Evaluating Text Classification Models on Multilingual Documents

Master Thesis

Julia Eigenmann

Home University
University of Fribourg

Supervisors:
Ines Arous
Dr. Mourad Khayati
Prof. Dr. Philippe Cudré-Mauroux

eXascale Infolab
Department of Informatics, University of Fribourg

September 02 , 2021

Abstract

Machine learning models often require large annotated datasets for training in order to obtain accurate results. However, the scarcity of the labeled data is a bottleneck for many applications, including text classification. The problem becomes even more challenging in the case of multilingual textual documents. In such a case, annotators are required to be experts in annotating the data in different languages. Existing methods have limited performance on classifying textual documents by using only a small set of labeled data.

In this thesis, we propose solving this problem by using heuristic rules to label a large set of multilingual documents and apply different classification models to them. We compare language-dependent with language-independent classification approaches and report the results of our comparison. Our results show that:

- Language-independent classifiers perform overall better than the language-dependent ones for underrepresented languages; this is probably due to their small training dataset. Language-dependent classifiers with large training dataset might outperform the language-independent classifiers with training dataset of comparable size.
- Linear SVC, Random Forest, FastText, Logistic Regression and DistilBert are well-performing classification models, whereas Multinomial Naive Bayes achieves only satisfying performance results. DistilBert performs best.

Acknowledgements

I want to thank all the people involved in the work of this thesis, in particular, highly-experienced PhD student Ines Arous, for her continuous assistance and patient coaching. She significantly contributed to the process of this master thesis, whose realization would not have been possible without her expertise and help.

Many thanks to Prof. Dr. Philippe Cudré-Mauroux and Dr. Mourad Khayati for their organization and kind support.

And, of course, special thanks to the computer engineering company *Softcom Technologies SA* for offering this fruitful opportunity and for their generous cooperation, notably to Laure Zurkinden and Pascal Meyrat, who meticulously gathered the first labeled data.

Thanks to my friends and family which supported and accompanied me through the challenges during the thesis work.

Contents

List of Figures

Chapter 1

Introduction

Supervised learning techniques build an accurate prediction model by learning from a large amount of labeled data. The labeling process is not an easy task: Sometimes, it needs domain-specific expert knowledge while the data are labeled by hand. In text classification, the language and the specificity of domains are another burden for gathering large amounts of data, labeled or unlabeled. Most languages are under-resourced to train a successful prediction model, or the resources are not evenly available in all domains. Moreover, the languages and domains are permanently evolving. Therefore, multilingual classification methods are of particular interest.

A practical use case is provided by the computer engineering company *Softcom Technologies SA*. This company searches for potential clients by scanning through the work projects advertised by the government procurement of Switzerland. These announced projects are published on the online platform `simap.ch` and represent textual data in different languages. The scanning process of *Softcom* consists of the following two steps: First, an employee manually studies each single project description on `simap.ch` that recently has been tendered. Then, a group of experts reviews the projects received from the first step, filtering a second time. Only a small percentage among the invitations to tender published on `simap.ch` are saved (perceived 5%), and for these, *Softcom* considers applying for the invitation to tender. However, this scanning process might miss potentially interesting projects. Further, manually scanning through the entire website is time-consuming and is even more challenging due to the multilingual aspect of the data. A convenient automatic scanning system simplifies this workload and delivers a more accurate detection of interesting projects.

Most existing methods do not distinguish between different languages within a dataset. Some techniques are multilingual by design, such as the multilingual Bert (Devlin et al., 2019; Tanskanen, 2020; Liu et al., 2020; Pires, Schlinger, and Garrette, 2019). The latter learns word embedding that generalizes across multiple languages. These cross-lingual word embeddings are learned in a shared vector space for several languages. Cross-lingual word embeddings are less accurate for some language combinations such as English and Japanese.

In this thesis, we compare six different classification models to provide Softcom with tendering documents describing services that match their expertise. The models aim more concretely at classifying accurately the projects from `simap.ch` into *interesting* and *not interesting*. A project is considered as interesting for the company *Softcom* if the demand matches with their services. For developing the models, *Softcom* kindly provided us with a dataset of some ground-truth labeled project examples. These examples have been labeled twice; once by the employee during the first step and once by the experts in the second step. The appended label criteria additionally help understand their label decision. By intuitively analyzing the dataset with criteria supplied by Softcom, we create a labeling procedure that annotates the

rest of the unlabeled data to extend the amount of labeled data. By combining existing text representation algorithms with existing text classification algorithms, we configure the classification models. With a large amount of labeled data and the classification models, we proceed in two main folds. First, we fine-tune the hyperparameters of the models. Second, with the tuned hyperparameters, the models undergo two different language-related classification approaches to handle the multilingual aspect of the dataset: language-dependently and -independently. This master thesis provides a comparison and analysis of the six tuned models and the two language-related classification techniques. As a result, we conclude that language-independent classifiers perform overall better than the language-dependent ones, especially for underrepresented languages, and Linear SVC, Random Forest, FastText, Logistic Regression and DistilBert are all well-performing classification models. DistilBert performs best. The language-dependent approach might outperform the language-independent approach when the training dataset is large enough.

