label binarisation and is visualised in Table 2.4.

Variable	**encoding**		
	A	**B**	**C**
A	1	0	0
B	0	1	0
C	0	0	1
B	0	1	0
A	1	0	0
C	0	0	1

TABLE 2.3: The conversion of a single column of categorical variables into their one-hot encodings

Variable	**encoding**		
	A	**B**	**C**
[A,B]	1	1	0
[B]	0	1	0
[A,C]	1	0	1
[B,C]	0	1	1
[∅]	0	0	0
[A,B,C]	1	1	1

TABLE 2.4: The conversion of a column containing arrays of categorical variables into multiple binarised columns

2.7 Evaluation

To be able to evaluate the results of the different approaches, evaluation metrics have to be selected.

Most of the methods used to evaluate classification algorithms in machine learning use *true positive* (TP), *true negative* (TN), *false positive* (FP), and *false negative* (FN) as a basis. These values are calculated by comparing the predictions made by a model with the ground truth values. Table 2.5 shows a confusion matrix which shows which label applies to the possible combinations of ground truth and predictions. By applying this labelling to all the predictions made by the model, the total values for the metrics can be calculated.

Accuracy is a metric commonly used in machine learning, which is calculated using the following formula:

$$acc = \frac{TP + TN}{TP + TN + FP + FN}$$

The accuracy formula takes all the correctly classified predictions and divides them by the total number of predictions. Although this method generally works well, it would not

Actual label / **Predicted label**	**Positive**	**Negative**
Positive	TP	FN
Negative	FP	TN

TABLE 2.5: A confusion matrix

work well when using unbalanced data sets, as the classes that occur infrequently will also have a very small influence on the result. As an example, if a data set has 100 samples, of which 99 classes are labelled *positive* and one *negative*. A model that naively always predicts the *positive* class for all samples would still achieve an accuracy score of 99%, even though the model does not consider the *negative* class.

Zhao et al.[24] used metrics such as *Recall*:

$$Recall = \frac{TP}{TP + FP}$$

which is used to minimise the number of false positives (the number of negative samples classified as positive) and *Precision*:

$$Precision = \frac{TP}{TP + FN}$$

which is used to determine the number of positive samples that are correctly classified. These metrics can be combined to calculate the *F-score* (with a value β to determine the importance of recall) using the following function:

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

The value for β is commonly set to 1, giving the following formula for the *F1 score*:

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot TP}{(2 \cdot TP) + FN + FP}$$

For ML models that are used to generate recommendations, it can be interesting to evaluate a model based on its top K most likely classes, which can be done through the *Top_K_accuracy*:

$$Top\_K\_acc = 1(y \in \hat{y}_1, \hat{y}_2, ..., \hat{y}_K)$$

Where $\hat{y}$ is a list of predicted classes, sorted by probability

One of the ways that data can be evaluated through a visualisation is with a boxplot. Boxplots show how a variable is distributed by plotting the interquartile range, as well as outliers. An example boxplot is described in Figure 2.8.

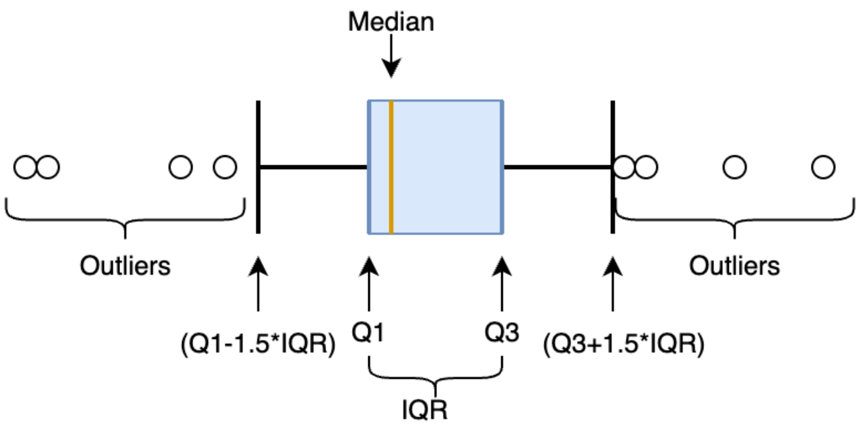


FIGURE 2.8: An example boxplot and its structure

2.7.1 Evaluation for multi-class classifiers

Both of the evaluation metrics mentioned above have functions that are designed for binary classification and need some adaptation to use for multi-class classification. For accuracy, the solution is generally to change the formula to be the following:

$$Acc = \frac{num\_correctly\_predicted}{num\_predicted}$$

For the F_β-score , a more common approach is to apply an averaging method. These approaches calculate the F_β-score for each class individually in a one-versus-rest approach. With the one-versus-rest approach, the multiclass classification problem is converted to several binary classification problems. For example, given the classes `[ dog, cat, sheep]`, the one-versus-rest evaluation would calculate 3 binary evaluations, namely: `(dog, [cat, sheep]), (cat, [dog, sheep]) and (sheep, [dog, cat]).`

When using *Macro* averaging for F_β-score , all scores are averaged. Every class is counted equally heavily, independent of the number of samples in the class. Alternatively, *weighted* averaging calculates the score based on the number of samples in each class. In practice, the difference between these approaches is that *weighted* F_β-score can be high even if a minority class is never correctly classified by a model, while *macro* F_β-score would better reflect the lower performance on uncommon classes.

2.7.2 Cross validation

To compare multiple approaches, the data set is divided into a train and a test set. The split allows the approaches to be compared on an identical test set. Afterwards, a K-fold cross-validation is applied to ensure that the results for individual models are consistent. Cross-validation divides the train set into multiple sets of training and validation sets, as visualised in Figure 2.9. By creating multiple splits on the training data, individual results of the model can be validated while still allowing models to be compared on the same test set.

2.8 Related works

Recently, researchers have investigated the classification of invoice data into ledger codes in several countries.

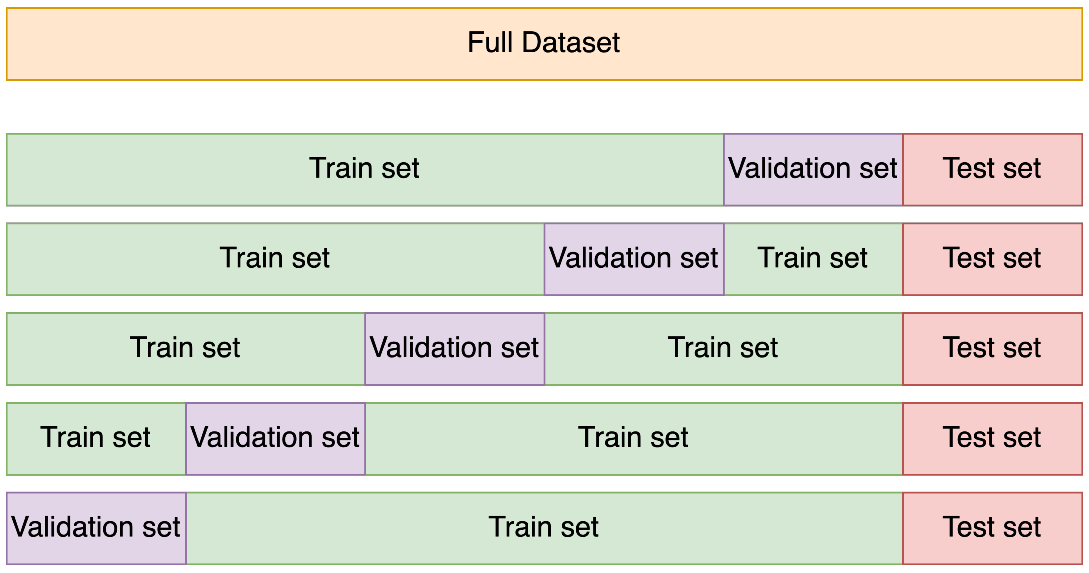


FIGURE 2.9: Example of a K-Fold cross-validation split with $K = 5$

Bergdorf [2] investigated the classification of Swedish electronic invoices comparing rule-based approaches to random forest and Support Vector Machine (SVM) approaches. Their work is similar to this work in that it implements a system for the classification of ledger account codes. However, a major difference is that in their research, Bergdorf trains a classification model on data from a single organisation, whereas in this thesis a model designed to function for many organisations is implemented. Another difference is that their work does not include the use of the textual description of the invoice line, instead focussing on contextual data such as the financial year and VAT codes. Training a model for each organisation could lead to improved performance, but would require a large amount of validated data to be available from every organisation before the model starts to become effective in its predictions. A consequence of this design choice is that it would most likely also make the model ineffective for new organisations that are starting with the software. Training a model specifically targeted towards an individual organisation also has an advantage. Bergdorf included company-specific information, such as financial year, in their data set, which could be relevant to an individual organisation, as companies might have started buying certain types of products starting from a specific year. However, the financial year of the invoice would not be relevant to a model designed for multiple organisations, as the number is not generalisable for multiple companies. Bergdorf's results show a similar performance between the different approaches, with random forest being the model that slightly outperformed the alternatives with the highest accuracy and F_2 score.

Munoz et al.[18] used a hierarchical classifier to predict ledger account codes in invoice data in Australia. Similarly to Bergdorf, their research focuses on the classification of invoices for a single organisation. In their work, they compared different text processing approaches, comparing Word2Vec, bag-of-words, and two versions of a BERT model. Both BERT models were pre-trained, but only one of them was further fine-tuned on the target data. Their research showed that the fine-tuned BERT model performed the best. They also compared classification methods, comparing a hierarchical deep learning model to a non-hierarchical deep learning network, a logistic regression model, and a random forest model. The hierarchical deep learning model performed the best, with a macro F_1 score of 0.568, followed closely by the non-hierarchical deep learning network which achieved a score of 0.543. In their work, they also compared the training time of the different deep neural networks, which showed that the non-hierarchical deep learning network was the fastest to train. For their classifier, they used a data set consisting of 31.332 invoice lines, with 239 target classes ($\pm$131 samples per class on average). A similarity of their work to this research is the data imbalance, with some classes occurring only once, while other classes had more than 3000 samples. The problem of data imbalance was further complicated by the way their data was split into training and test sets, which were performed by date (before and after September 2020). As a result, 17 classes never appeared in the training set, which were present in the test set. The approach differs from the approach taken in this research, where the classes were limited to ensure that each class had a large enough sample size to prevent the model from overfitting.

Johansson[12] investigated the classification of ledger account codes in Swedish invoices. Their research also limited their data set to that of a single company, which used a total of 39 different ledger account codes. From these account codes, the labels that occurred more than 50 times were kept for classification, with the other classes grouped to form an "other" class, which left 21 classes for their classification experiment. In their experiment, Johansson used an n-gram bag-of-words TF-IDF algorithm to create the textual embeddings for descriptions in combination with "shallow" machine learning algorithms such as SVM, Naive-Bayes (NB), Logistic Regression (LR) and Random Forest (RF). The researcher investigated the application of stopword removal and stemming in their data preprocessing, but claimed that the improvement these steps brought was negligible (up to 1%) and therefore not worth the extra computation time. Their implementation achieved the highest accuracy using a linear SVM classifier, which achieved a macro F_1 score of 0.75. However, because of the low sample size for some of the classes (the minimum was set at 50 samples), it is uncertain whether these results are representative of the true performance of the model on a larger data set. Furthermore, in contrast to our research, the research performed by Johansson only investigated the classification of the textual description of invoice lines, leaving out potentially important information from contextual data.

Kiekbusch et al.[13] applied a CNN on the description of invoice lines to predict ledger codes in Brazilian invoice data. For their experiments, they collected data from multiple companies, keeping only samples that were labelled with a class that occurred more than 2000 times. The data were then undersampled to balance the data set, which was done by removing data points from the classes with a high sample count until all classes were of equal size. For their first experiment, they compared the performance of a single multiclass CNN to an ensemble network of binary CNNs. The ensemble network contained a binary classifier for each class, which could classify a sample as positive or negative to be part of that class. These models were merged into an ensemble approach that was fine-tuned. The results of their experiment showed that the accuracies of both models were similar but that the ensemble model achieved a higher precision score, at the cost of a lower recall score.

In their second experiment, they compared a word-based CNN and a character-based CNN to a baseline model that uses TF-IDF and an SVM model (the kernel was not specified). Both CNN models performed significantly better than the SVM approach, with a minimal difference in accuracy between the character-based and the word-based

CNN.

Similarly to the work by Johansson[12], their research focused only on the classification of invoice lines based on the textual description, leaving out contextual information. Another difference in our research from theirs is the choice to apply balancing to the data set, as well as the implementation of a CNN for the classification.

2.9 Research gap

The previous work shows a variety of approaches that have been used for the prediction of ledger codes in invoice data. The most obvious difference between the related works and the proposed research is that our research will focus on data in the Dutch language, as opposed to English, Swedish, and Brazilian. Furthermore, only one of the previous works investigated the creation of a system that was designed to predict ledger codes for several companies, as opposed to a model for an individual company. Their research focused on implementing a CNN and mentioned that future work should investigate an approach that involved transfer learning, which is investigated in our research through the implementation of a classifier based on BERT.

Chapter 3

Methodology

3.1 Data processing

This chapter describes the collection and preprocessing of the data set, as well as the methods applied for the experiments. For the data mining process, the first five steps of the CRISP-DM process [23], developed by Wirth and Hipp and described in Section 2.1, are followed. The business understanding, data understanding, and data preparation phases are described for all of the experiments together, while the modelling (this chapter) and evaluation (Chapter 4) phases are described individually for each of the experiments. An overview of the research activities that have been conducted in this research is shown in Figure 3.1. The diagram is divided into four groups; the first steps are *Data collection and cleaning*, where we examine the structure of the data and explore potentially relevant contextual data. The steps taken for this part of the research are described in Section 3.1.2. Data collection and cleaning are followed by the *Data pre-processing* phase, where we clean up and process the collected data to prepare it for later classification. The pre-processing steps are described in Section 3.1.3. In the third phase of the research, the models are developed and the results of the research questions are generated. The methodology for the development of the model is described in Section 3.2, and the results are described in Chapter 4. The final phase of this research is the *Prototyping* phase, where the best model developed in the model development phase is implemented into the existing bookkeeping system. The implementation of the architecture is described in Chapter 5.

3.1.1 Business understanding

The research carried out is part of a project created by a bookkeeping software company to improve and simplify the bookkeeping process of their customers. In their current implementation, customers have to choose from a list of categories they have created themselves, which are linked to RCSFI codes. The current implementation, as visualized in Figure 3.2, shows that the user chooses the category from a drop-down menu containing all the available options. Since the number of available categories can be very large, selecting the correct ledger code for an invoice line can be a complicated and error-prone process. Therefore, the company is interested in innovating by implementing a system that can aid the user in selecting ledger codes by implementing a machine learning-based classifier. The classifier could use information such as invoice line descriptions to make its predictions. Although the bookkeeping software manages both sales and purchase invoices, this research is focused only on the classification of purchase invoices. The decision was made to only use purchase invoices for classification because they are significantly different

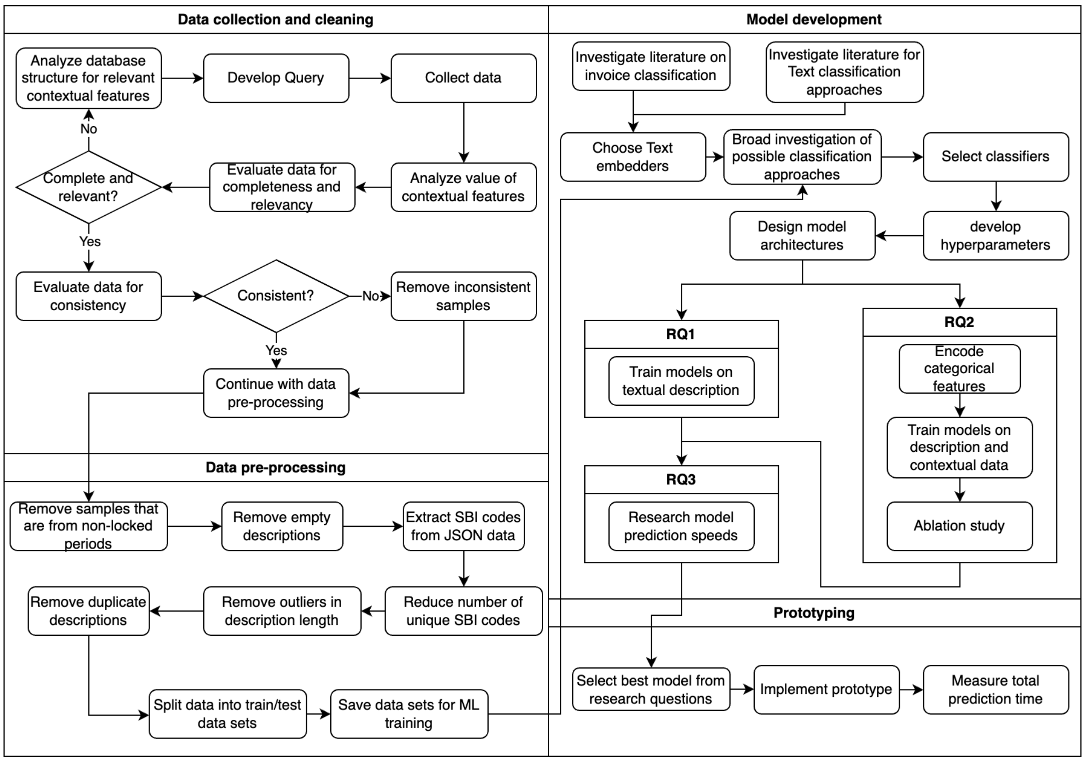

Data collection and cleaning
Analyze database structure for relevant contextual features
Develop Query
Collect data
No
Complete and relevant?
Evaluate data for completeness and relevancy
Analyze value of contextual features
Yes
Evaluate data for consistency
Consistent?
No
Remove inconsistent samples
Yes
Continue with data pre-processing
Data pre-processing
Remove samples that are from non-locked periods
Remove empty descriptions
Extract SBI codes from JSON data
Remove duplicate descriptions
Remove outliers in description length
Reduce number of unique SBI codes
Split data into train/test data sets
Save data sets for ML training
Model development
Investigate literature on invoice classification
Investigate literature for Text classification approaches
Choose Text embedders
Broad investigation of possible classification approaches
Select classifiers
Design model architectures
develop hyperparameters
RQ1
Train models on textual description
RQ2
Encode categorical features
Train models on description and contextual data
Ablation study
RQ3
Research model prediction speeds
Prototyping
Select best model from research questions
Implement prototype
Measure total prediction time

Figure 3.1: A research activity diagram showing the steps taken in this research.

from each other in the bookkeeping software. As a result, both types of invoices cannot be combined in a single model, and the purchase invoices showed a greater variety of ledger codes per company (visualised in Figure 3.3). The software company focuses its software on smaller companies that generally use the bookkeeping software only to deliver a single type of product, which means that only one ledger code is used for all sales invoices of that company. Since the number of unique ledger codes in purchase invoices is significantly larger, these are more interesting to create a prediction model for.