This thesis is structured as follows: Chapter 2 introduces the algorithms used for the experiments, chapter 3 presents the steps of our work methodology within this thesis, chapter 4 shows the experiments and their results as well as an analysis of the results, chapter 5 gives a conclusive summary of the entire thesis.

Chapter 2

Background

In this chapter, we introduce the text representation and classification algorithms that have been used in the experiments. A text representation algorithm translates text documents into numerical word features. A text classification algorithm automatizes the classification of textual data. We clarify the functioning of the algorithms, present their training objective and classifier formula.

2.0.1 Term Frequency - Inverse Dense Frequency (TF-IDF)

The Term Frequency-Inverse Dense Frequency (TF-IDF) algorithm (Sammut and Webb, 2010) is a text representation algorithm that determines the weight of a word in a particular document within the collection of documents. This is done by combining the Term Frequency TF (Luhn, 1958) of a vocabulary word with its Inverse Document Frequency IDF (Jones, 1972). The TF of a word describes its frequency in a document, and the IDF expresses its frequency within a collection of documents. The TF of a word t in document d is computed as follows:

$$TF(t,d) = \frac{\text{frequency of } t \text{ in } d}{\text{length of } d} \tag{2.1}$$

And we calculate the IDF of a word t within a collection D with following formula:

$$IDF(t,D) = \log \frac{D}{D_t} \tag{2.2}$$

Where D_t represents the number of documents where the word t occurs. The TF-IDF of a word t is then expressed as follows:

$$TFIDF(t,d,D) = TF(t,d) \cdot IDF(t,D) \tag{2.3}$$

A document is represented as a V-sized vector of TF-IDF scores, where each score corresponds to a vocabulary word. The V is the vocabulary size.

Since the TF-IDF represents the weight of importance of words within a document, this helps to compare the similarity between the documents. However, with TF-IDF we cannot compare the similarity between the words; it gives no information on the semantics of the words as with word embeddings.

2.0.2 FastText as word embedding algorithm

FastText (Bojanowski et al., 2016; Joulin et al., 2016) is a method that works as a word embedding algorithm or as a text classification algorithm. In this section, we are

going to introduce the two Word2Vec neural network models for learning word representations in FastText: the skip-gram and CBOW (Mikolov et al., 2013a; Mikolov et al., 2013b). These models are trained by considering each word's context within a sentence.

The skip-gram model predicts the neighboring words given the word in the middle, while the CBOW model predicts the word in the middle given the neighboring words. Their ultimate goal is to optimize the weight matrices, which are the word representations, by training an accurate prediction model.

In FastText, the skip-gram model and CBOW model are extended such that they can consider the subword information of a word. The models learn word embeddings such that each word embedding is computed as the sum of the word's character n-grams embeddings. This helps to represent better rare words or morphologically rich languages like German. Training FastText for word embeddings tends, therefore, to be slower and requires more memory than training Word2Vec (Di, Bhardwaj, and Wei, 2018). Even though FastText is trained while considering each word's neighboring words within a sentence, it is said to be non-contextual, meaning that there is only one embedding for one word no matter its context; e.g., the word *bank* as a seat or as money deposit has the exactly same embedding.

The training objective of the skip-gram model is to maximize the probability of predicting correctly or minimize the probability of predicting each word context for a given target word wrongly by optimizing its weight matrices: The input weight matrix W_i and the output weight matrix W_o. The input weight matrix W_i is the one that encodes the word representations that we are looking for. Let's have a context window of size c and the input target word w_t. Then the c next neighbors to the left as well as to the right from the target word in a text are considered as the context words of the target word (i.e. from w_{t-1} to w_{t-c} and from w_{t+1} to w_{t+c}). Based on this window panel, we compose training samples to feed the neural network. A training sample is defined by the target word (input) and a context word (output), such that all context words appear once together with the target word within the set of training samples. Therefore, for each target word, we create $2 \cdot c = C$ training samples. While training, the window panel shifts from target word to target word through all training documents. Each word token in the documents is therefore handled once as a target word. Suppose that all training documents together contain in total T word tokens. Then we can formulate the skip-gram model's training objective as follows (Bojanowski et al., 2016; Mikolov et al., 2013a; Nalisnick and Ravi, 2017)[1]:

$$\arg\min_{W_i, W_o} \left[-\sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} w_{t+j} \log p(w_{t+j} | w_t) \right] \tag{2.4}$$

At each training iteration, i.e., with each training sample, we do the following: The input is a V-sized one-hot encoded vector representing the target word w_t, where V is the vocabulary size. The input is multiplied by the input weight matrix W_i of size $V \times D$ to project the target word embedding vector of size D, where D is the dimension hyperparameter (see section 4.5.1). The input weight matrix W_i acts like a look-up table for word embeddings: Each row represents the word embedding vector for a particular word in the vocabulary. The D-sized vector output is, therefore, the word embedding for the target word w_t. The projected target word embedding is then multiplied by the output weight matrix W_o of size $D \times V$ outputting a V-sized vector. The latter in turn is fed into a softmax function to predict a

[1]The cost function has been translated into cross-entropy loss function

probability distribution over the vocabulary $p(w_{t+j}|w_t)$; for each word in the vocabulary is computed the probability for being the context word w_{t+j} of w_t. We take the difference between the predicted probability distribution $p(w_{t+j}|w_t)$ and the actual probability distribution w_{t+j}, where the latter is represented as a V-sized one-hot vector. This difference we call the prediction error. We then update the weights W_i and W_o through backpropagation such that the prediction error is minimized (Kim et al., 2017; Kim, 2019a; Kim, 2019b).

The training objective of the CBOW model is to maximize the probability of predicting correctly or to minimize the probability of predicting wrongly the word in the middle (the target word), given the neighboring words, by optimizing its weight matrices. A training sample consists of the sum of all context words (input) and the target word (output). Therefore for each target word we create 1 training sample. We can formulate the CBOW model's training objective as follows (Nalisnick and Ravi, 2017)[2]:

$$\arg\min_{W_i, W_o} \left[-\sum_{t=1}^{T} w_t \log p(w_t|w_{t-c}, ..., w_{t+c}) \right] \tag{2.5}$$

The training procedure of the CBOW model is similar to the skip-gram model. At each training iteration, i.e. with each training sample, we do the following: The input is the sum of all C context words, where each context word is represented as a V-sized one-hot vector. The input is multiplied by the input weight matrix W_i of size $V \times D$ to project the sum of the word embeddings of the context vectors, which is a single D-sized vector. The D-sized vector output is divided by the number of context words C to output an averaged word embedding. This averaged D-sized word embedding is multiplied by the output matrix W_o of size $D \times V$ to output a V-sized vector. The latter is fed into a softmax function to predict a probability distribution over the vocabulary $p(w_t|w_{t-c}, ..., w_{t+c})$; For each word in the vocabulary is computed the probability for being the target word w_t of the context words $w_{t-c}, ..., w_{t+c}$. We take the difference between the predicted probability distribution $p(w_t|w_{t-c}, ..., w_{t+c})$ and the actual probability distribution w_t, where the latter is represented as a V-sized one-hot vector. We then update the weights W_i and W_o through backpropagation such that the prediction error is minimized (Sahil, 2021).

2.0.3 Bert as word embedding algorithm (the pre-training Bert model)

The Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2019; Vaswani et al., 2017) is a method that generates word embeddings in the *pre-training* phase as well as performs downstream natural language processing (NLP) tasks such as text classification in the *fine-tuning* phase. In this section, we introduce the pre-training Bert model.

Bert has pre-trained word embeddings already for more than 70 different languages based on unlabeled plain texts. The vocabulary of Bert for the English language is the WordPiece vocabulary with 30'000 words that have been pre-trained by texts from the BooksCorpus and English Wikipedia. Bert's vocabulary consists of words and subwords. Its WordPiece tokenizer splits words like *playing* into *play* and *##ing* (Jindrich, 2019; TensorFlow, 2021; Wikipedia, 2021a).

Bert pre-trains contextual, more exactly, *deeply bidirectional* representations: While reading sentences, the representation of a particular (sub-)word depends on its left- and on its right context (referring to *bidirectional*) combined and not independently

[2]The cost function has been translated into cross-entropy loss function

from each other (referring to *deep*). For the exact same word, a different left- and right-context might produce a different word embedding. That means the same word can adopt different meanings depending on its context (e.g., The word *bank* as a seat or as money deposit) (Google, 2020). This deep bidirectionality is enabled by employing the *masked language model* (MLM) objective, referred to as the first pre-training task: Given a sentence, some words are randomly masked, and the objective is to predict the masked words. However, for Bert, the masking task has been slightly extended: During pre-training, Bert's task is to predict randomly chosen words (15%) within a sentence such that 80% from them are masked, 10% are replaced by another word, and 10% keep the original word. This extended masking procedure ensures that Bert optimizes representations not only for masked words but also for non-masked words by predicting if a word is correct or not. Bert will not be informed which of the words in the sentence are the randomly chosen ones and so under prediction examination; this leads to the optimization of the embeddings for each word token. Besides the MLM pre-training task, Bert also makes *next sentence predictions* (NSP), referred to as the second pre-training task. While pre-training, Bert gets fed by sequences that are single sentences or pairs of sentences. With pairs of sentences, Bert's second pre-training task consists in predicting if the second sentence is the correct next sentence of the first one. Half of the pairs have the correct next sentences. The purpose of that pre-training task is to teach Bert to understand sentence relationships, which is useful for downstream NLP tasks such as question answering. Bert, therefore, performs during pre-training in complete two tasks: the MLM and the NSP, while minimizing their combined loss function (Horev, 2018; Devlin et al., 2019; Montantes, 2021).