3.1.2 Data understanding

In this section, we examine the data set, its features, and data quality. The data set is collected using the database of the bookkeeping software company that contains invoice data. The data set was collected using an SQL query that returned one row of data per line on the invoices. Since the information in the invoice lines could be sensitive, customers of the bookkeeping software can give explicit permission in the application for their data to be used for analytical purposes. Therefore, the invoice lines were only collected from the administrations that have given that explicit permission. The fields collected by the query are described in Table 3.1.

The results of the query were exported to a Comma Separated Values (csv) file, and loaded into a *pandas DataFrame* for analysis. An example invoice describing the origin of the features, including contextual features, can be found in Figure 3.4.

The complete data set contains 13.318.544 samples, each of which is a single invoice line. As an invoice can have multiple lines, the total number of unique invoices is 11.133.564, resulting in an average of ±1.2 invoice lines per invoice.

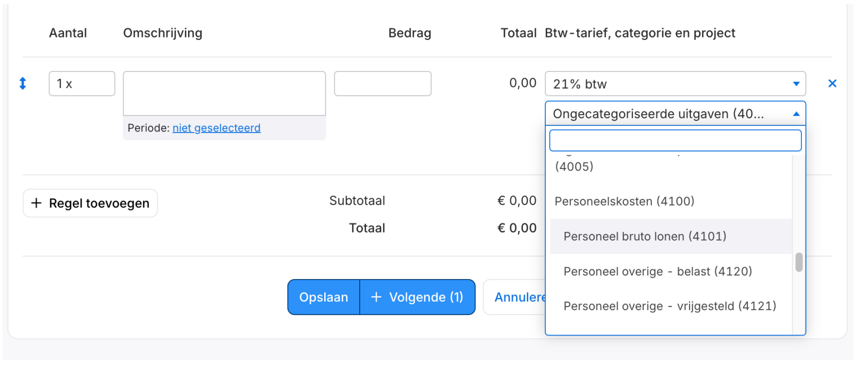

Aantal
Omschrijving
Bedrag
Totaal
Btw-tarief, categorie en project
1 x
0,00
21% btw
Ongecategoriseerde uitgaven (40...
Periode: niet geselecteerd
(4005)
Personeelskosten (4100)
Personeel bruto lonen (4101)
Personeel overige - belast (4120)
Personeel overige - vrijgesteld (4121)
+ Regel toevoegen
Subtotaal
€ 0,00
Totaal
€ 0,00
Opslaan
+ Volgende (1)
Annulere

FIGURE 3.2: The current implementation in the bookkeeping software to select a category for an invoice. In this example, there are 26 options available for the user to choose from.

The target variable for the data set is the RCSFI code, of which the data set contains 736 unique values. Upon further examination, it showed that the data set contains a large imbalance in sample count per label (see Figure 3.5).

For the first experiment, classifiers will be trained using only the invoice line descriptions. These descriptions are used to describe the product purchased by the company. These descriptions are generally brief or are sometimes just the order number. The word count of the invoice descriptions is described in Figure 3.6

Data quality

In this section, we describe our analysis of the completeness, uniqueness, and consistency of the data. The data accuracy is difficult to assess for the data set due to its size and sometimes abstract product descriptions in the invoice.

A calculation of the null values for each of the columns shows that the data set contains 17 empty descriptions (0.0001% of the full data set), 2.291 empty *administration legal entity type* fields (0.0172%), 114.303 empty contact names (0.85822%), and 1.599 empty administration SBI fields (0.0120%). The other columns did not contain empty values.

The description column was examined for the uniqueness of the data. There are 3.420.817 unique descriptions in the data set (25.6846% of the entire data set). To examine data consistency, unique combinations of the description and RCSFI columns were calculated. There are 3.802.139 unique combinations of descriptions and RCSFI. As a consequence, there exist 381.322 rows in which the same description is attached to another label in the data set. If the other fields in the data set are included in the comparison (excluding the date values), a comparison shows that there are 76.864 rows in which another sample existed with the same features but that are labelled differently.

3.1.3 Data preparation

In this section, we describe the steps taken to prepare the data for classification, as well as the approaches taken to improve the quality of the data.

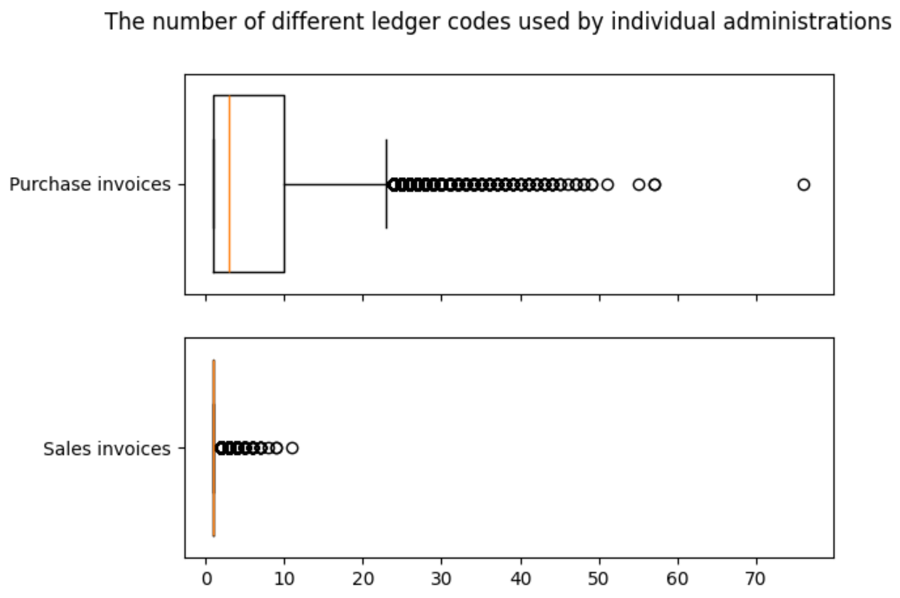


FIGURE 3.3: A boxplot showing the number of unique ledger codes used by each administration in the data set. This graph shows that companies generally only use a single ledger code for their sales invoices, whereas the number of unique ledger codes used for purchase invoices is significantly higher.

As described in Section 3.1.1, the purpose of the proposed model is to predict the correct ledger code. In the data set, the most frequent code is *WBedAlkOal*, which can be translated to *"General costs"* and is used as the default ledger code by the bookkeeping software for purchase invoices and receipts. The label does not specify what type of purchase is made, and this label should only be used when there is no other label that matches the type of purchase. Since the label does not indicate a specific expense category, it is not relevant to be able to achieve the research goal. As a result, all data samples labelled with this label have been removed, shrinking the data set from 13.318.544 to 8.234.789 invoice lines.
After this step, the samples that contained an empty description are removed (six occurrences). These were removed as the classifiers trained in the first experiment only used the textual description, meaning that an empty description would make it impossible for a model to classify the sample. Invoice lines with descriptions longer than 25 words have been removed because 99.5% of the descriptions contained less than or equal to 25 words (see Figure 3.7), and manual inspection showed that a large number of invoices with a higher word count were incorrect data. For example, an entire invoice (multiple invoice lines and therefore multiple ledger codes) was inserted into the description field of a single invoice line. The removal of these outliers reduced the data set from 8.234.783 to 8.221.251 samples.

Duplicate descriptions have been removed because it would weigh the descriptions with multiple occurrences heavier as the model is trained on that specific sample multiple times per epoch, or data leakage if the model sees the same sample in both the training and testing phase. Due to this, duplicate descriptions are removed. This filtering does not take the labeling of the sample in account. In practice, the removal results in only

Feature	**Description**	**Type**
Invoice line description	A piece of text used to describe the product or service provided for the invoice line	Textual
Invoice line price	A field used to express the cost of the individual invoice line	Numerical
Document type	A field that describes whether the document is either a purchase invoice or a scanned receipt	Boolean
Currency	The currency used in the invoice	Categorical
Administration lock date	A field used to describe up until which date the invoices are locked for an administration	Date
Invoice date	The date on which the invoice was sent	Date
Administration legal entity type	A field that describes the type of company that receives the invoice	Categorical
Contact company name	The company name of the company that has sent the invoice	Textual
Invoice line tax rate	The rate under which the product or service is taxed	Categorical
Administration SBI codes	A set of codes that describe the fields in which the administration operates	Categorical
Invoice line RCSFI code	A field that describes the bookkeeping category of the invoice line	Categorical

Table 3.1: The fields collected from the database

the first sample being kept, an approach that does not consider whether the sample is labelled correctly, and it is possible that only a mislabelled sample is kept. The removal of duplicated data reduced the size of the data set from 8.234.783 to 2.040.088 samples.

It is infeasible to manually validate every individual sample for its data accuracy. A major concern for the data set is the possibility of mislabelled data points, which limits the potential effectiveness of a model. The problem of mislabelled samples cannot be completely prevented, but some steps have been taken to improve the quality of the data set.

In the software, companies enter a ledger code when creating invoice lines. However, a company can update the invoice lines to correct errors and change the ledger code after sending the invoice. Consequently, it makes it so that there is no guarantee of the correctness of the labelling. However, once a bookkeeping year has closed, it should be impossible to change the ledger codes (since these have already been processed). To ensure that ledger codes in closed bookkeeping years cannot be changed, bookkeeping software has made it possible to lock a period of invoices. For some companies, bookkeeping years are closed after an accountant has validated the invoices and ledgers. This makes it interesting to limit the data set to only invoices for which the creation date is within a locked period. Since these invoice lines are locked, the assumption can be made that they have been validated by the company. Verification of these invoices should reduce the number of mislabelled samples, increasing data quality. The removal of the samples that are not in a company's locked period reduced the data set from 2.040.088 to 586.642 samples.

To ensure that the sample size of every class in the data set is sufficiently large, the

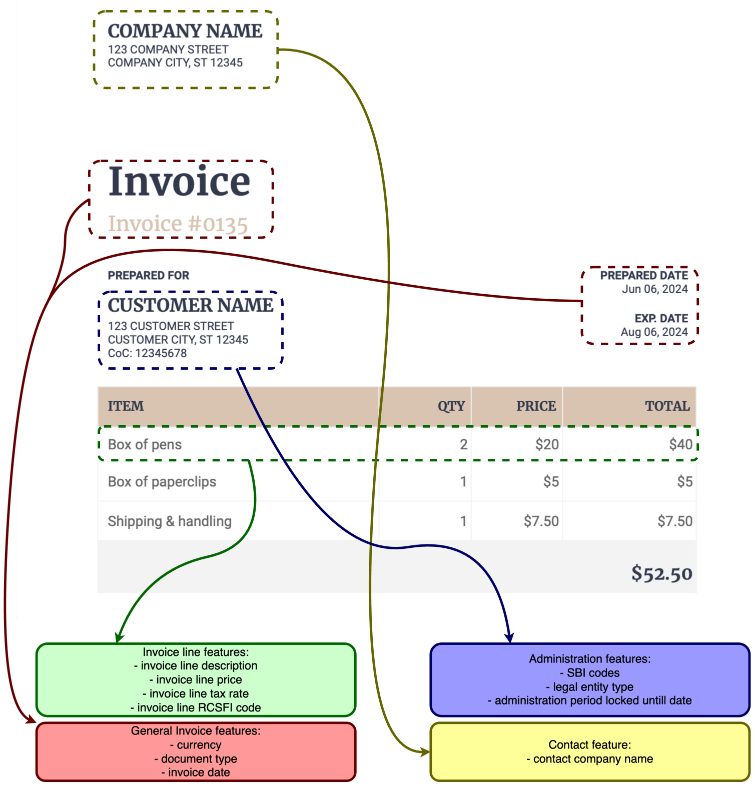


FIGURE 3.4: An example invoice describing the features collected in the query and their origin.

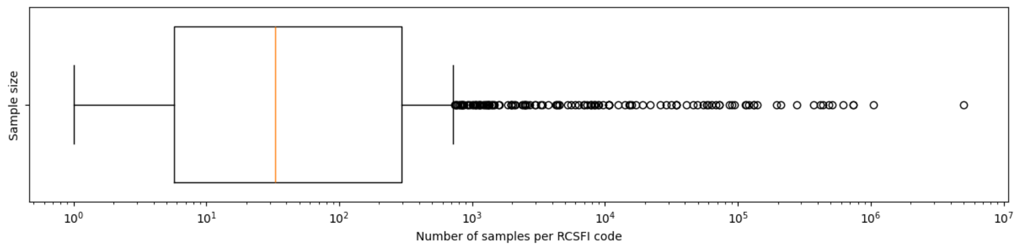


FIGURE 3.5: A boxplot showing the class imbalance in the full data set

classes that occur very rarely are removed from the data set. The same approach was taken by Kieckbusch et al.[13], where the minimum number of samples required for a class to be included in the training was set to 2000. However, for our research, the number of samples required was established at 5000, as the dimensionality of the data set was greater due to the inclusion of contextual data in the experiments. The removal of uncommon labels reduces the data set to 518.939 samples and 28 classes. The final ledger codes used and their hierarchical structure, as well as English descriptions, are described in Appendix A.

The total number of SBI codes available in the taxonomy is 943. To reduce the unique number of codes, the choice was made to generalise the codes to their first two digits. The simplification of the SBI codes limited the total number of unique values to 80 in the data set.

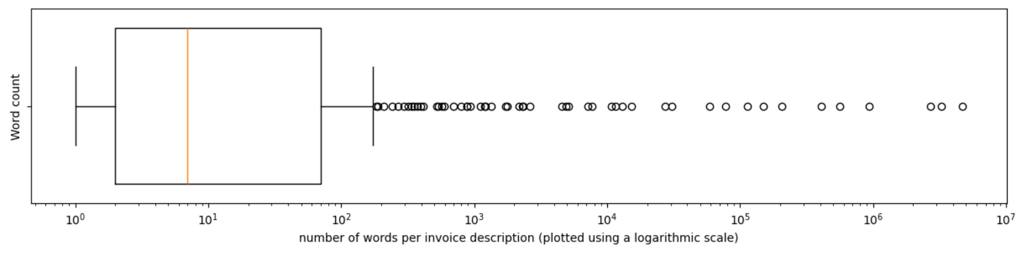


FIGURE 3.6: A boxplot of the number of words in the description field of invoice lines

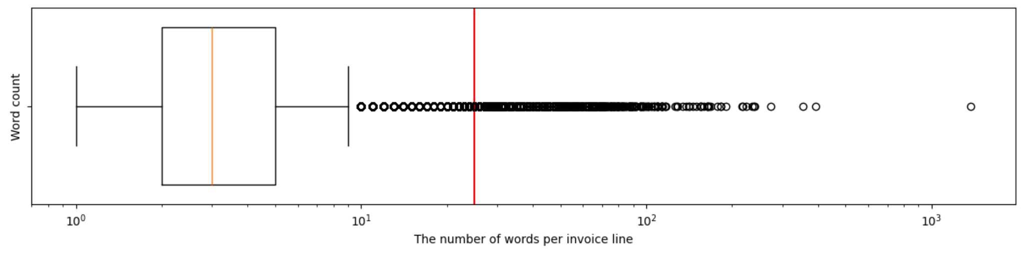


FIGURE 3.7: A boxplot showing the word counts of invoice descriptions, with the red line set at 25 words

3.2 Experiment setup

The research is divided into three experiments. The goal of the first experiment is to examine the performance of classification models trained to predict the ledger code that should be attached to the invoice lines. The input for these models only contains a short textual description of the product/service. The second experiment continues on the results of the first experiment by adding contextual information to the input of the models and researching the performance increase that contextual information can deliver. The final experiment examines the prediction speed of the different models, examining approaches to speed up predictions, and their impact on prediction performance. These results are then used to implement a classification system for the invoice data.

3.2.1 Experiment 1

The objective of the first experiment is to investigate the optimal approach to classifying invoice lines into ledger codes using only textual descriptions of invoice lines. For this to work, the descriptions need to be converted to a different representation that is better suited for machine learning. Our goal is to compare contextual and non-contextual approaches. We hypothesise that contextual approaches to text embedding will not significantly improve the performance of textual classification for invoice data since the pieces of text are generally very brief, and therefore lack the contextual information that the contextual embedders utilise. Non-contextual embedders do not rely on contextual data, which should increase the classification speed of these models, which would also make these models more suited for a practical implementation where prediction speed is essential.

To test this hypothesis, three embedders were selected for this experiment. A RoBERTa-based model was chosen to test the performance of contextual embedders, while a FastText model will be used to test the performance of non-contextual embedders. Finally, a TF-

IDF embedder was implemented as a baseline embedder.

TF-IDF

TF-IDF has been chosen as a textual data processing approach due to its simplicity and relatively good performance in classifying invoices, as was shown by Johansson[12]. TF-IDF benefits from applying preprocessing steps to decrease the number of unique words, since grouping, for example, inflections of words together through lemmatisation increases the samples in which the grouped word is found. Another common step taken when applying TF-IDF is the removal of stop words. Words such as *that* and *the* are so common in the English language that they cannot realistically be used to improve classification results and are therefore often removed from the data set. To prepare the textual data for categorisation, the following steps have been taken to reduce the vocabulary size.

- punctuation removal
- lemmatisation
- capitalisation removal
- stopword removal

The punctuation removal was performed in combination with the lemmatisation using the *SpaCy* Python library. This library contains text-processing tools for a variety of languages, including Dutch. The *nl_core_news_sm*[1] pipeline was used as it was relatively fast, while still performant. Lemmatisation is chosen over stemming, as preliminary testing showed that the lemmatised text achieved a significantly higher classification score compared to the stemmed text. For the removal of capitalisation, the Python function *str.lower()* was used. Finally, the stopwords were removed by matching the words in the text to a list of Dutch stopwords provided by the *NLTK* Python library. The final vocabulary size of the data set is 1.209.931.

The resulting data set has been split into a training and validation set, using a Stratified K-Fold cross-validation. The stratification applied to cross-validation ensures that the class distribution is the same between the train and the validation set. For each split, a *TFIDFVectorizer()* is fitted to the training data and used to transform the training and validation data into vectors.

FastText

For our experiment, FastText was chosen because it implemented character n-grams to develop its word embeddings. This approach allows the model to collect information from out-of-vocabulary words, as well as words with spelling errors, which should increase the performance of the model over bag-of-words approaches. For the implementation of FastText in our research, a data set consisting of pre-trained word vectors in the Dutch language developed by Grave et al.[10] was used in combination with the FastText Python library. The word embedding model generates word embedding vectors consisting of 300 values. To calculate a sentence vector, each word embedding is divided by its L2 norm[2]. Once these values are calculated, the average value of the normalised vectors with a positive value is calculated to determine the sentence vector that can be used for text classification.

[1]`https://spacy.io/models/nl#nl_core_news_sm`

[2]The L2 norm is the distance of the vector coordinate to the origin

There is no preprocessing of the text required for FastText to perform well, as the character *n* gram approach can use information from infractions and capitalisation to make its predictions.

BERT

The chosen version of RobBERT is RobBERTje[7], a distilled version of RobBERT that is faster to train. Knowledge distillation is the process in which a smaller *student* model is taught by a larger *teacher* model. The distillation allows the new model to be smaller but still have a very similar level of performance. By incorporating the classification head with the RobBERTje model, the classification network can be trained at the same time that the BERT-based model is fine-tuned.

The specific model chosen for our research was:
`DTAI-KULeuven/robbertje-1-gb-shuffled`[3]. This version of RobBERTje was chosen because it performed slightly better compared to the non-shuffled version on the *Dutch Book Review data set*, a classification task, and while it performed marginally worse compared to the large RobBERT model, the training and prediction speeds were significantly higher.

Classifiers

For the TF-IDF embeddings, a Complement Naive-Bayes classifier was trained, as well as two Linear Support Vector classifiers.

Complement Naive-Bayes was chosen as a general baseline as Johansson's work[12] showed that Naive-Bayes can be relatively effective for the classification of invoice lines. For our experiments, however, the choice was made to use the Complement Naive-Bayes approach as opposed to the Multinominal Naive-Bayes approach taken by the related work. Complement Naive-Bayes works by calculating the probabilities that a word is not associated with the classes. The prediction is then made by calculating the class with the lowest probability of not being the correct class. This approach improves the performance of the Naive-Bayes models for imbalanced data sets, as described by Rennie et al. [19]

The Linear Support Vector Classifier was chosen as a classification approach due to its high performance shown in related works[2, 12].
Next to the regular Linear SVC, a hierarchical network of Linear SVC was trained. This network was chosen to investigate the effectiveness of hierarchical classifiers for this specific data set. The approach taken in our research differs from the hierarchical deep learning approach proposed by Munoz et al.[18]. The choice for the current approach is that, in the available data set, the labels in the data set do not show a balanced hierarchy (as visualised in Appendix A). Through the implementation of the hierarchical SVC, the hierarchical classifier can be directly compared to the non-hierarchical SVC. This will show whether the hierarchy present in the data set could be used to increase the classification performance.

We chose not to implement a deep neural network for the TF-IDF embeddings as the vocabulary size of 1.209.931 would result in a very large number of trainable parameters which could not realistically be trained in a reasonable time.

[3]`https://huggingface.co/DTAI-KULeuven/robbertje-1-gb-shuffled`

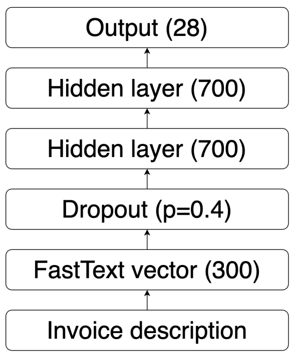


FIGURE 3.8: The architecture of the DNN trained on FastText embeddings.

FastText embeddings are used to train a deep neural network, as it showed good performance in the work of Munoz et al.[18]. FastText was chosen over alternative embedders due to its speed and effectiveness, which was described by Bojanowski et al.[4] The structure of the neural network is described in Figure 3.8. A possible alternative to this approach would be to use FastText directly for classification. However, during the preliminary exploration of the classifiers, this approach did not achieve good results compared to the DNN, while also not allowing for the one-hot encoded contextual data to be included in the model for the second experiment.

For the BERT based approach, a classification head was attached to the pre-trained RobBERTje network. The classification head contains a single layer of neurons with 768 inputs (the number of outputs of the RobBERTje network) and 28 outputs (the number of classes). Then it was trained using a stratified K-fold cross-validation on the training set, after which the results were verified using the test set. The model architecture for this model is described in Figure 3.9.

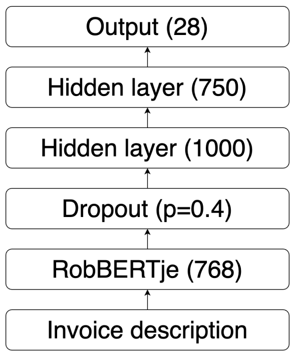


FIGURE 3.9: The architecture of the DNN trained on the RobBERTje embeddings.

Optimisation

To ensure that optimal parameters were selected for each classifier, a grid search was performed. With grid search, an exhaustive search is applied for a group of input parameters, where each combination is tested. The combination of parameters with the highest performance was then used to collect the results from the test data set. For the TF-IDF-based classifiers, the classifiers were fitted using a K-fold cross-validation with $K = 5$ for all combinations of parameters. For the FastText-based DNN and BERT-based classifier, the classifier was trained for a single train-validation split, after which the highest scoring classifier was retrained using K-fold cross-validation with $K = 5$. The choice was made not to cross-validate every combination of hyperparameters because deep learning approaches took significantly longer to train (up to multiple hours), in combination with a larger number of hyperparameters that could be fine-tuned. The full list of the hyperparameters used in this research can be found in Table 3.2. Figure 3.10 contains a diagram that describes the steps used to train the models and find the optimal hyperparameter combination.

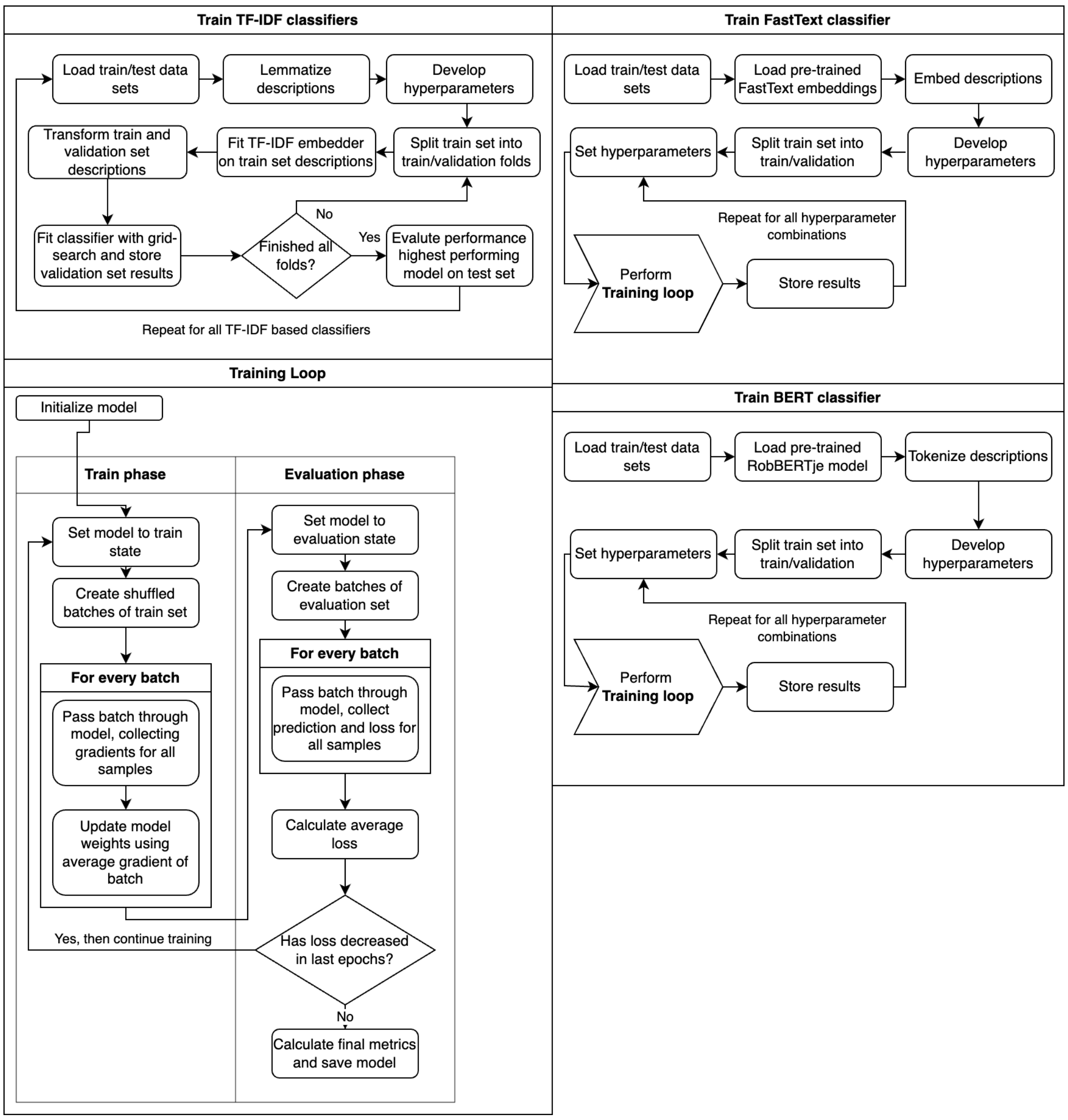

Train TF-IDF classifiers
Load train/test data sets
Lemmatize descriptions
Develop hyperparameters
Split train set into train/validation folds
Fit TF-IDF embedder on train set descriptions
Transform train and validation set descriptions
Fit classifier with grid-search and store validation set results
Finished all folds?
No
Yes
Evalute performance highest performing model on test set
Repeat for all TF-IDF based classifiers
Train FastText classifier
Load train/test data sets
Load pre-trained FastText embeddings
Embed descriptions
Develop hyperparameters
Split train set into train/validation
Set hyperparameters
Repeat for all hyperparameter combinations
Perform Training loop
Store results
Train BERT classifier
Load train/test data sets
Load pre-trained RobBERTje model
Tokenize descriptions
Develop hyperparameters
Split train set into train/validation
Set hyperparameters
Repeat for all hyperparameter combinations
Perform Training loop
Store results
Training Loop
Initialize model
Train phase
Evaluation phase
Set model to train state
Create shuffled batches of train set
For every batch
Pass batch through model, collecting gradients for all samples
Update model weights using average gradient of batch
Set model to evaluation state
Create batches of evaluation set
For every batch
Pass batch through model, collect prediction and loss for all samples
Calculate average loss
Has loss decreased in last epochs?
Yes, then continue training
No
Calculate final metrics and save model

FIGURE 3.10: A diagram describing the training process for the machine learning models

Classifier	**Parameters**
Linear SVM TF-IDF	TF-IDF n-gram range = [(1,1), (1,2),(1,3)] Linear SVM C = [0.3, 0.7, 0.9]
ComplementNB TF-IDF	TF-IDF n-gram range = [(1,1), (1,2),(1,3)] ComplementNB alpha = [0.3, 0.7, 1]
Hierarchical SVM TF-IDF	TF-IDF n-gram range = [(1,1), (1,2),(1,3)] Linear SVM C = [0.3, 0.7, 0.9]

DNN
FastText
	Weight decay = [1e-5, 5e-6, 1e-6]
	Learning rate = [1e-3, 5e-4]
	Gamma = [0.9, 0.95, 0.97]
	Dropout = [0.4, 0.5]

BERT	Learning rate = [1e-4, 5e-5, 1e-5 5e-6]
	Weight Decay = [1e-5, 1e-6]
	Dropout = [0.3, 0.4, 0.5]
	Gamma = [0.85, 0.9, 0.95]

Table 3.2: The parameters used to optimise the trained models

3.2.2 Experiment 2

For the second experiment, contextual information was collected and included in the classifiers. The following contextual features were chosen to be used in the classification process.

- Log_{10} of the price
- Administration SBI-codes
- Administration legal entity type
- Document type
- Invoice line tax rate
- Currency
- Contact company name

The hypothesis under which Log_{10} of the price was chosen as a feature is that the price of the invoice line will give information on the type of product or service purchased. For example, *small office supplies* will generally have a lower cost compared to *property costs.* The prices in the data set range from -1.920.000 to 54.342.428. It is important to note that for the price value, the currency is not directly taken into account. For the samples containing the highest value on the price field, the currency attached to the invoice is the Indonesian Rupiah, where 17.000 Rupias have the same value as one Euro at the time of writing. To decrease the difference between the largest and smallest values, the Log_{10} of the value is used. Through rescaling, the distance between the largest and smallest values will decrease significantly and also make it so that the value does not have significantly higher values compared to the other contextual variables. Since the price can be negative, the Log_{10} is taken using the absolute value of the price, which is used as the input of the model in combination with a boolean value indicating whether the original value is positive or negative.

An Standaard BedrijfsIndeling/Standard Business Categories (SBI) code is used to describe the activities of a company, which are provided by the Dutch Chamber of Commerce. The codes are built in a tree structure and can be either four or five digits long. An example of the tree structure is visualised in Table 3.3. We think that these codes can be relevant for the classification, as the field a company operates in should influence the types of purchases made by a company. The feature is similar to the industry in which the company is based, which Bergdorf has successfully used as a feature[2] in their research. To reduce the dimensionality of the feature in the data set, the codes have been shortened to the first two characters, after which they have been binarised.

Description	**SBI code**
Retail (not in cars)	47
Supermarkets and warehouses	47.1
Supermarkets	47.1.1
Warehouses (non-food)	47.1.9
Specialised stores in food and beverages	47.2
Stores for potatoes, vegetables and fruits	47.2.1

Table 3.3: An example of the tree structure of SBI codes (the punctuation is added for clarity)

The *legal entity type* of an administration describes the type of company that is being run, which can vary from *sole proprietor* (*eenmanszaak* in Dutch) to privately traded companies (*bv* in Dutch). The *legal entity type* was included because it might improve the classification performance under the hypothesis that different legal entity types have a different company structure and therefore could make different purchases and use different ledger codes. Furthermore, some ledger codes do not apply to sole proprietors, but do apply to other business types (as of writing, there are 631 RCSFI codes applicable for sole proprietors, while there are 791 codes applicable for privately traded companies[4]). Initial data investigation showed that some labels were significantly less common for specific legal entity types. For example, the code for *Costs of outsourced work* is ±10% of the total data set. However, for *Professional or public partnership* (*Maatschappen* in Dutch) entities, this is only ±6%.

The invoice type can be either a purchase invoice or a (scanned) receipt that is entered into the website. This feature could be relevant because, for example, office supplies might often be bought in a physical store, after which a receipt is scanned. Larger purchases might be made online more often, where they are processed through an invoice. In Figure 3.11, the Sankey diagram is used to show the relation between the invoice types and the ledger codes. The graph shows that, for example, the RCSFI code *WBedVkkRep*, which describes *Representation costs*, is significantly more common for receipts compared to invoices. An explanation for this can be found by looking into what the *representation costs* ledger code is used for, which is commonly used for business lunches and dinners, where invoices are not commonly used.

The invoice line tax rate describes how a product is taxed. There are several options ranging from 21% to 0% and other options such as shifting taxes. The feature can give information on the type of product, as some products are taxed differently than others. The tax rate was successfully used as a feature by Munoz et al.[18] in their implementation of ledger code classification.

[4]https://boekhoudplaza.nl/cmm/rgs/rgs_dashboard.php

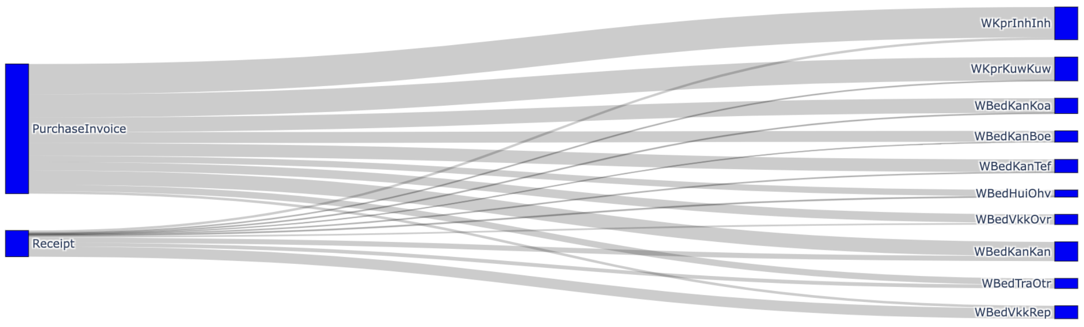


FIGURE 3.11: A Sankey diagram showing the 2 document types linked to the 10 most common RCSFI codes in the data set.

The invoice currency was chosen as a context feature as it could indicate information on the origin of the product. The currency was added as a feature because an investigation of its values showed that some ledger codes were more common with specific currencies. For example, the ledger code for *automatisation* has 129.471 instances where the currency was the Euro (out of a total of 2.040.088 instances), which is around ±6%. For the US dollar, this ledger code occurred 16.873 times (out of a total of 33.987), occurring a total of ±50%. The difference in the frequency of the ledger codes when the samples are grouped by currency shows that using the currency as a feature could improve the prediction performance of the classifiers.

The text field containing the company name is included as contextual data under the hypothesis that company names can contain information that could be useful to determine the type of ledger codes used for that contact. This field is filled in by the administration, so there could be some ambiguity in this field. However, some company names may be often associated with specific ledger codes. For example, *KPN B.V.* is a very large Dutch telecom provider and occurs a total of 168.936 times as the contact in the rows of the data set. 126.024 (±75%) of these samples have been assigned the *Telephone costs* ledger code, showing that the company name can be a significant indication of the correct ledger code. Furthermore, company names can also contain information that suggests the field in which the contact works, which can be interpreted by models to determine the business field in which the contact operates, and which can indicate the ledger code, similar to the SBI code. For example, *Parkmobile Benelux B.V.* is a common contact name in the data set. The company provides, as the name suggests, an app that can be used to pay for parking spots. The contact name was chosen over the contact SBI code because the SBI codes are collected through a system that requires the Chamber of Commerce (CoC) code of a company. However, in contrast to the CoC code for the company using the bookkeeping software, this field is not filled in for a large number of their contacts (over 30%). Requiring the contact CoC field for the data samples would significantly reduce the number of trainable samples and limit the practical usefulness of the model, since the number is not a mandatory field and would not be filled in for contacts located abroad.