Let's denote an embedded sequence of size $E \leq 512$ as $T = (C, T_1, T_2, ..., T_{E-1})$. The C and T_i are H-sized hidden vectors of real numbers. The H is the dimension of the embeddings (see section 4.5.1). The C represents the whole sequence and is used for the NSP classification and 15% of T_i are used for the MLM classification. Let N be the total number of input sequences. Training the MLM classifier can be expressed with the cross-entropy loss function as follows (Horev, 2018; Devlin et al., 2019; Montantes, 2021)[3]:

$$L_{MLM} = \arg\min_{W} \left[-\frac{1}{N} \sum_{n=1}^{N} \sum_{i=1}^{E} y_{ni} M_{ni} \log(\text{softmax}(T_{ni} \cdot W_{MLM})) \right] \tag{2.6}$$

Where $M_{ni} = 1$ if T_{ni} belongs to the 15% of picked out tokens for MLM and $M_{ni} = 0$ otherwise. The W_{MLM} is a task-specific weight matrix of size $V \times H$ where V is the size of Bert's (sub-)word token vocabulary. The softmax$(T_{ni} \cdot W)$ predicts a probability distribution over the V (sub-)words for being the token T_{ni}. The y_{ni} is a V-sized one-hot vector representing the actual (sub-)word token. The weights W are permanently updated such that the difference between softmax$(T_{ni} \cdot W)$ and y_{ni} is minimized. Training the NSP classifier can be similarly expressed with the cross-entropy loss function as follows (Horev, 2018; Devlin et al., 2019; Montantes, 2021)[4]:

$$L_{NSP} = \arg\min_{W} \left[-\frac{1}{N} \sum_{n=1}^{N} y_n \log(\text{softmax}(C_n \cdot W_{NSP})) \right] \tag{2.7}$$

The W_{NSP} is the task-specific weight matrix of size $2 \times H$. The softmax$(C_n \cdot W)$ predicts the probability over the classes (*is next sentence* or *is not next sentence*) and the y_n

[3]The function has been computed based on the given references

[4]The function has been computed based on the given references

here represents the actual class as a 2-sized one-hot vector. Likewise, the weights W are permanently updated such that the difference between softmax($C_n \cdot W$) and y_n is minimized. The overall training procedure of the pre-training Bert model consists in training the NSP and MLM classifier together, minimizing their combined cross-entropy loss (Montantes, 2021; Devlin et al., 2019):

$$L_{NSP} + L_{MLM} \tag{2.8}$$

Bert's pre-training model architecture is a Transformer encoder with a classifier layer on the top. The original Transformer architecture has an encoder and a decoder part (Vaswani et al., 2017). Bert has the advantage to pre-train different embeddings for the same word, depending on the word's context, by working in a deep bidirectional manner. This makes the pre-training Bert model very accurate. Due to its complex Transformer architecture and bidirectionality, however, Bert is highly computationally expensive (this can be alleviated by training on TPU or GPU instead of CPU) (Devlin et al., 2019; Google, 2020; Khan, 2019).

2.1 Text classification algorithms

2.1.1 Logistic Regression Classification (logit, MaxEnt)

Logistic Regression (Cox, 1958) is a classification algorithm that predicts the probability estimation of data for several possible class labels (e.g., 0 and 1) through an optimal hyperplane that is determined through the sum of all training hyperplanes. For example, it predicts that a given data will be classified as 1 with a probability of 65% and classified as 0 with a probability of 35%.

Let's have a dataset $\{x_i, y_i\} \in D$ ready, where each data i is supplied by a V-sized feature vector x_i of real numbers (which are the TF-IDF scores in our case) and by a binary class label y_i. The class label y_i is either 1 or -1. The dataset D is divided into a training set D_{train} of size n and into a testing set D_{test} of size m.

Geometrically speaking, during the training procedure we aim at fitting a line or a plane, also called the decision boundary, that linearly classifies the training dataset as accurately as possible into the classes. A trained decision boundary hyperplane can be defined as follows (Pothabattula, 2019; Banerjee, 2020):

$$\left(\sum_{i=1}^{n} \sum_{j=1}^{V} w_{ij} \cdot x_{ij} \right) + b \tag{2.9}$$

Where the w are the learned weights, the x are the features, and b is a constant representing the bias. The w and b of the hyperplane have to be computed while training on the training set D_{train}. By feeding the trained hyperplane into the softmax function, we obtain a probability distribution over the classes. For binary classification problems, we can use the sigmoid function instead, which outputs a single probability value. Suppose we finished training a Logistic Regression classifier on our training data D_{train} and we want to predict for the i-th testing data $\{x_i, y_i\} \in D_{test}$ the probability for being classified as 1. This can be represented as follows (Google, 2021):

$$\begin{aligned} \text{Classifier}(x_i) &= \text{sigmoid}(w \cdot x_i + b) \\ &= \frac{1}{1 + e^{-(w \cdot x_i + b)}} \\ &= P(y_i = 1) \end{aligned} \tag{2.10}$$

We evaluate the classifier's performance by comparing the predicted class probability $P(y_i = 1)$ with the actual class probability y_i given by D_{test}.