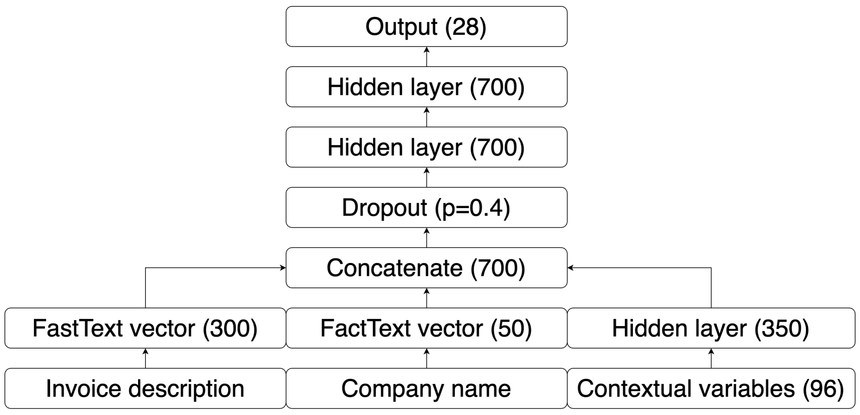


FIGURE 3.12: The model architecture of the DNN using FastText in Experiment 2

Classifiers

For classifiers based on TF-IDF, the company name was embedded with a separately trained TF-IDF vectorizer, after which all variables are concatenated and used by the classifiers.

The FastText-based DNN was redesigned for the second experiment. The company name was transformed using a FastText embedder that had its output dimensionality reduced to 50. This choice was made because the vocabulary size was significantly smaller (375.470 for company names and 1.473.822 for descriptions). The other variables were encoded and passed through one ReLU activated layer that was used to increase the dimensionality of the categorical variables to 350 (the same as the textual features) before being concatenated with FastText embeddings. The rest of the network was identical to that of Experiment 1, and the model architecture is described in Figure 3.12.

For the BERT based approach, the company description was appended to the description, separating the texts using the $[SEP]$ token. The other contextual features were, similarly to the DNN approach, passed through a single ReLU-activated layer before being concatenated to the embeddings, with a classification head that consists of 2 ReLU-activated hidden layers and an output layer. The model architecture is described in Figure 3.13.

Optimization

The hyperparameter tuning steps used in Experiment 1 (described in Table 3.2) were repeated in Experiment 2. To train the classifiers, the approach used in the first experiment (Figure 3.14) is updated to include the processing of contextual information and the name of the company name that sent the invoice. The changes are visualised in Figure 3.14.

Ablation study

To analyse the effect of the different contextual features included in the second experiment, an ablation study is performed to determine the effect of the exclusion of individual

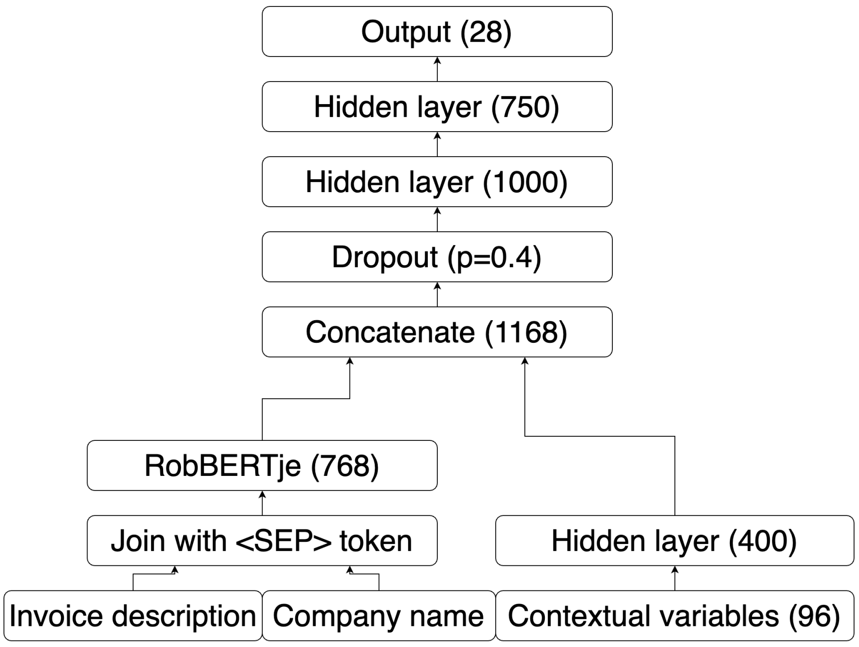


FIGURE 3.13: The model architecture of the DNN using RobBERTje embeddings in Experiment 2

contextual features. In Figure 3.15, the steps taken to perform the ablation study are described. For the ablation study, we choose the model that achieves the highest Macro F1 score and repeat the training where in each training phase, one contextual variable is excluded. This process should reveal the importance of individual contextual features.

3.2.3 Experiment 3

In the third experiment, we investigate the optimal approach for a practical implementation of a classification model. We investigate the prediction speed of the best classifiers from the previous two experiments. The prediction speed is compared with the classification performance, after which a final setup is selected for the practical implementation of the optimal classifier, which is used to develop a prototype. The prediction speed of the classifiers is determined by calculating the predictions of the first 1000 samples of the test set, splitting the total time into preparation and prediction times. The classification process is repeated 20 times to ensure that the results are valid. The predictions are performed on the CPU of an Apple MacBook with an Apple M1 Pro processor and 16GB of RAM. Since models based on a DNN can greatly benefit from the use of a GPU, the training for these models is also applied on the MacBook GPU, as well as a virtual machine using an NVIDIA A10G processor and 4 cores of an AMD EPYC 7R32 processor. It is important to note that for these experiments, the preprocessing of the data is still performed on the CPU, but the inference is performed on a GPU.

3.2.4 Prototype development

As part of this research, we develop a prototype that implements the best-performing model into the existing application to validate the results of the research. To implement the model, we create a web application that hosts an Application Programming Interface

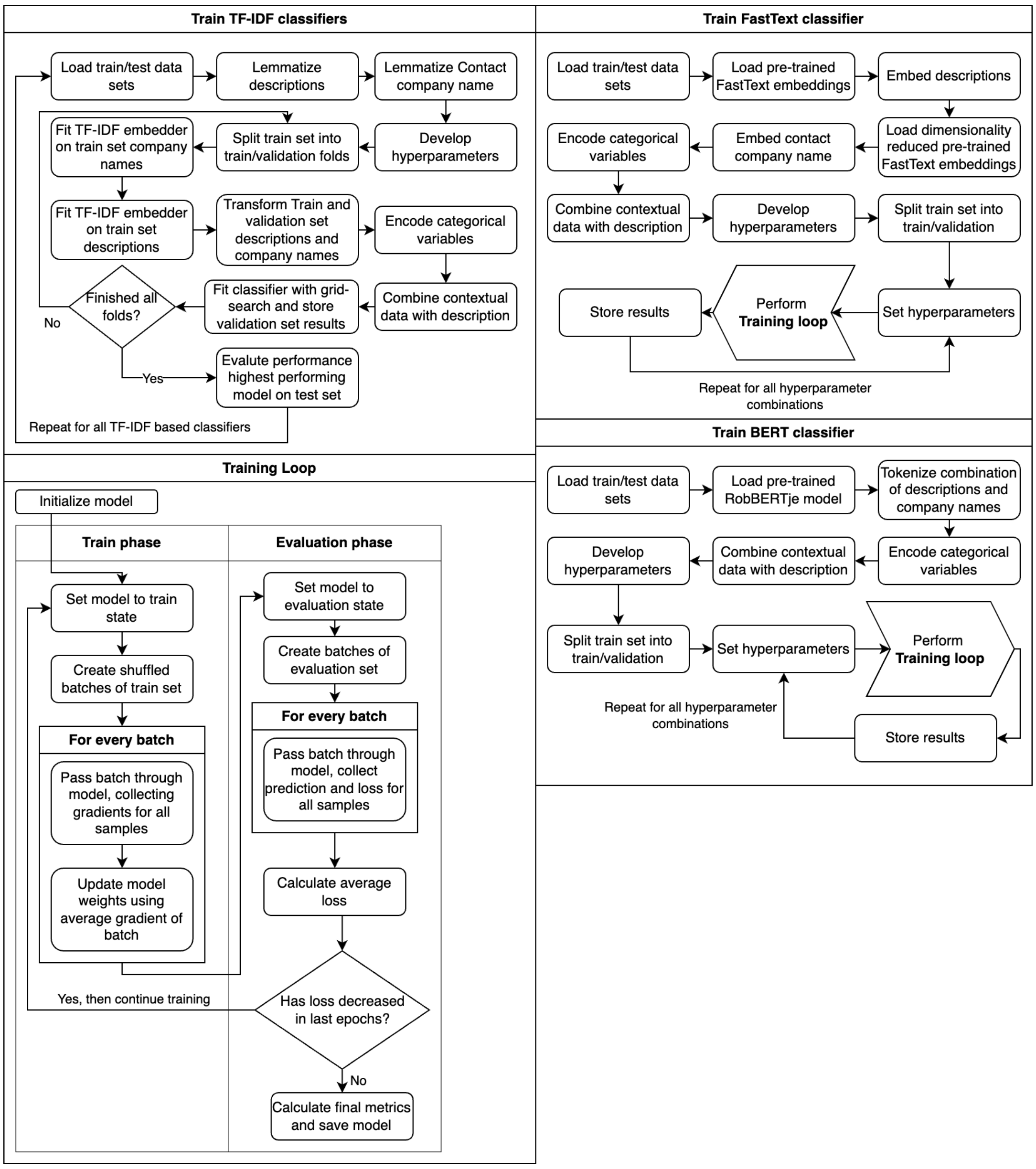

Train TF-IDF classifiers
Load train/test data sets
Lemmatize descriptions
Lemmatize Contact company name
Fit TF-IDF embedder on train set company names
Split train set into train/validation folds
Develop hyperparameters
Fit TF-IDF embedder on train set descriptions
Transform Train and validation set descriptions and company names
Encode categorical variables
Finished all folds?
No
Fit classifier with grid-search and store validation set results
Combine contextual data with description
Yes
Evalute performance highest performing model on test set
Repeat for all TF-IDF based classifiers
Train FastText classifier
Load train/test data sets
Load pre-trained FastText embeddings
Embed descriptions
Encode categorical variables
Embed contact company name
Load dimensionality reduced pre-trained FastText embeddings
Combine contextual data with description
Develop hyperparameters
Split train set into train/validation
Store results
Perform Training loop
Set hyperparameters
Repeat for all hyperparameter combinations
Train BERT classifier
Load train/test data sets
Load pre-trained RobBERTje model
Tokenize combination of descriptions and company names
Develop hyperparameters
Combine contextual data with description
Encode categorical variables
Split train set into train/validation
Set hyperparameters
Perform Training loop
Repeat for all hyperparameter combinations
Store results
Training Loop
Initialize model
Train phase
Evaluation phase
Set model to train state
Create shuffled batches of train set
For every batch
Pass batch through model, collecting gradients for all samples
Update model weights using average gradient of batch
Set model to evaluation state
Create batches of evaluation set
For every batch
Pass batch through model, collect prediction and loss for all samples
Calculate average loss
Yes, then continue training
Has loss decreased in last epochs?
No
Calculate final metrics and save model

FIGURE 3.14: A diagram describing the process of training the machine learning classifiers for the second experiment

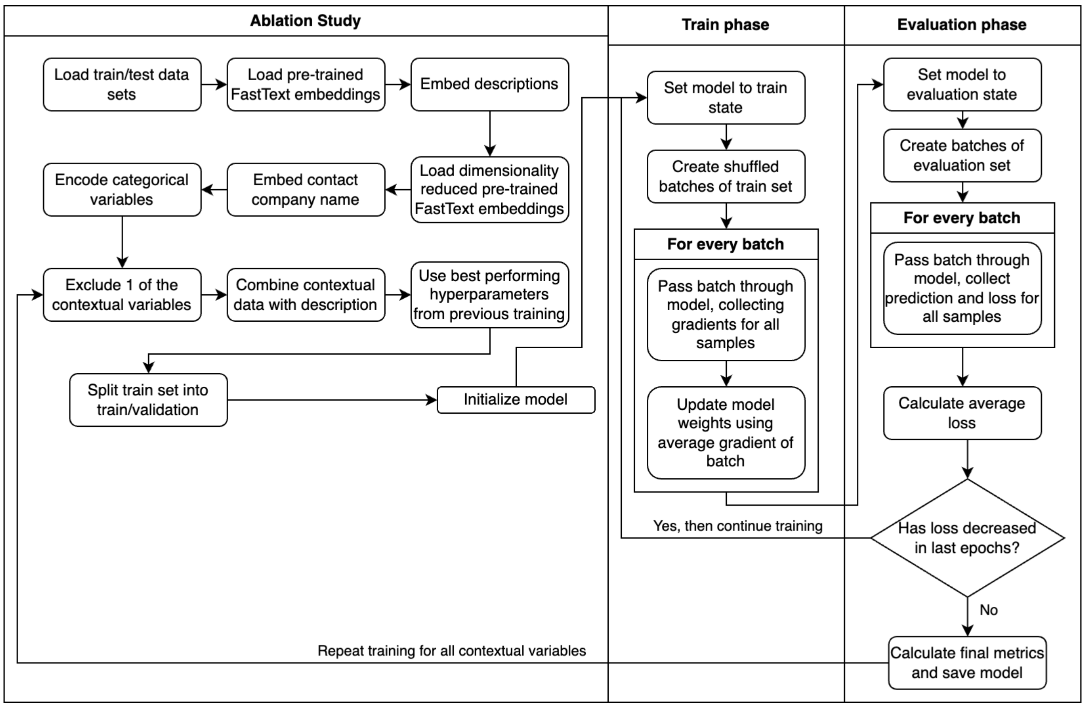

Ablation Study
Train phase
Evaluation phase
Load train/test data sets
Load pre-trained FastText embeddings
Embed descriptions
Load dimensionality reduced pre-trained FastText embeddings
Embed contact company name
Encode categorical variables
Exclude 1 of the contextual variables
Combine contextual data with description
Use best performing hyperparameters from previous training
Split train set into train/validation
Initialize model
Set model to train state
Create shuffled batches of train set
For every batch
Pass batch through model, collecting gradients for all samples
Update model weights using average gradient of batch
Yes, then continue training
Set model to evaluation state
Create batches of evaluation set
For every batch
Pass batch through model, collect prediction and loss for all samples
Calculate average loss
Has loss decreased in last epochs?
No
Calculate final metrics and save model
Repeat training for all contextual variables

FIGURE 3.15: The steps to perform the ablation study

(API) that can be used to request a prediction. This application can then be used by the existing bookkeeping system to show the user the predictions. A more detailed description of the implementation of the prototype is described in Section 5.2.

To evaluate the prototype, we use the *Network* tab in Google Chrome's[5] DevTools to measure the time it takes for the request to the prediction API to be completed. By measuring exact timings in a browser, we include any overhead caused by HTTP requests and the additional processing required to show the prediction to the user in the time it takes to predict the category. This approach will give a realistic prediction speed in a practical setting compared to the time it takes to purely predict the category, as is done in Experiment 3.

Furthermore, we evaluate the prototype by asking developers of the bookkeeping software to test the prototype. For this, we create an example invoice (see Appendix D) that the developers will fill in on the website. The developers can also come up with examples themselves to allow them to test the accuracy of the predictions. After the developers are finished with their testing, they are asked the following questions:

- What was your overall experience using the prototype?
- What is your opinion on the responsiveness of the predictions?
- How do you perceive the correctness of the predictions?
- What could be improved about this prototype?
- Do you believe that this concept can be used to improve the user experience of the bookkeeping software?

[5]https://www.google.com/chrome/

The answers to these questions are used to improve the prototype and as future work to further develop the prototype so that it can be used by the users of the bookkeeping software.

Chapter 4

Results and discussion

4.1 Experiment 1

In this section, we describe the results of the first experiment, which are used to answer the first sub-question of this research described in Section 1.2. Through hyperparameter tuning, the combinations described in Table 4.1 were found to be the optimal parameters for the individual models.

Classifier	**Optimal Hyperparameters**
Naive-Bayes	{Alpha: 0.5, N_gram_range: (1,3)}
Linear SVC	{C=0.9, N_gram_range: (1,2)}
Hierarchical SVC	{C=0.9, N_gram_range: (1,2)}
FastText DNN	{Weight_decay: 1e-05, Learning_rate: 0.001, Gamma: 0.95, Dropout: 0.4}
BERT DNN	{Weight_decay: 1e-06, Learning_rate: 0.0001, Gamma: 0.85, Dropout: 0.4}

Table 4.1: The optimal hyperparameters for each of the classifiers trained for this experiment. The complete results for all the parameter combinations are described in Appendix C.

In Table 4.2, the results of the classifiers with the optimal hyperparameters on the test set are described. The table shows that BERT achieves the highest Macro F1 score, with the Linear SVC achieving a similar score. The full results, including F1 scores on the validation set for each fold, are described in Appendix B.1.

Classifier	**Embeddings**	**Macro F1**	**Accuracy**	**Top 3 acc**	**Top 5 acc**
Complement NB	TF-IDF	0.5582	0.6139	0.7888	0.8432
Linear SVC	TF-IDF	0.5920	0.6458	0.8230	0.8800
Hierarchical SVC	TF-IDF	0.5322	0.6020	-	-
DNN	FastText	0.5604	0.6221	0.8278	**0.9001**
DNN	BERT	**0.6020**	**0.6545**	**0.8296**	0.8961

Table 4.2: The test results for the classifiers. The Top-K acc values for the Hierarchical classifier could not be generated as the HiClass library does not support the calculation of probabilities.

Comparing the models based on the TF-IDF embeddings, the non-hierarchical linear SVC significantly outperforms the other classifiers in all of the metrics chosen. Important to note here is that, due to limitations of the HiClass library used for the implementation of the hierarchical classifier, it was not possible to generate top K accuracies for this model. However, comparing the Macro F1 and accuracy scores of this model to the non-linear model, one can reasonably conclude that for this data set, the hierarchy in the labels does not provide sufficient information to increase the performance of classifiers.

For models based on a DNN, the model based on BERT embeddings achieved the highest scores on every performance metric except for the Top-K accuracy, where the FastText embedded model slightly outperforms the BERT embedded model.

4.2 Experiment 2

In the second experiment, we examine whether the performance of the models trained in Experiment 1 can be improved by including contextual data from the invoice. For this experiment, the hyperparameter tuning is repeated on the new models. The optimal hyperparameters are identical to the hyperparameters found in experiment 1 (described in Table 4.1). The complete results, including F1 scores on the validation set for each fold and the performance increases compared to the first experiment, are described in Appendix B.2. The results of the validation set for all combinations of hyperparameters are described in Appendix C. With the optimal parameters, the results of the test set are calculated. The results are summarised in Table 4.3.

Classifier	**Embeddings**	**Macro F1**	**Accuracy**	**Top 3 acc**	**Top 5 acc**
Complement NB	TF-IDF	0.6382	0.7045	0.8603	0.9158
Linear SVC	TF-IDF	0.6743	0.7089	0.8572	0.9080
Hierarchical SVC	TF-IDF	0.5987	0.6549	-	-
DNN	FastText	**0.7092**	**0.7594**	**0.9065**	**0.9490**
DNN	BERT	0.6808	0.7385	0.8920	0.9404

Table 4.3: The test results for the classifiers using contextual data. The Top-K acc values for the Hierarchical classifier could not be generated as the HiClass library does not support the calculation of probabilities.

In this experiment, the FastText-based model outperforms the other models in every calculated performance metric and is the only model that achieved a Macro F1 score greater than 70%. Although Figure 4.1 shows that all classifiers trained in the first experiment benefit from the inclusion of contextual data, some models improved significantly more compared to others. For example, the Macro F1 score for the hierarchical support vector classifier increased by ± 0.065, while the same score increased by ± 0.148 for the DNN based on FastText embeddings.

4.2.1 Ablation study

The second experiment also includes an ablation study to determine the performance implications of individual contextual features. We chose the DNN trained on FastText embeddings using contextual data for this study, as it achieved the highest performance scores in the earlier part of the experiment. The results for this are described in Table 4.4. The results of the ablation study show that the exclusion of the features *administration SBI codes* and *contact company name* has the largest impact on classification performance.

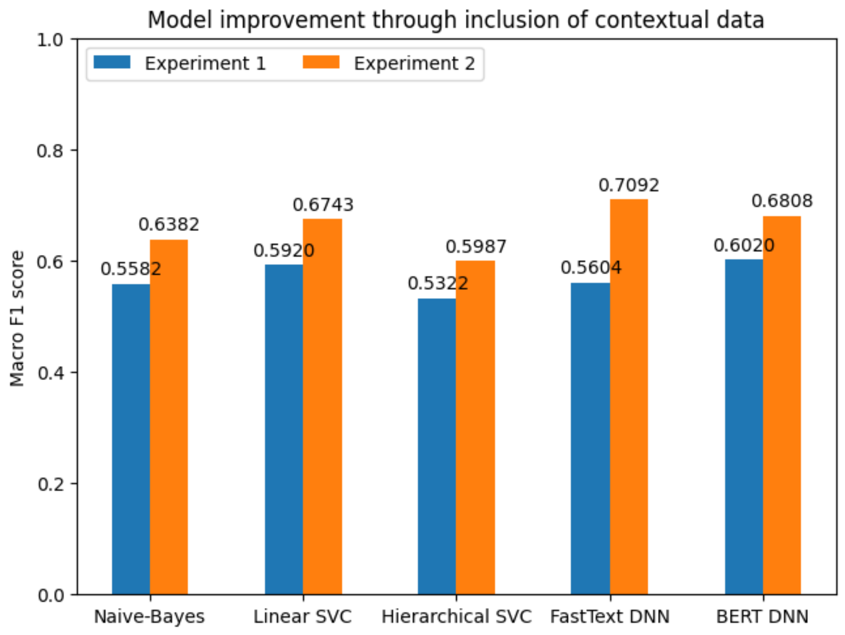


FIGURE 4.1: A bar chart showing the improvement in classification performance through the inclusion of contextual data

Although the removal of other individual contextual features does not show a significant impact on performance, the model trained using only *contact company name* and *administration SBI codes* as contextual data achieved a macro F1 score of ±0.68, showing some performance degradation.

Excluded Feature	**Macro F1**	**Acc**	**Top 3 Acc**	**Top 5 Acc**
Invoice line log price	0.7034	0.7548	0.9052	0.9484
Administration SBI codes	0.6377	0.6975	0.8753	0.9318
Administration legal entity type	0.6940	0.7475	0.9011	0.9450
Document type	0.7070	0.7564	0.9061	0.9488
Currency	0.7045	0.7550	0.9061	0.9487
Invoice line tax rate	0.7047	0.7547	0.9063	0.9492
Contact company name	0.6702	0.7230	0.8878	0.9369
Baseline	0.7092	0.7594	0.9065	0.9490

TABLE 4.4: performance of the DNN using FastText embeddings, performing an ablation study excluding a contextual feature.

4.3 Experiment 3

In the third experiment, we examine the potential practical value of the developed model by examining the prediction speed. The prediction speed of a model is relevant for this research since, for a practical implementation, the model needs to be able to give customers a fast prediction so as not to get in the way of the user. The results, as described in Table 4.5, show that the fastest models to create predictions are the FastText-based DNN models. Although the prediction speeds of the Linear SVC models are very fast, the preprocessing step takes very long because of the lemmatisation of the sentences. The BERT-based classification models had the lowest prediction speed when calculating their predictions on a CPU.

Classifier	Processing	Prediction	Total	Std
Complement Naive-Bayes non-contextual	2.7044	0.0601	2.7644	0.1486
Complement Naive-Bayes contextual	2.6608	0.0643	2.7252	0.0848
Linear SVC non-contextual	2.8007	0.0024	2.8031	0.1887
Linear SVC contextual	2.6200	0.0009	2.6209	0.0779
Hierarchical SVC non-contextual	2.6715	0.0325	2.6764	0.0697
Hierarchical SVC contextual	2.6110	0.0051	2.6161	0.0299
FastText DNN non-contextual	0.0131	0.0029	0.0160	0.0011
FastText DNN contextual	0.0240	0.0078	0.0318	0.0014
BERT non-contextual	0.4851	3.2198	3.7047	0.0791
BERT contextual	0.5635	11.8785	12.442	0.1035

Table 4.5: Inference time to calculate 1000 predictions in seconds, averaged over 20 runs.

As described in Section 3.2.3, the prediction speeds for the models using a DNN are also measured when calculated on a GPU. Using the MacBook GPU, only the BERT models show a significant increase in prediction speeds. Running the models on an Nvidia GPU results in a larger improvement, but this effect was somewhat limited by a decrease in processing speed caused by the slower CPU cores available in the virtual machine. The results of the classification speed on hardware with a dedicated GPU show only a significant performance increase when applying large models such as BERT, and for smaller DNN models, GPU inference does not significantly impact the classification speed. A possible explanation for this would be that the model inputs need to be moved from the CPU memory to the memory of the dedicated graphics card. This would explain the improvement seen on the large models, as the time it takes to move the inputs is negligible compared to the increase in processing power.

Classifier	CPU	GPU	Processing	Prediction	Total	Std
FastText DNN non-contextual	Apple M1 Pro (10 cores)	Apple M1 Pro (14 cores)	0.0157	0.0004	0.0161	0.0030
FastText DNN contextual	Apple M1 Pro (10 cores)	Apple M1 Pro (14 cores)	0.0268	0.0008	0.0275	0.0009
BERT non-contextual	Apple M1 Pro (10 cores)	Apple M1 Pro (14 cores)	0.4757	3.1907	3.6664	0.0759
BERT contextual	Apple M1 Pro (10 cores)	Apple M1 Pro (14 cores)	0.6140	3.1306	3.7446	0.0856
FastText DNN non-contextual	AMD EPYC 7R32 (4 cores)	NVIDIA A10G (9216 cores)	0.0214	0.0003	0.0218	0.0021
FastText DNN contextual	AMD EPYC 7R32 (4 cores)	NVIDIA A10G (9216 cores)	0.0396	0.0079	0.0475	0.0475
BERT non-contextual	AMD EPYC 7R32 (4 cores)	NVIDIA A10G (9216 cores)	0.9328	0.6770	1.6098	0.271
BERT contextual	AMD EPYC 7R32 (4 cores)	NVIDIA A10G (9216 cores)	1.1364	0.6194	1.7558	0.0068

Table 4.6: GPU inference time to calculate 1000 predictions in seconds, averaged over 20 runs.

For a practical implementation, the requirement for a GPU to achieve reasonable prediction speeds is a significant downside, as servers containing a GPU are more expensive. This makes models that can predict fast on a CPU better suited from a practical standpoint. This makes the FastText-based DNN from the second experiment the most suitable classification model to use for further analysis, as the model achieves both the highest scores for our performance metrics and a very high prediction speed on a CPU.

4.4 Error analysis

In this section, we further analyse the DNN trained on FastText embeddings and contextual data. This model was chosen for analysis as this model achieved both a very high prediction speed and good classification scores. The confusion matrix (see Figure 4.2) shows that the ledger codes *BMvaBeiVvp* "Acquisition of inventory") and *BMvaObeVvp* ("Acquisition of other fixed assets") are often confused by the model. Examination of the misclassifications shows that the misclassified samples are very similar. For example, the description of a sample labelled as *BMvaBeiVvp* is *Apple MacBook Pro 16"Touch Bar*, while another sample labelled as *BMvaObeVvp* has *Macbook Pro touch bar 16 inch* as the description. This does not mean that the samples are mislabelled as the correct label depends on the practical use of the product purchased. Similarly, samples with labels *WBedAutOak* ("Other car costs") and *WbedAutRoa* ("Car repair and maintenance") are also commonly misclassified.

Manual inspection of the predictions made on the test set shows that there are also a number of misclassifications in which the provided description does not indicate what the ledger code could be. For example, the description *"Invoice 1847"* does not indicate the product described in this invoice line.

The confusion matrix also shows a vertical coloured line for classes *WKprInhInh* ("Purchase of trade goods"), *WKprKuwKuw* ("Outsourced work costs"), and *WBedKanKan* ("Office supply purchases"). These classes are commonly predicted as the label and this can be explained by looking at the sample distribution as these are the three most commonly used classes in the dataset. The model has adapted to this, and this has caused the probability of these classes being predicted to increase.

4.5 Summary

In our experiments, we compare the performance of the TF-IDF, FastText, and BERT embedders on the classification of Dutch invoice data. In our first experiment, we examine the performance of different classification models using TF-IDF, FastText, and RobBERTje to embed textual invoice descriptions to determine their ledger code. The results of this experiment show that a DNN based on BERT embeddings achieves the highest Macro F1 score when predicting only using textual descriptions.

In our second experiment, we incorporated contextual data into the classifiers to examine whether this information could be used to improve classification performance. The results show a major improvement in the Macro F1 score when contextual data is included. The model with the highest Macro F1 score for this experiment is the DNN based on FastText embeddings, achieving a Macro F1 score of ±71%. This classifier was then used in an ablation study to investigate which contextual feature has the largest impact

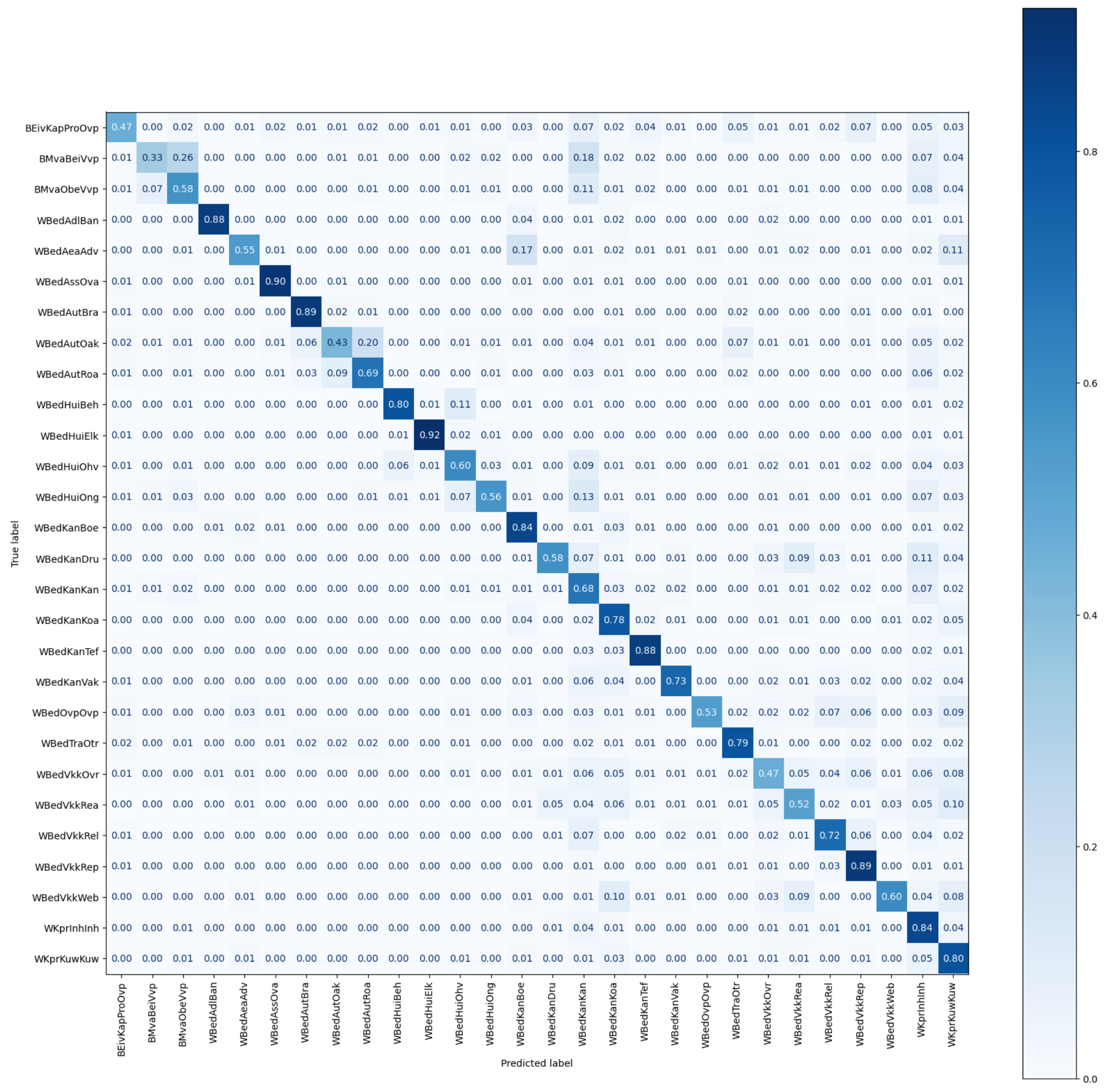


FIGURE 4.2: A confusion matrix showing the amount of misclassifications of the model. The values are normalised on the true label axis.

on the Macro F1 score. The ablation study showed that *Administration SBI codes* and *Contact company name* had the largest impact on the classification performance.

In our third experiment, we investigated the prediction speed of the previously developed classifiers. The results of this experiment show that the DNN based on FastText embeddings was the fastest to make its predictions. The results also show that the lemmatisation applied for the TF-IDF approaches requires significant processing time and that the BERT-based DNN takes the longest to calculate its predictions. The classifiers that incorporated a DNN are also tested on a GPU, with the BERT based model showing a significant improvement in prediction speed, whereas the FastText model did not.

Finally, we applied an error analysis to the results of the best-performing model on the test set. The confusion matrix shows that some hierarchically close classes are often confused with each other and that the classes that are the most common in the dataset are also more commonly predicted compared to the less common classes. Furthermore, analysis of misclassified samples also shows many samples in which the textual description does not adequately describe the product or service purchased, making the classification significantly more difficult.

Chapter 5

Prototype development

To show the practical value of this experiment, this research includes the development of a prototype that implements a classification model to recommend ledger categories to users of the Moneybird bookkeeping software. This chapter is split into two parts; the first part explains the current system and the requirements of the prototype, while the second section explains the implemented architecture and the resulting prototype.

5.1 Prototype requirements

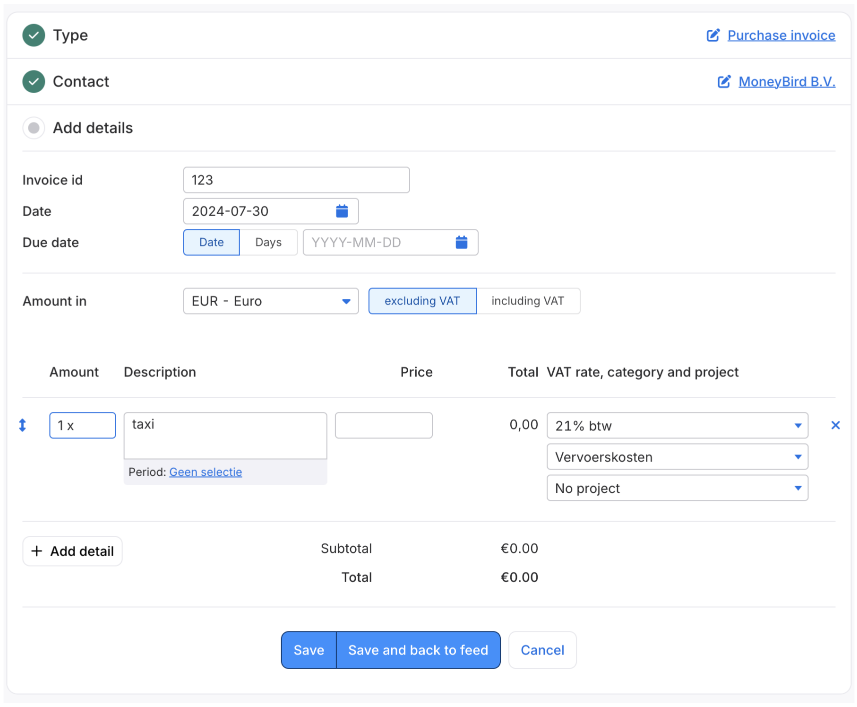
When a Moneybird user creates a purchase invoice, the first step is to create or select the other party in the transaction (called the contact) from a list containing all their contacts. Once the contact has been selected, the user is shown the form described in Figure 5.1. In this form, the user enters information required to complete the invoice, such as description, currency, price, and tax rate. These fields need to be used by the prediction model. The category field (this is *"Vervoerskosten"* in the example, which can be translated to *"Transport costs"*) is the field where the prediction must be implemented.

The goal of the prototype is to show that it is possible to use the machine learning models developed to assist users in choosing the correct ledger codes. Because of this, the model needs to be able to predict the ledger codes for an invoice line based on the users' input in the fields, updating its predictions when the fields change.

Another consideration is that, for the ledger accounts, each user has a subset of the ledger codes enabled for their administration. Each of these ledger accounts is linked to a taxonomy item which contains the RCSFI code that the model can predict. Because of this, the prototype has to link the predicted ledger code to a taxonomy item and then determine whether the user has enabled a ledger account using this item. The relationships between the different database tables are described in Figure 5.2. If no ledger account exists that is linked to the predicted taxonomy item, the prototype should show the user a suggestion to create a new ledger account with the predicted ledger code.

5.2 Prototype architecture

For the implementation of the prototype, we decided together with Moneybird to implement a simple recommendation system that recommends the three most likely ledger codes


Type — Purchase invoice

Contact — MoneyBird B.V.