For a given training data $\{x_i, y_i\} \in D_{train}$, if y_i is +1 and $w \cdot x_i > 0$, or if y_i is -1 and $w \cdot x_i < 0$, then x_i is on the correct class-side. If y_i is +1 and $w \cdot x_i < 0$ or if y_i is -1 and $w \cdot x_i > 0$, then the x_i has been separated by the plane into the wrong class-side. We therefore want to choose w (and b) to build the hyperplane such that the most possible training data are correctly classified (Banerjee, 2020):

$$\arg\max_{w,b} \sum_{i=0}^{n} y_i(w \cdot x_i) + b \tag{2.11}$$

In other words, formula 2.11 expresses the training procedure for the Logistic Regression classifier. It can be also translated into misclassifying the least possible of training data by using the logistic loss function and the regularization term $\lambda \cdot \|w\|$ (Pothabattula, 2019):

$$\arg\min_{w,b} \sum_{i=0}^{n} \log\left(1 + e^{-(y_i(w \cdot x_i)+b)}\right) + \lambda \cdot \|w\| \tag{2.12}$$

The $\|w\|$ is the weight vector distance describing the regularizer or penalty, and is used as hyperparameter to be set as $L1$ or $L2$ regularisation (see section 4.5.1). The regularisation avoids overfitting by penalizing some outlier weight values, in particular large weight values. For that, we need to calculate the weight vector distance either through the Euclidean norm for the $L2$ regularisation or through the Manhattan norm for the $L1$ regularisation. The $L2$ regularisation norm for w, denoted as $\|w\|_2$, is defined as the sum of the square weights as follows:

$$\|w\|_2 = \sum_{i=0}^{n} w_i^2 \tag{2.13}$$

For faster computation we can remove the square root function from the Euclidean norm. The $L1$ regularisation norm for w, denoted as $\|w\|_1$, is defined as the sum of the absolute weights as follows:

$$\|w\|_1 = \sum_{i=0}^{n} |w_i| \tag{2.14}$$

The advantage of applying $L1$ is the sparsity. The weight vector w is sparse when most of its values are zero. With $L1$ the weight values that are less important become zero, while with $L2$ they become smaller or zero. Therefore, $L1$ is advantageous for feature selection in the case of a large set of features. The λ is the regularization strength and is a hyperparameter too; we can also refer to it as the $alpha$ or C parameter (see section 4.5.1). The C is more exactly the inverse regularization strength, i.e. $C = 1/\lambda = 1/alpha$. The regularization strength hyperparameter controls, as its name suggests, the strength of the regularization. The lower the λ value (or the higher the C parameter value), the lower the training error but, the higher the generalization error (overfitting). The higher the λ value (or, the lower the C parameter

value), the higher the training error (underfitting) but, the lower the generalization error. Also, the higher the λ value, the more sparse becomes the weight vector w when using $L1$ (Paul, 2018; Melcher, 2018; Tyagi, 2021).

Logistic Regression classifier is a simple algorithm as it simply fits the sum of all training hyperplanes. This makes it quick in classifying new data. Also, the learned weights directly give information about the importance of the features. Its quite simple architecture however makes it difficult to compute more complex relationships. Logistic Regression assumes linearity of the training data which is not always guaranteed. This assumption in turn can be eased through regularization which also helps against overfitting. It can tend to overfitting namely when having lots of features but few training data (miyaRanjanRout, 2020; Grover, n.d.).

2.1.2 Random Forest Classification

Random Forest (Ho, 1995; Breiman, 2001) is an ensemble classification (or regression) algorithm predicated on the *bootstrap aggregating* or *bagging* and *feature-bagging* technique. The class label of a data is predicted by training a range of learning algorithms on bootstrapped training sets while using only a subset of the features, aggregating the class predictions obtained from each learning algorithm and finally returning the ultimate prediction through majority class voting (or by averaging for regression). The learning algorithms here are the decision trees.

Given a dataset $\{x_i, y_i\} \in D$, where each data i is supplied by a V-sized feature vector x_i of real numbers and by a binary class label y_i. The dataset D is divided into a training set D_{train} of size N and a testing set D_{test} of size Q.

Suppose we finished training a Random Forest model on our training data D_{train} and we want to predict the class label of the i-th testing data $\{x_i, y_i\} \in D_{test}$. The Random Forest classifier can be then expressed as follows (Wikipedia, 2021c; Hastie, Tibshirani, and Friedman, 2001):

$$\begin{gathered}\text{Classifier}(x_i) = \text{majority vote over } T_b(x_i) = \widehat{y_i} \\ \text{where } b = 1, ..., B\end{gathered} \tag{2.15}$$

The T_b represents a trained decision tree in Random Forest, and B is the number of decision trees. We evaluate the classifier's performance by comparing the predicted class label $\widehat{y_i}$ with the actual class label y_i given by D_{test}.

Bootstrap aggregating or *bagging* yields B new training sets D_b of same size n by randomly sampling with replacement from the training dataset D_{train}, i.e. some training data from D_{train} may appear multiple times in a training set D_b. In Random Forest, the training sets D_b are bootstrapped: The new training sets D_b are of the same size as the original training dataset D_{train}, i.e., $n = N$, and for large N the sets contain around 63.2% of unique training samples whereas the other samples are duplicates. Then, on each training set D_b is trained a proper decision tree T_b (Wikipedia, 2021b).

A decision tree consists of nodes and branches. A node of a trained decision tree represents a feature variable. Each node is split by if-branches into subtrees. An if-branch accepts a specific possible feature value or a range of possible feature values that the actual feature value has to match with. In Random Forest, a node is partitioned into two subtrees where the if-branch conditions are determined through threshold values (i.e., higher or lower than some value). Random Forest chooses the next feature node such that it divides the dataset into two parts of the possible equal size. A terminal node of a completely trained decision tree represents a classification outcome (Breiman et al., 1984; Hastie, Tibshirani, and Friedman, 2001).