Add details

Invoice id	123
Date	2024-07-30
Due date	Date Days YYYY-MM-DD
Amount in	EUR - Euro [✓ excluding VAT] including VAT

Amount	Description	Price	Total	VAT rate, category and project
1 x	taxi Period: Geen selectie		0,00	21% btw Vervoerskosten No project

+ Add detail

Subtotal	€0.00
Total	€0.00

Save | Save and back to feed | Cancel

FIGURE 5.1: A view of the form used to create purchase invoices in Moneybird

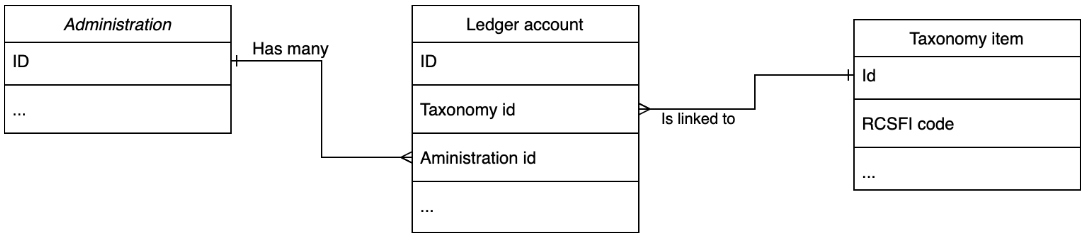


FIGURE 5.2: An Entity Relationship Diagram diagram describing relevant fields in the Moneybird database structure. Each administration has many ledger accounts, each of which links to a single Taxonomy item, which contains the RCSFI code.

according to the model. We chose to implement the FastText-based DNN that incorporates contextual data, as it achieves a top-3 accuracy score of 90% while also achieving a high prediction speed. This makes this model well suited for a recommendation system in which the user is shown several options to choose from, since the probability that the correct label is one of the options is very high. The implementation of the prototype can be split into three areas, the website front-end and back-end implementation, as well as the creation of the prediction service. An overview showing the request structure of the prototype is described in Figure 5.3. Moneybird uses *Rails*[1], a *Ruby*[2] based web framework for their website. In the rest of this section, we explain the structure of the different parts of the prototype implementation.

[1]https://rubyonrails.org/
[2]https://www.ruby-lang.org/en/

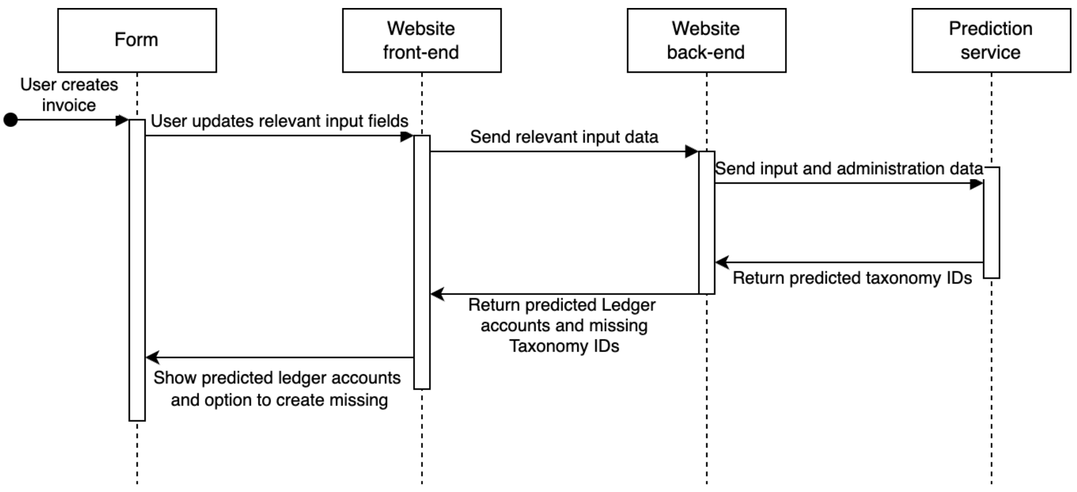

Form
Website front-end
Website back-end
Prediction service
User creates invoice
User updates relevant input fields
Send relevant input data
Send input and administration data
Return predicted taxonomy IDs
Return predicted Ledger accounts and missing Taxonomy IDs
Show predicted ledger accounts and option to create missing

FIGURE 5.3: A sequence diagram describing the flow of data to show the user a prediction

5.2.1 Prediction service

The prediction service is developed as a simple Flask[3] API. The purpose of this API is to provide an endpoint where requests can be made containing the (con)textual information required by the model to make its prediction. Once the endpoint receives data, it performs the pre-processing steps such as text embedding using FastText and binarising the categorical variables. This endpoint will return the ID of the taxonomy item in the database that corresponds to the predicted RCSFI code.

5.2.2 Back-end implementation

In the back-end implementation, a Rails controller is created that collects all the features required to make a prediction and sends the request to the prediction service.

Once the controller receives the prediction, the controller searches whether the administration has the three predicted ledger codes enabled, returning either the ledger account ID and name if it exists or the name and ID of the taxonomy item if missing.

5.2.3 Front-end implementation

To create the predictions in the front-end, event listeners are created in the fields relevant to the prediction. Fields such as the contact are already stored in the back-end at this step of the form and should therefore not be collected by the front-end. For textual fields such as price and description, the front-end waits until the user has stopped typing for 0.5 seconds before requesting a prediction. For the drop-down fields, the prediction is requested when the value is changed.

To calculate the prediction, the selected currency, textual description, price, and tax rate are extracted from the form. These values are sent to the back-end where they are used to calculate a prediction.

[3]https://flask.palletsprojects.com/en/3.0.x/

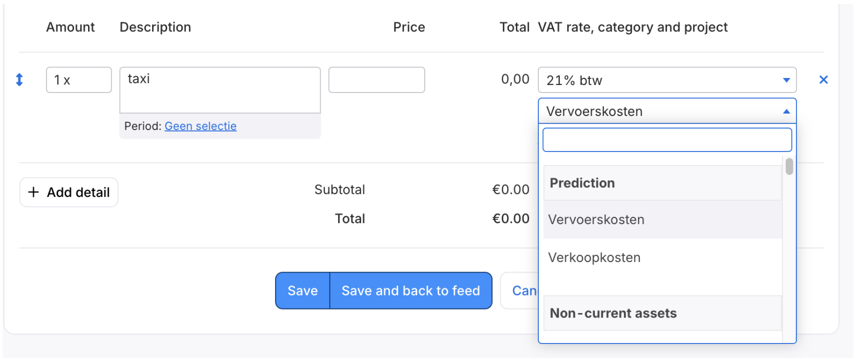


FIGURE 5.4: The drop-down after the prediction has been made, showing a section of predicted codes

Once the prediction is returned, the drop-down is extended to include a section containing the predicted ledger accounts (see Figure 5.4). If one of the predicted ledger codes is not enabled for the user, a text box is shown containing links to a page where the ledger code can be enabled (see Figure 5.5). Finally, if the user has not manually chosen a ledger code before the prediction is made, the most probable ledger code enabled for the administration is automatically selected for the user.

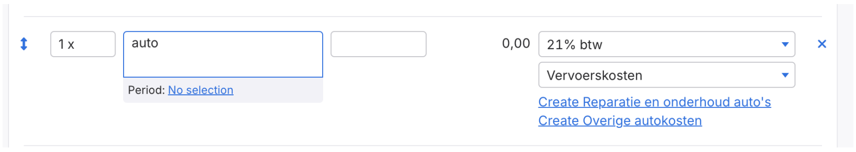

1 x
auto
Period: No selection
0,00
21% btw
Vervoerskosten
Create Reparatie en onderhoud auto's
Create Overige autokosten

FIGURE 5.5: An example of the button to create a non-existing category for a predicted class

5.3 Prototype results

Although the prototype has not yet been used by Moneybird customers, a lot of useful information can be obtained from the implementation. Using the *Network* tab in Google Chrome, we can see that the request that returns the ledger accounts takes a total time that ranges from 100 to 200 milliseconds.

Two experienced Moneybird developers, one of which had experience with the structure of the RCSFI taxonomy, tested the prototype, giving feedback on the functionality and value of the concept for their product. They were given an example invoice (Figure D) to fill in on the website, as well as the option to test the predictions of the examples they came up with themselves. The testers were asked whether they believed the predictions were correct and their opinion on the design of the implementation. The general feedback on the functionalities was positive. The developers said that the responsiveness and quality of the predictions were generally good and that they could see the value of the prototype to the user experience. The developers also noted some points of consideration, the most important point being that the design of the current implementation should be improved. An example of an improvement would be to let the user select a non-existing ledger account and then create it when the invoice is saved. This would remove the step of manually creating the ledger account for the user, reducing the number of clicks that a user would have to make. Another improvement that would greatly increase the practical value of the concept would be to implement this system into their document scanner. In the document scanner, users can directly upload their invoices, after which the invoice data is automatically extracted from the invoice. Currently, users still have to manually select the category for the scans, but this can be improved by applying the prediction to the data extracted from the scan.

Finally, the developers mentioned the importance of the correctness of the data. Recently, some work has been performed that has resulted in a larger number of ledger codes for a company's possessions becoming available to users. The data used for machine learning mostly consists of older data, which results in the predictions not reflecting what is currently considered the correct ledger code by Moneybird.

Chapter 6

Conclusion

Using the results described in Chapter 4, the main research question can now be answered by answering the three subquestions.

The first sub-question: *How do feature extraction methods such as word embeddings or TF-IDF affect the performance of ledger category classification models?* can be answered using the results from the first experiment. When comparing classifiers using TF-IDF, FastText and BERT, the results show that BERT and FastText significantly outperform the TF-IDF classifiers. One possible explanation is that the BERT and FastText approaches are pre-trained in the Dutch language which, in combination with the limited number of words per sample, makes it so that the models have an easier time generating useful embeddings for the descriptions. Furthermore, the experiment has shown a lower performance for the hierarchical SVC compared to the non-hierarchical version. This could be explained through the assumption that the performance of individual layers of the model is only marginally better compared to the non-hierarchical classifier and that each layer of the hierarchy causes a further decrease in the performance of the model.

For the second sub-question: *What is the impact of contextual data, such as company information, on the performance of classification models designed to classify ledger codes for invoice data?*, contextual data was collected and used as input to the model to supplement the textual description. The results of the experiment showed an increase in the classification performance of each of the classifiers trained in Experiment 1. The improvement in the Macro F1 score ranged from $\pm 7\%$ to $\pm 15\%$, with the DNN model trained on FastText embeddings improving the most. The BERT-based model showed smaller improvements compared to DNN, which could be explained by the even lower amount of contextual information available to the model when embedding the company name. Since BERT takes context into account for its classification, it may perform badly when that context is missing.

The final sub-question: *How scaleable are the classification models developed and what considerations should be made before they are deployed?* can be answered by examining the results of the third experiment. This experiment shows that the FastText DNN model is the fastest model to create predictions from input data. There exists a small performance difference between the version that includes contextual data and the model that does not. The experiment showed a marginal difference in performance when the prediction is performed on a GPU compared to a CPU, which means that it would not be required for the model to have access to a GPU. This will simplify a potential deployment as the costs associated with hosting servers with dedicated graphics cards are significantly higher.

The main research question: *How can classification be applied to predict the correct ledger code for Dutch invoice lines?* can now be answered. The analysis showed that it is possible to predict the ledger codes of Dutch invoice lines using their textual descriptions; furthermore, the research shows that the addition of contextual data such as price, contact name, and company SBI codes can be implemented to increase the performance of the models. The results of the experiments show that the FastText embedded DNN that implemented contextual data was the highest performing model (Macro F1 score of ±71%) that still achieved a high prediction speed, even on a CPU. Although the proposed model is still limited in the number of classes, it works well as a proof of concept that, with the inclusion of more training data, provides an effective way to predict a larger number of ledger codes. Therefore, we conclude that a DNN that implements invoice descriptions as well as contextual information, implementing a FastText embedder to process the textual data, would be the optimal approach for the classification of invoice lines for the Dutch bookkeeping software company.

The research is followed up by a prototype implementation. The prototype shows that machine learning is a viable approach to suggest ledger codes. Although the implementation should be improved before the predictions can be rolled out to customers, the prototype shows that the predictions are fast enough and generally correct. Feedback on the prototype from Moneybird developers was generally positive, as they considered the prediction speed fast enough to not slow down the users. The developers also suggested improvements and additions to the implementation that should further improve the usability of the model in their software.

6.1 Limitations

The results of the ablation study show that some contextual variables have a very limited impact on the performance of the classification model. Some contextual features could have a larger impact if further processed. For the invoice line price, the Log_{10} was taken from the absolute value of the variable, which was combined with a Boolean value indicating whether the price is positive or negative. However, this does not take the currency into account. The data set sample containing the largest value for the price feature had a value of more than 50 million in Indonesian rupiah which, when converted to euros, would be just over three thousand. It is possible that the performance of the trained models would have been better if currency conversion had been applied.

In the first two research questions, we optimised the Macro F1 score of the classifiers. During the early investigation stages, we examined text pre-processing steps commonly taken for TF-IDF embedding. In our investigation, we compared the performance of lemmatisation and stemming and concluded that lemmatisation performs better. However, in the third experiment, we concluded that the lemmatisation process is computationally expensive, making the classifiers that implement lemmatisation impractical for use in a production environment. Due to time constraints, we were unable to test the prediction speed of the TF-IDF based classifers on stemmed text. Although models using TF-IDF on stemmed text might perform worse compared to lemmatised text, they are likely to achieve significantly higher prediction speeds, making them more viable for a practical implementation.

A significant limitation found was that it was difficult to ensure that the data set was of sufficient quality. Although steps were taken to improve the quality, such as keeping only the *locked* samples of the data set for classification (as described in Section 3.1.3), there will still be samples of which it is very unclear whether they are properly labelled. Even after the removal of many uncertain samples, it was infeasible to determine whether all remaining samples were correctly labelled. This was not only due to the size of the data set but also to the ambiguity of the sometimes extremely brief textual descriptions. As an example, one of the samples that was classified as *printwork* had *"bottle opener"* as a description. The administration may have ordered, for example, a bottle opener that was engraved with their company logo, but it was impossible to determine whether this was the case from the description. Instances such as these were often misclassified by the models, as the textual description did not seem to match the assigned label.

In our research, we used a subset of the RCSFI taxonomy, only using 28 different labels in our classification. Although the taxonomy contains more than one thousand different ledger codes, only a small subset of these was sufficiently present in the data set. Due to the large skewness of the data set, only 28 ledger codes had passed the threshold set at 5000 samples. This is a limitation to the research, as well as the prototype. For research, it is possible that some ledger codes not included in this research could make it more difficult to distinguish between similar classes in the classification. The small number of classes is also a limitation for the prototype because when the correct label is not included in the model's training data set, the model will confidently misclassify the sample. This can lead to users making more mistakes in their bookkeeping as they might automatically assume that the classification is correct.

6.2 Threats to validity

A potential threat to the validity of our research might be in the preprocessing of the data. For our research, the choice was made to remove samples where it was still possible for a user to change the ledger code. This eliminated a significant number of samples that could be mislabelled. However, not every administration uses the feature that locks periods. This part of the user base has not been included in the training data of the created models. The removal may have caused a certain part of the user base to be under-represented in the database, which could lead to lower classification performance for these users compared to the users of which the data are included in the training process. Similarly, users who have not allowed their data to be used for analytical purposes could also be distributed differently from users who have given permission, causing the model to not work well for these users.

In the pre-processing steps of this research, filtering was applied to ensure that identical descriptions could not appear in both the training and test sets. Only the first occurrence of samples containing a duplicate descriptions were kept to ensure that the test set could not contain samples previously seen in training. However, this also removed samples that have an identical description as the sample that was kept but were labelled differently. Therefore, it is possible that the label that was kept is mislabelled and that the correct label for the sample was removed. The choice of filtering using this approach could have caused the data quality to decrease.

6.3 Future works

As part of our research, we developed a prototype that implemented one of the models to create predictions in the bookkeeping software. Although the prototype is functional, future works should improve the prototype to further improve the user experience by, for example, automatically creating the non-existing ledger accounts when they are selected by the user. This will reduce the effort users have to make in categorising invoice lines and further improve their experience.

With our research, a baseline application was built that can predict ledger codes based on invoice data. With an accuracy of up to 75%, it is clear that a distinction can be made between invoice lines with different ledger codes. However, there is still room for improvement. For example, future work could investigate a system in which the model could be fine-tuned for individual (large) administrations. The current system is designed to work with all administrations in the data set, but this approach could limit the potential performance of the model compared to a classifier that was adapted to work specifically with a single administration.

Another process that could be investigated in future work is the inclusion of more labels in the classifiers. Our work has only included ledger codes that have more than 5000 samples after the preprocessing, but as the available data increases as more invoices are added to the database, more labels might be able to be included in the model. Furthermore, it is interesting to investigate whether the number of samples required for a label to be included could be lowered, as this could also increase the number of classes that are included and increase the practical usefulness of the application.

The results of the third experiment showed that the TF-IDF approaches were very fast at classifying preprocessed samples, but that the preprocessing steps took very long. The hypothesis behind this is that the lemmatisation step in the text preprocessing is very long, causing the model to not scale very well. Future research could investigate whether the application of stemming instead of lemmatisation harms the classification performance of the TF-IDF based models and the impact on the speed of preprocessing.

Finally, future research should investigate the possibilities to further increase the data quality of the data set. Currently, it is difficult to determine the correctness of the labelling of a sample, which causes uncertainty in the true performance of the model. Removing samples where, for example, the textual description of the invoice line is vague or not informative might improve the performance of the model.

Chapter 7

Glossary

API Application Programming Interface. 38, 40, 51

BERT Bidirectional Encoder Representation Transformers. 15, 16, 20, 22, 31–33, 37, 43, 47, 54, 63, 65

BPE Byte-Pair encoding. 15, 16

CNN Convolutional Neural Network. 12, 21, 22

CoC Chamber of Commerce. 36

CRISP-DM CRoss-Industry Standard Process for Data Mining. 4, 5, 23

csv Comma Separated Values. 24

DNN Deep Neural Network. 11, 32, 33, 37, 38, 43–47, 54, 55, 63, 65

LR Logistic Regression. 21

NB Naive-Bayes. 21

RCSFI Reference Classification System of Financial Information. 1, 4, 5, 23, 25, 35, 36, 49, 51, 52, 56

ReLU Rectified Linear Unit. 11, 12, 37

RF Random Forest. 21

RoBERTa Robustly Optimised BERT Approach. 16

SBI Standaard BedrijfsIndeling/Standard Business Categories. 25, 28, 34, 35, 55

SVM Support Vector Machine. 8, 20, 21

Tanh Hyperbolic Tangent. 11, 12

TF-IDF Term Frequency Inverse Document Frequency. 2, 13, 21, 30, 31, 37, 54, 57

Bibliography

[1] Julien Abadji, Pedro Ortiz Suarez, Laurent Romary, and Benoît Sagot. Towards a Cleaner Document-Oriented Multilingual Crawled Corpus. 1 2022.

[2] Johan Bergdorf. *Machine learning and rule induction in invoice processing : Comparing machine learning methods in their ability to assign account codes in the bookkeeping process*. PhD thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2018.

[3] Christopher M Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. 7 2016.

[5] Wietse de Vries, Andreas van Cranenburgh, Arianna Bisazza, Tommaso Caselli, Gertjan van Noord, and Malvina Nissim. BERTje: A Dutch BERT Model. 12 2019.

[6] Pieter Delobelle, Thomas Winters, and Bettina Berendt. RobBERT: a Dutch RoBERTa-based Language Model. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3255–3265, Stroudsburg, PA, USA, 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.findings-emnlp.292.

[7] Pieter Delobelle, Thomas Winters, and Bettina Berendt. RobBERTje: A Distilled Dutch BERT Model. *Computational Linguistics in the Netherlands Journal*, 11:125–140, 12 2021. URL: https://www.clinjournal.org/clinj/article/view/131.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 10 2018.

[9] Kunihiko Fukushima. Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):322–333, 1969. doi:10.1109/TSSC.1969.300225.

[10] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning Word Vectors for 157 Languages. 2 2018.

[11] Katarzyna Janocha and Wojciech Marian Czarnecki. On Loss Functions for Deep Neural Networks in Classification. 2 2017.

[12] Samuel Johansson. Classification of Purchase Invoices to Analytic Accounts with Machine Learning. Technical report, 2022. URL: https://aaltodoc.aalto.fi/server/api/core/bitstreams/6ae6bd8b-2b59-460d-96d0-198f76cfe66f/content.

[13] Diego Santos Kieckbusch, Geraldo Pereira Rocha Filho, Vinicius Di Oliveira, and Li Weigang. Towards Intelligent Processing of Electronic Invoices: The General Framework and Case Study of Short Text Deep Learning in Brazil. pages 74–92. 2023. doi:10.1007/978-3-031-24197-0{_}5.

[14] Anders Krogh and John A Hertz. A Simple Weight Decay Can Improve Generalization. In *Neural Information Processing Systems*, 1991. URL: https://api.semanticscholar.org/CorpusID:10137788.

[15] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. 7 2019.

[16] Fábio M Miranda, Niklas Kohnecke, and Bernhard Y Renard. HiClass: a Python Library for Local Hierarchical Classification Compatible with Scikit-learn. *Journal of Machine Learning Research*, 24(29):1–17, 2023. URL: http://jmlr.org/papers/v24/21-1518.html.

[17] Nelson Morgan and Hervé Bourlard. Generalization and parameter estimation in feedforward nets: Some experiments. *Advances in neural information processing systems*, 2, 1989.

[18] Justin Munoz, Mahdi Jalili, and Laleh Tafakori. Hierarchical classification for account code suggestion. *Knowledge-Based Systems*, 251:109302, 9 2022. doi:10.1016/j.knosys.2022.109302.

[19] Jason Rennie, Lawrence Shih, Jaime Teevan, and David Karger. Tackling the Poor Assumptions of Naive Bayes Text Classifiers. *Proceedings of the Twentieth International Conference on Machine Learning*, 41, 5 2003.

[20] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958. URL: https://api.semanticscholar.org/CorpusID:12781225.

[21] Gerard Salton, Edward A. Fox, and Harry Wu. Extended Boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036, 11 1983. doi:10.1145/182.358466.

[22] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[23] R Wirth and Jochen Hipp. CRISP-DM: Towards a standard process model for data mining. *Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining*, 6 2000.

[24] Xiaoqing Zhao, Zhongyuan Jiang, and Jianfeng Ma. A Survey of Deep Anomaly Detection for System Logs. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 7 2022. doi:10.1109/IJCNN55064.2022.9892726.

Appendix A

Label hierarchy

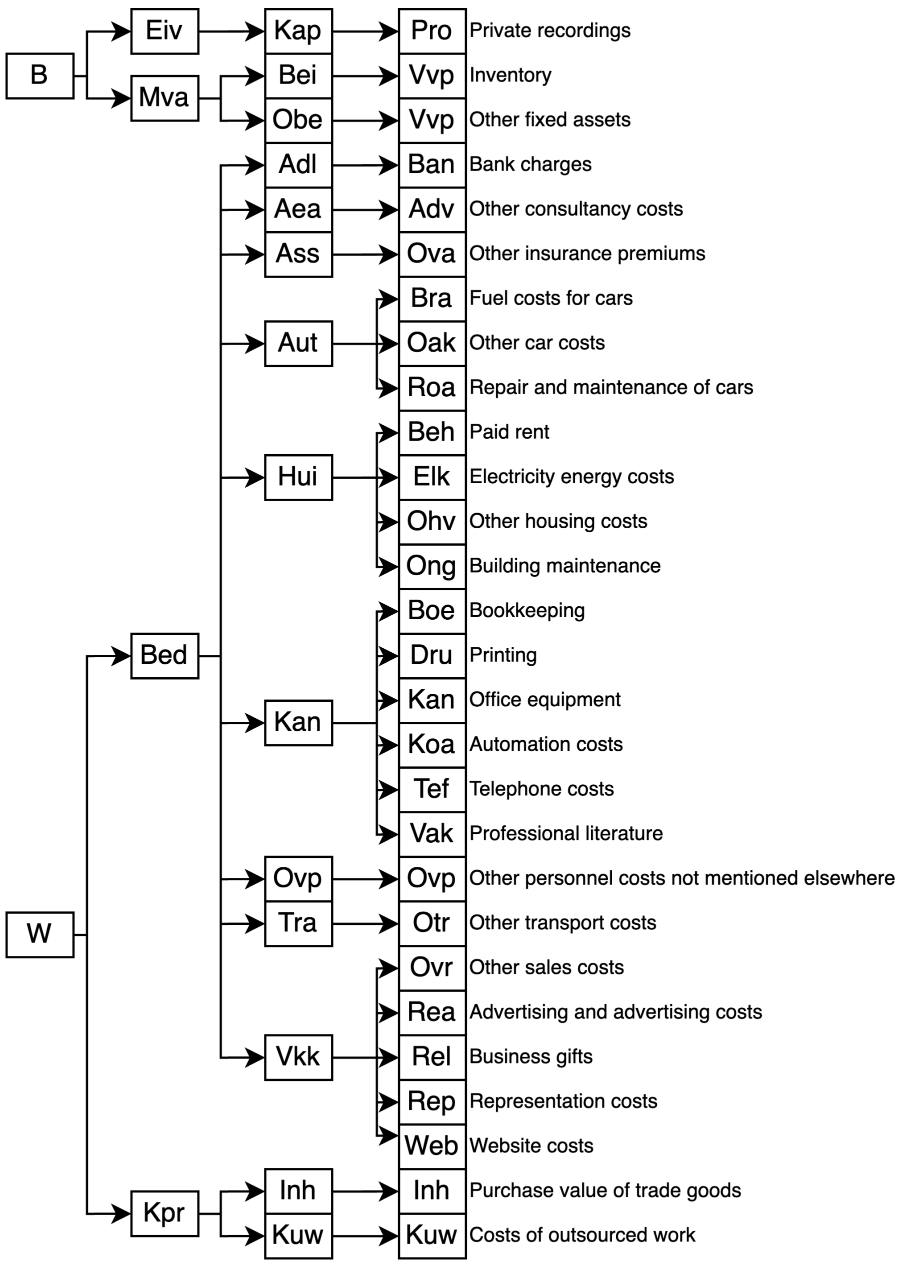


FIGURE A.1: The label hierarchy in the data set

Appendix B

Classifier validation and test scores

B.1 Experiment 1

B.1.1 Naive-bayes classifier

Fold	Macro F1
1	0.5696
2	0.5694
3	0.5665
4	0.5678
5	0.5701
mean	0.5687
std	0.0013

(A) Validation Macro F_1-scores

Metric	Score
Acc	0.6139
Macro F1	0.5582
Top 3 acc	0.7888
Top 5 acc	0.8432

(B) Test scores

Table B.1: The results for the TF-IDF embeddings with a complement naive-bayes classifier

B.1.2 Support vector classifier

Fold	Macro F1
1	0.6042
2	0.6016
3	0.6005
4	0.6021
5	0.6020
mean	0.6021
std	0.0012

(A) Validation Macro F_1-scores

Metric	Score
Acc	0.6458
Macro F1	0.5920
Top 3 acc	0.8230
Top 5 acc	0.8800

(B) Test scores

Table B.2: The results for the TF-IDF embeddings with a Linear SVC classifier

B.1.3 Hierarchical Support vector classifier

Fold	Macro F1
1	0.5506
2	0.5492
3	0.5494
4	0.5521
5	0.5506
mean	0.5540
std	0.0012

(A) Validation Macro F_1-scores

Metric	Score
Acc	0.6020
Macro F1	0.5322

(B) Test scores

Table B.3: The results for the TF-IDF embeddings with a Hierarchical Linear SVC classifier

B.1.4 FastText-based DNN classifier

Fold	Macro F1
1	0.5566
2	0.5628
3	0.5595
4	0.5601
5	0.5583
mean	0.5595
std	0.0023

(A) Validation Macro F_1-scores

Metric	Score
Acc	0.6221
Macro F1	0.5604
Top 3 acc	0.8278
Top 5 acc	0.9001

(B) Test scores

Table B.4: The results for the DNN using FastText embedded descriptions

B.1.5 BERT-based DNN classifier

Fold	Macro F1
1	0.5976
2	0.5930
3	0.6045
4	0.5975
5	0.5953
mean	0.5976
std	0.0043

(A) Validation Macro F_1-scores

Metric	Score
Acc	0.6545
Macro F1	0.6020
Top 3 acc	0.8296
Top 5 acc	0.8961

(B) Test scores

Table B.5: The results for the BERT-based model using textual descriptions

B.2 Experiment 2

B.2.1 Naive-bayes classifier

Fold	Macro F1
1	0.6452
2	0.6462
3	0.6457
4	0.6472
5	0.6468
mean	0.6462
std	0.0007

(A) Validation Macro F_1-scores

Metric	Score	Difference
Acc	0.7045	+0.0906
Macro F1	0.6382	+0.0800
Top 3 acc	0.8603	+0.0715
Top 5 acc	0.9158	+0.0726

(B) Test scores, compared to the results from Experiment 1

Table B.6: The results for the TF-IDF embeddings with a complement naive-bayes classifier trained using textual information, as well as contextual data

B.2.2 Support vector classifier

Fold	Macro F1
1	0.7055
2	0.7092
3	0.7043
4	0.7035
5	0.7070
mean	0.7059
std	0.0020

(A) Validation Macro F_1-scores

Metric	Score	Difference
Acc	0.7089	+0.0631
Macro F1	0.6743	+0.0823
Top 3 acc	0.8572	+0.0342
Top 5 acc	0.9080	+0.0280

(B) Test scores, compared to the results from Experiment 1

Table B.7: The results for the TF-IDF embeddings with a Linear SVC classifier trained using textual information, as well as contextual data

B.2.3 Hierarchical Support vector classifier

Fold	Macro F1
1	0.6502
2	0.6521
3	0.6507
4	0.6518
5	0.6531
mean	0.6516
std	0.0012

(A) Validation Macro F_1-scores

Metric	Score	Difference
Acc	0.6549	+0.0529
Macro F1	0.5987	+0.0665

(B) Test scores, compared to the results from the same classifier in Experiment 1

Table B.8: The results for the TF-IDF embeddings with a Hierarchical Linear SVC classifier using both textual descriptions and contextual information

B.2.4 FastText-based DNN classifier

Fold	Macro F1
1	0.7107
2	0.7021
3	0.7089
4	0.7092
5	0.7036
mean	0.7069
std	0.0038

(A) Validation Macro F_1-scores

Metric	Score	Difference
Acc	0.7594	+0.1373
Macro F1	0.7092	+0.1488
Top 3 acc	0.9065	+0.0787
Top 5 acc	0.9490	+0.0489

(B) Test scores, compared to the results from Experiment 1

Table B.9: The results for the DNN trained on FastText embedded textual data and contextual data

B.2.5 BERT-based DNN classifier

Fold	Macro F1
1	0.6906
2	0.6872
3	0.6914
4	0.6809
5	0.6846
mean	0.6869
std	0.0043

(A) Validation Macro F_1-scores

Metric	Score	Difference
Acc	0.7385	+0.0840
Macro F1	0.6808	+0.0788
Top 3 acc	0.8920	+0.0624
Top 5 acc	0.9404	+0.0443

(B) Test scores, compared to the results from Experiment 1

Table B.10: The results for the BERT-based model using contextual data

Appendix C

Hyperparameter tuning Results

C.1 BERT without contextual data

learning rate	weight decay	dropout	gamma	accuracy	F1_score	loss
0.0001	1e-05	0.4	0.85	0.6537	0.5938	1.3801
0.0001	1e-05	0.4	0.9	0.6553	0.5916	1.3587
0.0001	1e-05	0.5	0.85	0.6577	0.5929	1.3436
0.0001	1e-05	0.5	0.9	0.6545	0.589	1.2963
0.0001	1e-06	0.4	0.85	0.6593	0.5983	1.388
0.0001	1e-06	0.4	0.9	0.6562	0.5922	1.2875
0.0001	1e-06	0.5	0.85	0.6582	0.5963	1.3908
0.0001	1e-06	0.5	0.9	0.6564	0.5949	1.3417
5e-05	1e-05	0.4	0.85	0.6561	0.5927	1.3204
5e-05	1e-05	0.4	0.9	0.6568	0.5938	1.3102
5e-05	1e-05	0.5	0.85	0.6584	0.5969	1.3118
5e-05	1e-05	0.5	0.9	0.656	0.5896	1.3017
5e-05	1e-06	0.4	0.85	0.6588	0.5934	1.317
5e-05	1e-06	0.4	0.9	0.659	0.5978	1.3177
5e-05	1e-06	0.5	0.85	0.6587	0.5953	1.2994
5e-05	1e-06	0.5	0.9	0.6579	0.5943	1.2998
1e-05	1e-05	0.4	0.85	0.6407	0.5611	1.2704
1e-05	1e-05	0.4	0.9	0.6465	0.571	1.2683
1e-05	1e-05	0.5	0.85	0.6414	0.5623	1.2789
1e-05	1e-05	0.5	0.9	0.644	0.567	1.2745
1e-05	1e-06	0.4	0.85	0.6391	0.5586	1.2715
1e-05	1e-06	0.4	0.9	0.6449	0.5675	1.2691
1e-05	1e-06	0.5	0.85	0.641	0.5611	1.2813
1e-05	1e-06	0.5	0.9	0.6419	0.564	1.2772
5e-06	1e-05	0.4	0.85	0.6209	0.5314	1.314
5e-06	1e-05	0.4	0.9	0.6339	0.5508	1.2861
5e-06	1e-05	0.5	0.85	0.6188	0.5271	1.3246
5e-06	1e-05	0.5	0.9	0.6323	0.5477	1.2936
5e-06	1e-06	0.4	0.85	0.62	0.5301	1.3177
5e-06	1e-06	0.4	0.9	0.6351	0.5529	1.2839
5e-06	1e-06	0.5	0.85	0.6182	0.5268	1.3262
5e-06	1e-06	0.5	0.9	0.6316	0.5474	1.2968

Table C.1: The macro F1 scores for the different hyperparameter combinations in the BERT-based classifier trained without the contextual data

C.2 BERT with contextual data

learning rate	**weight decay**	**dropout**	**gamma**	**accuracy**	**F1 score**	**loss**
0.0001	1e-05	0.4	0.85	0.7418	0.6884	1.1307
0.0001	1e-05	0.4	0.9	0.741	0.6871	1.0617
0.0001	1e-05	0.5	0.85	0.7423	0.6875	1.0843
0.0001	1e-05	0.5	0.9	0.7423	0.6856	1.0985
0.0001	1e-06	0.4	0.85	0.7437	0.6868	1.1351
0.0001	1e-06	0.4	0.9	0.7361	0.6821	1.0145
0.0001	1e-06	0.5	0.85	0.7394	0.6829	1.0963
0.0001	1e-06	0.5	0.9	0.7396	0.6855	1.1031
5e-05	1e-05	0.4	0.85	0.7419	0.6809	1.0327
5e-05	1e-05	0.4	0.9	0.7409	0.6862	1.0526
5e-05	1e-05	0.5	0.85	0.7421	0.6854	1.0273
5e-05	1e-05	0.5	0.9	0.7387	0.6795	1.0729
5e-05	1e-06	0.4	0.85	0.741	0.6861	1.0384
5e-05	1e-06	0.4	0.9	0.7427	0.6848	1.0404
5e-05	1e-06	0.5	0.85	0.7432	0.6837	1.0355
5e-05	1e-06	0.5	0.9	0.7419	0.6866	1.04
1e-05	1e-05	0.4	0.85	0.7235	0.6486	0.9823
1e-05	1e-05	0.4	0.9	0.7242	0.6509	0.9807
1e-05	1e-05	0.5	0.85	0.7202	0.6458	0.9914
1e-05	1e-05	0.5	0.9	0.7236	0.6533	0.992
1e-05	1e-06	0.4	0.85	0.7203	0.645	0.9916
1e-05	1e-06	0.4	0.9	0.727	0.6572	0.9849
1e-05	1e-06	0.5	0.85	0.7197	0.6451	1.0001
1e-05	1e-06	0.5	0.9	0.7276	0.6568	0.9946
5e-06	1e-05	0.4	0.85	0.6937	0.604	1.0444
5e-06	1e-05	0.4	0.9	0.7124	0.6337	1.002
5e-06	1e-05	0.5	0.85	0.6918	0.5989	1.0578
5e-06	1e-05	0.5	0.9	0.7064	0.6243	1.0198
5e-06	1e-06	0.4	0.85	0.6933	0.6022	1.0473
5e-06	1e-06	0.4	0.9	0.711	0.6321	1.0033
5e-06	1e-06	0.5	0.85	0.6923	0.6006	1.0561
5e-06	1e-06	0.5	0.9	0.711	0.6321	1.0108

Table C.2: The macro F1 scores for the different hyperparameter combinations in the BERT-based classifier trained with the contextual data