The training procedure of a Random Forest decision tree differs from the general decision tree learning algorithm: At each node split, (i) m features variables are randomly selected from the V features variables, where $m \leq V$. Typically $m = \sqrt{V}$ for classification problems. This process can be referred to as *feature bagging* or *random subspace method*. The next steps similarly occur in the general decision tree learning algorithm, which consists in (ii) choosing the best feature variable among the m feature variables as a node and (iii) splitting the node into two children nodes. These three steps (i, ii and iii) are repeated at each node split until reaching the minimum node size of the decision tree. The size of a node describes the number of training data associated with that node. The terminal node of a tree with the smallest node size is referred to as the minimum node size. Algorithm 1 below describes the training procedure of the Random Forest model (Wikipedia, 2021c; Hastie, Tibshirani, and Friedman, 2001):

Algorithm 1 Training procedure of a Random Forest model

Require: Training data-set D_{train},
Ensure: Trained Random Forest model $\{T_b\}_1^B$

for b = 1, ..., B:
    (a). Randomly sampling with replacement from
        the training data-set D_{train}, generating D_b
    (b). While training the decision tree T_b on D_b,
        recursively iterate over following steps at each
        node split until the minimum node size is attained.
        i. Choose randomly m feature variables from the V feature variables
        ii. Select the best feature variable among the m feature variables as node
        iii. Split node into two children nodes
end for
return $\{T_b\}_1^B$

Random Forest makes use of a set of decision trees trained on bootstrapped training samples, where each tree is trained with a different subset of features. This makes Random Forest less prone to overfitting and more adapted to training dataset with many features. In contrast, rare training data are less well represented. Also, building a Random Forest is complex and computationally expensive, but it tends to output more accurate classification results than Logistic Regression, for example, (Team, 2020; Pradhan, Mamgain, and Colleen Farrelly Colleen Farrelly, 2020).

2.1.3 Linear Support Vector Classification

The Linear Support Vector classifier (LSVC) is a binary classification algorithm of Support Vector Machines (Cortes and Vapnik, 1995) which predicts the class label of a data through the optimal hyperplane that is determined through the best margin.

Let's have a dataset $\{x_i, y_i\} \in D$ ready, where each data i is supplied by a V-sized feature vector x_i of real numbers and by a binary class label y_i. The class label y_i is either 1 or -1. The dataset D is divided into a training set D_{train} of size n and a testing set D_{test} of size m.

While training we aim at fitting a hyperplane (see section 2.1.1). Let's define the trained optimal hyperplane classifier for LSVC as follows:

$$w_0 \cdot x + b_0 = 0 \tag{2.16}$$

Suppose we finished training a Linear Support Vector classifier on our training data D_{train} and we want to predict the class label of the i-th testing data $\{x_i, y_i\} \in D_{test}$. This can be represented as follows (Cortes and Vapnik, 1995; Manning, Raghavan, and Schütze, 2008a):

$$\text{Classifier}(x_i) = \text{sgn}(w_0 \cdot x_i + b_0) = \widehat{y_i} \tag{2.17}$$

The sign function sgn predicts the class label of the testing data $x_i \in D_{test}$ as 1 if $w \cdot x_i + b \geq 0$ and as -1 if $w \cdot x_i + b \leq 0$. We evaluate the classifier's performance by comparing the predicted class label $\widehat{y_i}$ with the actual class label y_i given by D_{test}.

Suppose the training dataset D_{train} is perfectly linearly separable. The optimal hyperplane divides the training dataset into two classes such that the distance between the nearest training data points x_i from either class-side and the optimal hyperplane is at maximum. These nearest training data points we call *support vectors*. The optimal hyperplane is computed by fitting two hyperplanes that are parallel to each other, and the optimal hyperplane lies in the middle between them. We call the region between the two parallel hyperplanes as margin. The support vectors lie within the two parallel hyperplanes, and it is them who determine the margin. The goal is, therefore, to determine the support vectors such that the margin is maximized. The two parallel hyperplanes can be expressed by following equations:

$$\begin{aligned} w \cdot x_i + b &= 1 \\ w \cdot x_i + b &= -1 \end{aligned} \tag{2.18}$$

For the i-th training data point $\{x_i, y_i\} \in D_{train}$, if y_i is +1 and $w \cdot x_i \geq 1$, or if y_i is -1 and $w \cdot x_i \leq -1$, then the training data point x_i is on the correct class-side, outside from the margin. This can be summarized for all training data points by a single condition as follows:

$$\forall_i y_i(w \cdot x_i + b) \geq 1 \tag{2.19}$$

The distance between the two parallel hyperplanes, which determine the margin, is defined through the optimal hyperplane as follows:

$$\frac{2}{\|w_0\|} \tag{2.20}$$

We choose w (and b) to build the optimal hyperplane such that the distance between the two parallel hyperplane, or the margin, is maximized and all training data are

correctly classified, outside from the margin:

$$\begin{gathered} \arg\max_{w,b} \frac{1}{\|w_0\|_2} = \\ \arg\min_{w,b} \|w_0\|_2 = \\ \arg\min_{w,b} w_0 w_0 \\ \text{s.t. } \forall_i \, y_i(w_0 x_i + b_0) \geq 1 \end{gathered} \tag{2.21}$$

We applied the Euclidean norm respectively the $L2$ norm for w_0, denoted as $\|w_0\|_2$, to compute the distance. Formula 2.21 represents the training procedure of the Linear Support Vector classifier (Cortes and Vapnik, 1995; Schölkopf and Smola, 2002).