C.3 FastText without contextual data

learning rate	weight decay	dropout	gamma	accuracy	F1 score	loss
0.001	1e-05	0.4	0.9	0.6198	0.5486	1.3126
0.001	1e-05	0.5	0.9	0.6082	0.5326	1.3528
0.001	1e-05	0.4	0.95	0.6268	0.5611	1.3135
0.001	1e-05	0.5	0.95	0.6165	0.5479	1.3379
0.001	1e-05	0.4	0.97	0.6190	0.5539	1.3417
0.001	1e-05	0.5	0.97	0.6161	0.5498	1.3494
0.0005	1e-05	0.4	0.9	0.6111	0.5312	1.3370
0.0005	1e-05	0.5	0.9	0.5976	0.5143	1.3853
0.0005	1e-05	0.4	0.95	0.6221	0.5506	1.3149
0.0005	1e-05	0.5	0.95	0.6113	0.5354	1.3514
0.0005	1e-05	0.4	0.97	0.6216	0.5534	1.3215
0.0005	1e-05	0.5	0.97	0.6002	0.5251	1.3856
0.001	5e-06	0.4	0.9	0.6212	0.5534	1.3157
0.001	5e-06	0.5	0.9	0.6089	0.5359	1.3589
0.001	5e-06	0.4	0.95	0.6242	0.5610	1.3300
0.001	5e-06	0.5	0.95	0.6172	0.5480	1.3516
0.001	5e-06	0.4	0.97	0.6172	0.5520	1.3564
0.001	5e-06	0.5	0.97	0.6126	0.5489	1.3661
0.0005	5e-06	0.4	0.9	0.6103	0.5335	1.3373
0.0005	5e-06	0.5	0.9	0.5984	0.5166	1.3838
0.0005	5e-06	0.4	0.95	0.6227	0.5525	1.3151
0.0005	5e-06	0.5	0.95	0.6103	0.5382	1.3531
0.0005	5e-06	0.4	0.97	0.6204	0.5524	1.3330
0.0005	5e-06	0.5	0.97	0.6133	0.5445	1.3592
0.001	1e-06	0.4	0.9	0.6213	0.5535	1.3235
0.001	1e-06	0.5	0.9	0.6083	0.5366	1.3646
0.001	1e-06	0.4	0.95	0.6215	0.5571	1.3466
0.001	1e-06	0.5	0.95	0.6157	0.5511	1.3622
0.001	1e-06	0.4	0.97	0.6130	0.5464	1.3702
0.001	1e-06	0.5	0.97	0.6046	0.5397	1.4019
0.0005	1e-06	0.4	0.9	0.6107	0.5329	1.3376
0.0005	1e-06	0.5	0.9	0.5986	0.5188	1.3858
0.0005	1e-06	0.4	0.95	0.6230	0.5557	1.3219
0.0005	1e-06	0.5	0.95	0.6096	0.5386	1.3608
0.0005	1e-06	0.4	0.97	0.6182	0.5506	1.3403
0.0005	1e-06	0.5	0.97	0.6088	0.5394	1.3694

Table C.3: The macro F1 scores for the different hyperparameter combinations in the FastText-based classifier trained without the contextual data

C.4 FastText with contextual data

learning rate	**weight decay**	**dropout**	**gamma**	**accuracy**	**F1 score**	**loss**
0.001	1e-05	0.4	0.9	0.7564	0.7012	0.8504
0.001	1e-05	0.4	0.9	0.7546	0.6986	0.8547
0.001	1e-05	0.4	0.9	0.7581	0.7056	0.8486
0.001	1e-05	0.5	0.9	0.7478	0.6914	0.8636
0.001	1e-05	0.4	0.95	0.7584	0.7077	0.8646
0.001	1e-05	0.5	0.95	0.7564	0.7055	0.8526
0.001	1e-05	0.4	0.97	0.7540	0.7025	0.8853
0.001	1e-05	0.5	0.97	0.7579	0.7061	0.8615
0.0005	1e-05	0.4	0.9	0.7478	0.6917	0.8582
0.0005	1e-05	0.5	0.9	0.7372	0.6749	0.8898
0.0005	1e-05	0.4	0.95	0.7536	0.7015	0.8565
0.0005	1e-05	0.5	0.95	0.7502	0.6961	0.8628
0.0005	1e-05	0.4	0.97	0.7542	0.7013	0.8706
0.0005	1e-05	0.5	0.97	0.7564	0.7044	0.8542
0.001	5e-06	0.4	0.9	0.7583	0.7060	0.8585
0.001	5e-06	0.5	0.9	0.7481	0.6931	0.8661
0.001	5e-06	0.4	0.95	0.7570	0.7064	0.8805
0.001	5e-06	0.5	0.95	0.7572	0.7060	0.8655
0.001	5e-06	0.4	0.97	0.7482	0.6943	0.8988
0.001	5e-06	0.5	0.97	0.7484	0.6955	0.8920
0.0005	5e-06	0.4	0.9	0.7483	0.6908	0.8626
0.0005	5e-06	0.5	0.9	0.7380	0.6781	0.8892
0.0005	5e-06	0.4	0.95	0.7557	0.7039	0.8615
0.0005	5e-06	0.5	0.95	0.7499	0.6946	0.8656
0.0005	5e-06	0.4	0.97	0.7548	0.7035	0.8782
0.0005	5e-06	0.5	0.97	0.7518	0.7005	0.8678
0.001	1e-06	0.4	0.9	0.7548	0.7018	0.8732
0.001	1e-06	0.5	0.9	0.7467	0.6931	0.8748
0.001	1e-06	0.4	0.95	0.7518	0.6995	0.9034
0.001	1e-06	0.5	0.95	0.7556	0.7027	0.8917
0.001	1e-06	0.4	0.97	0.7427	0.6871	0.9208
0.001	1e-06	0.5	0.97	0.7504	0.6977	0.9076
0.0005	1e-06	0.4	0.9	0.7473	0.6915	0.8666
0.0005	1e-06	0.5	0.9	0.7391	0.6788	0.8867
0.0005	1e-06	0.4	0.95	0.7574	0.7067	0.8677
0.0005	1e-06	0.5	0.95	0.7492	0.6949	0.8726
0.0005	1e-06	0.4	0.97	0.7540	0.7031	0.8849
0.0005	1e-06	0.5	0.97	0.7520	0.7000	0.8797

Table C.4: The macro F1 scores for the different hyperparameter combinations in the FastText-based classifier trained with the contextual data

C.5 Linear SVC without contextual data

C	ngram range	fold 1	fold 2	fold 3	fold 4	fold 5	mean	std
0.1	(1, 1)	0.5433	0.5448	0.5419	0.5446	0.5427	0.5435	0.0011
0.1	(1, 2)	0.5562	0.5575	0.5558	0.5604	0.5572	0.5574	0.0016
0.1	(1, 3)	0.5509	0.5522	0.5486	0.5537	0.55	0.5511	0.0018
0.3	(1, 1)	0.5692	0.57	0.567	0.5686	0.5681	0.5686	0.001
0.3	(1, 2)	0.5945	0.5934	0.5902	0.5941	0.593	0.593	0.0015
0.3	(1, 3)	0.5893	0.5896	0.5851	0.5897	0.5875	0.5882	0.0018
0.7	(1, 1)	0.5736	0.5744	0.5718	0.5731	0.5736	0.5733	0.0008
0.7	(1, 2)	0.6036	0.6024	0.6001	0.6016	0.6024	0.602	0.0011
0.7	(1, 3)	0.5992	0.5985	0.5965	0.598	0.598	0.5981	0.0009
0.9	(1, 1)	0.5727	0.5742	0.5729	0.5727	0.5731	0.5731	0.0006
0.9	(1, 2)	0.6042	0.6016	0.6005	0.6021	0.6021	0.6021	0.0012
0.9	(1, 3)	0.5998	0.5982	0.5964	0.5989	0.5978	0.5982	0.0011

TABLE C.5: The macro F1 scores for the different hyperparameter combinations in the TFIDF-embedded SVC classifier trained without the contextual data

C.6 Linear SVC with contextual data

C	ngram range	fold 1	fold 2	fold 3	fold 4	fold 5	mean	std
0.1	(1, 1)	0.6764	0.6793	0.677	0.674	0.6778	0.6769	0.0017
0.1	(1, 2)	0.6774	0.6802	0.6773	0.674	0.6771	0.6772	0.0019
0.1	(1, 3)	0.6738	0.6762	0.6747	0.671	0.6739	0.6739	0.0017
0.3	(1, 1)	0.6966	0.698	0.6946	0.6939	0.6966	0.696	0.0015
0.3	(1, 2)	0.7	0.7027	0.6992	0.6999	0.7007	0.7005	0.0012
0.3	(1, 3)	0.6977	0.7011	0.6972	0.6979	0.6989	0.6986	0.0014
0.7	(1, 1)	0.6983	0.6991	0.6959	0.6951	0.6998	0.6976	0.0018
0.7	(1, 2)	0.7055	0.7088	0.7037	0.705	0.7065	0.7059	0.0017
0.7	(1, 3)	0.7023	0.7077	0.7036	0.7025	0.7049	0.7042	0.002
0.9	(1, 1)	0.6974	0.6987	0.6952	0.694	0.6978	0.6966	0.0017
0.9	(1, 2)	0.7056	0.7093	0.7044	0.7036	0.707	0.706	0.002
0.9	(1, 3)	0.7021	0.7075	0.7045	0.7028	0.7049	0.7043	0.0019

TABLE C.6: The macro F1 scores for the different hyperparameter combinations in the TFIDF-embedded SVC classifier trained with the contextual data

C.7 Naive-Bayes without contextual data

alpha	**ngram range**	**fold 1**	**fold 2**	**fold 3**	**fold 4**	**fold 5**	**mean**	**std**
0	(1, 1)	0.5119	0.5125	0.511	0.5119	0.5128	0.512	0.0006
0	(1, 2)	0.5458	0.5454	0.544	0.5464	0.5475	0.5458	0.0012
0	(1, 3)	0.5369	0.5369	0.5363	0.5369	0.5384	0.537	0.0007
0.5	(1, 1)	0.519	0.5203	0.5179	0.5196	0.5202	0.5194	0.0009
0.5	(1, 2)	0.5674	0.5668	0.5637	0.5665	0.5678	0.5664	0.0014
0.5	(1, 3)	0.5696	0.5694	0.5665	0.5678	0.5701	0.5687	0.0013
1	(1, 1)	0.5199	0.5206	0.5189	0.5214	0.5205	0.5203	0.0009
1	(1, 2)	0.5651	0.5653	0.5605	0.5635	0.5659	0.5641	0.0019
1	(1, 3)	0.568	0.5684	0.5625	0.5662	0.5672	0.5665	0.0021

Table C.7: The macro F1 scores for the different hyperparameter combinations in the TFIDF-embedded Complement Naive-Bayes classifier trained without the contextual data

C.8 Naive-Bayes with contextual data

alpha	**ngram range**	**fold 1**	**fold 2**	**fold 3**	**fold 4**	**fold 5**	**mean**	**std**
0	(1, 1)	0.6063	0.6091	0.6097	0.6064	0.6079	0.6079	0.0014
0	(1, 2)	0.6152	0.6143	0.6157	0.6154	0.6176	0.6157	0.0011
0	(1, 3)	0.605	0.6036	0.6068	0.605	0.6062	0.6053	0.0011
0.5	(1, 1)	0.623	0.6238	0.6227	0.6214	0.6235	0.6229	0.0008
0.5	(1, 2)	0.6459	0.646	0.6458	0.646	0.6457	0.6459	0.0001
0.5	(1, 3)	0.6452	0.6462	0.6457	0.6472	0.6468	0.6462	0.0007
1	(1, 1)	0.6201	0.6208	0.6188	0.6178	0.6211	0.6197	0.0012
1	(1, 2)	0.6348	0.6362	0.6358	0.6342	0.6369	0.6356	0.001
1	(1, 3)	0.6321	0.6333	0.6322	0.6321	0.6353	0.633	0.0012

Table C.8: The macro F1 scores for the different hyperparameter combinations in the TFIDF-embedded Complement Naive-Bayes classifier trained with the contextual data

C.9 Hierarchical SVC without contextual data

ngram range	C	fold 1	fold 2	fold 3	fold 4	fold 5	mean	std
(1, 1)	0.1	0.4845	0.4845	0.484	0.4883	0.4843	0.4851	0.0016
(1, 1)	0.3	0.5138	0.5145	0.5117	0.5161	0.5152	0.5143	0.0015
(1, 1)	0.7	0.5235	0.5233	0.5214	0.5245	0.5243	0.5234	0.0011
(1, 1)	0.9	0.5245	0.5234	0.5225	0.5248	0.526	0.5242	0.0012
(1, 2)	0.1	0.4947	0.4916	0.4921	0.4966	0.4933	0.4937	0.0018
(1, 2)	0.3	0.535	0.5333	0.532	0.5349	0.5342	0.5339	0.0011
(1, 2)	0.7	0.5491	0.548	0.548	0.5514	0.5491	0.5491	0.0012
(1, 2)	0.9	0.5506	0.5492	0.5494	0.5521	0.5506	0.5504	0.001
(1, 3)	0.1	0.4877	0.4856	0.4859	0.4902	0.4861	0.4871	0.0017
(1, 3)	0.3	0.5282	0.5286	0.5277	0.5324	0.5283	0.529	0.0017
(1, 3)	0.7	0.5455	0.5436	0.5438	0.5463	0.5446	0.5448	0.001
(1, 3)	0.9	0.5467	0.5445	0.5456	0.5475	0.546	0.5461	0.001

Table C.9: The macro F1 scores for the different hyperparameter combinations in the TFIDF-embedded Hierarchical SVC classifier trained without the contextual data

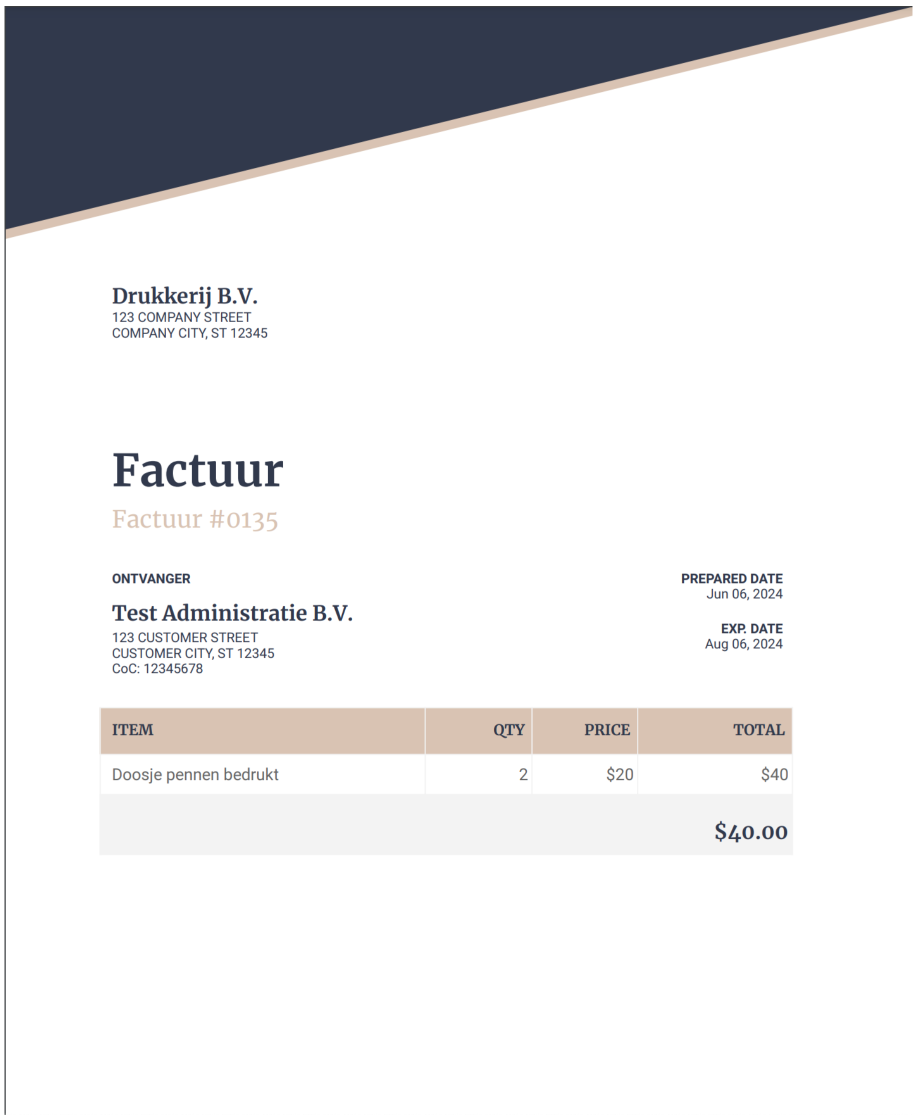
C.10 Hierarchical SVC with contextual data

ngram range	C	fold 1	fold 2	fold 3	fold 4	fold 5	mean	std
(1, 1)	0.1	0.6171	0.6176	0.6174	0.6144	0.6189	0.6171	0.0015
(1, 1)	0.3	0.6395	0.6384	0.6395	0.637	0.6413	0.6391	0.0014
(1, 1)	0.7	0.6444	0.6438	0.6426	0.6427	0.6458	0.6439	0.0012
(1, 1)	0.9	0.6437	0.6428	0.6416	0.642	0.6453	0.6431	0.0013
(1, 2)	0.1	0.6178	0.6181	0.6187	0.617	0.6184	0.618	0.0006
(1, 2)	0.3	0.643	0.6437	0.6438	0.6428	0.6467	0.644	0.0014
(1, 2)	0.7	0.6492	0.6515	0.6504	0.6523	0.6523	0.6511	0.0012
(1, 2)	0.9	0.6502	0.6521	0.6507	0.6518	0.6531	0.6516	0.001
(1, 3)	0.1	0.6145	0.6154	0.616	0.6145	0.6146	0.615	0.0006
(1, 3)	0.3	0.6397	0.6416	0.6414	0.6411	0.6434	0.6414	0.0012
(1, 3)	0.7	0.6478	0.6489	0.6491	0.6491	0.651	0.6492	0.001
(1, 3)	0.9	0.6492	0.6497	0.6498	0.65	0.6509	0.6499	0.0006

Table C.10: The macro F1 scores for the different hyperparameter combinations in the TFIDF-embedded Hierarchical SVC classifier trained with the contextual data

Appendix D

Prototype example invoice



Drukkerij B.V.
123 COMPANY STREET
COMPANY CITY, ST 12345

Factuur

Factuur #0135

ONTVANGER

Test Administratie B.V.
123 CUSTOMER STREET
CUSTOMER CITY, ST 12345
CoC: 12345678

PREPARED DATE
Jun 06, 2024

EXP. DATE
Aug 06, 2024

ITEM	QTY	PRICE	TOTAL
Doosje pennen bedrukt	2	$20	$40
			$40.00

FIGURE D.1: The invoice given to the developers testing the prototype