Now suppose we allow the training dataset not to get perfectly linearly separated and allow some training errors. The optimal hyperplane is therefore said to be constructed under soft constraints. We want to minimize the training errors represented by so-called *slack variables* ξ_i. We re-write formula 2.21 with soft constraints accordingly (Cortes and Vapnik, 1995; Sontag, 2014):

$$\begin{gathered} \arg\min_{w,b,\xi} \|w_0\| + \frac{1}{\lambda} \sum_{i=1}^{n} \xi_i \\ \text{s.t. } \forall_i \, y_i(w_0 x_i + b_0) \geq 1 - \xi_i \\ \forall_i \, \xi \geq 0 \end{gathered} \tag{2.22}$$

The training formula with soft constraints 2.22 can also be translated into (hinge) loss function and regularization term $\lambda \|w_0\|$ (Wikipedia, 2021d; Sontag, 2014):

$$\begin{gathered} \arg\min_{w,b,\xi} \sum_{i=1}^{n} \xi_i + \lambda \|w_0\| = \\ \arg\min_{w,b,\xi} \sum_{i=1}^{n} \max[1 - y_i(w_0^T x_i + b_0), 0] + \lambda \|w_0\| \end{gathered} \tag{2.23}$$

The $\|w_0\|$ is the penalty hyperparameter (here we applied $L2$) and the λ is the regularization strength parameter; the latter is also referred to as as $alpha$ or C (see sections 2.1.1 and 4.5.1 for more details).

Linear Support Vector machine fits *the* optimal hyperplane by maxizimizing the margin defined by support vectors. It is more complex than Logistic Regression and gives more accurate classification results; this makes the algorithm however more prone to overfitting. It provides in turn a regularization parameter which helps against overfitting. It works well for rather small training dataset with lots of features. It is less adapted for large dataset since it would significantly extend the training time (Du, 2020; Ray, 2017a).

2.1.4 Naive Bayes classifier for multinomial models

The Multinomial Naive Bayes classifier (Maron, 1961) is a probabilistic classification algorithm. It predicts the class label of a given data by computing the probability of each label for that data and returning the class label with the highest probability. It relies on the naive assumption that the features are independent of each other and considers a multinomial probability distribution for each feature.

Let's have a dataset $\{x_i, y_i\} \in D$ ready, where each data i is supplied by a V-sized feature vector x_i of real numbers and by a binary class label y_i. The class label y_i is either 1 or -1. The dataset D is divided into a training set D_{train} of size n and into a testing set D_{test} of size m.

While training we aim at fitting a hyperplane (see section 2.1.1). Suppose we finished training a multinomial Naive Bayes classifier on our training data D_{train} and we want to predict the class label of the i-th testing data $\{x_i, y_i\} \in D_{test}$ (Weinberger, 2017):

$$\begin{aligned} \text{Classifier}(x_i) &= \text{sgn}(w \cdot x_i + b) \\ &= \arg\max_y \text{P}(y \mid x_i) = \widehat{y_i} \end{aligned} \tag{2.24}$$

We evaluate the classifier's performance by comparing the predicted class label $\widehat{y_i}$ with the actual class label y_i given by D_{test}. It predicts the class label with the highest probability. We compute the probability for each class label based on the Bayes' rule (Bayes, 1763) and with the feature independence principle as follows (Fan, 2018; Weinberger, 2017):

$$\text{P}(y \mid x_i) = \text{P}(y) \prod_{j=1}^{d} \text{P}(x_{ij} \mid y) \tag{2.25}$$

The $P(y|x_i)$ is called the posterior probability, $P(x_i|y)$ is the likelihood, $P(y)$ is the prior probability of the class label. By computing the the prior probability and the likelihood for each class label $y \in \{+1, -1\}$ and taking the difference, we determine the w and b of the decision boundary hyperplane (Rennie et al., 2003; Weinberger, 2017; Teufel, 2014):

$$\begin{aligned} w &= \sum_{j=1}^{d} (w_{+1j} - w_{-1j}) = \sum_{j=1}^{d} \Big(\log \text{P}(x_{ij} \mid +1) - \log \text{P}(x_{ij} \mid \text{-}1) \Big) \\ b &= b_{+1} - b_{-1} = \log \text{P}(+1) - \log \text{P}(-1) \end{aligned} \tag{2.26}$$

The class prior $P(y)$ is calculated by counting the number of training data x_i that are assigned to class label y, denoted as n_y, divided by the total amount of training data n (Teufel, 2014; Manning, Raghavan, and Schütze, 2008b):

$$\text{P}(y) = \frac{n_y}{n} \tag{2.27}$$

The conditional probability $\text{P}(x_{ij}|y)$ is computed by summing up the values of feature j through all training data belonging to class label y, denoted as F_{yj}, divided by the total sum of all values of each feature across all training data with class label y, i.e., $\sum_{j=1}^{d} F_{yj}$. To avoid a zero probability estimate for $\text{P}(x_{ij}|y)$ in case that a feature value x_{ij} doesn't appear in class y within the training data at all, we apply a regularizing method: We add pseudo-occurrences α_j to each feature j. Commonly $\alpha_j = 1$ for each feature, then we call the method Laplace smoothing (Rennie et al., 2003; Manning, Raghavan, and Schütze, 2008b):

$$\text{P}(x_{ij}|y) = \frac{F_{yj} + \alpha_j}{\sum_{j=1}^{d} (F_{yj} + \alpha_j)} = \frac{F_{yj} + 1}{\sum_{j=1}^{d} (F_{yj} + 1)} \tag{2.28}$$

Below we represent the training of the multinomial Naive Bayes model in form of an algorithm (Weinberger, 2017; Manning, Raghavan, and Schütze, 2008b):

Algorithm 2 Training procedure of a multinomial Naive Bayes model

Require: Training dataset $(x_i, y_i) \in D_{train}$,
Set of class labels Y=(+1,-1),
Ensure: Trained multinomial Naive Bayes model (w, b)

$n_{+1} \leftarrow$ number of training data $x_i \in D_{train}$
    that belong to class label $+1$
$n_{-1} \leftarrow$ number of training data $x_i \in D_{train}$
    that belong to class label -1
$n \leftarrow$ size of D_{train}
for each $y \in Y$ = (+1, -1):
    $P_1(y) \leftarrow n_y / n$
    for each $j = 1, ..., V$
        $F_{yj} \leftarrow$ sum of values of feature j within training data of class y
        $P_2(x_{ij}|y) \leftarrow \dfrac{F_{yj} + 1}{\sum(F_{yj} + 1)}$
$w \leftarrow \sum_{j=1}^{d} \Big(\log P_2(x_{ij} \mid +1) - \log P_2(x_{ij} \mid -1) \Big)$
$b \leftarrow \log P_1(+1) - \log P_1(-1)$
return (w, b)

In Naive Bayes, the features are uncorrelated. If the features of our training data are highly correlated between them, then Naive Bayes would achieve poor classification results. The algorithm gives, however satisfying classifying results even though the independence assumption doesn't perfectly hold. This independence between the features makes the algorithm also simple and quick in training (Ray, 2017b).

2.1.5 FastText as a text classification algorithm

FastText (Bojanowski et al., 2016; Joulin et al., 2016) is a method that works as word embedding (i.e. text representation) algorithm or as text classification algorithm. In this section we are going to introduce the FastText for text classification.

In FastText, a text document is represented as a bag of n-gram words. The n-gram words allow to consider the word order and are used as features for that document. These features are embedded, and the FastText classifier takes the average of these feature embeddings to represent the whole text document as a single embedding. The text document embedding, also called the hidden layer, is then fed to a linear classifier. The FastText classifier predicts the probability estimation of several possible class labels (e.g., 1 and 0) for unseen data. For example, it predicts that a given data will be classified as 1 with a probability of 65% and classified as 0 with a probability of 35%.

Let's have a dataset of text documents D ready, where each data i is supplied by a normalized bag of V n-gram word features x_i and a C-sized one-hot vector y_i representing the class label. The dataset D is divided into a training set D_{train} of size n and testing set D_{test} of size m.

While training, we aim at fitting a hyperplane (see section 2.1.1). Suppose we finished training the FastText classifier on our training data D_{train} and we want to predict for the i-th testing data $\{x_i, y_i\} \in D_{test}$ the probability distribution over the

C class labels. The trained FastText classifier can be represented as follows (Joulin et al., 2016):

$$\text{Classifier}(x_i) = \text{softmax}(BA \cdot x_i + b) = \widehat{y_i} \tag{2.29}$$

We evaluate the classifier's performance by comparing the predicted probability distribution over the classes $\widehat{y_i}$ with the actual probability distribution y_i given by D_{test}. The A and B are the weight matrices. A is a look-up table over the feature embeddings of size $V \times H$ where H is the dimension hyperparameter of the embeddings (see section 4.5.1). B is a linear output transformation matrix of size $H \times C$ (Bhattacharjee, 2018; Google, nd). For each of the V features from x_i, we read from the look-up table A to match the corresponding embedding to it. Then, these matched feature embeddings are averaged to produce a single text document embedding a_i, also called the hidden variable (Bhattacharjee, 2018):

$$a_i = \frac{1}{V} \sum_{j=1}^{V} A \cdot x_{ij} \tag{2.30}$$

When training on $\{x_i, y_i\} \in D_{train}$, we want to choose the weights A and B and the bias b to build the hyperplane classifer while minimizing as much as possible the training errors. The training procedure of the FastText classifier can be expressed by following cross-entropy loss function (Joulin et al., 2016; Bhattacharjee, 2018):

$$\arg\min_{B,A,b} \left[-\frac{1}{n} \sum_{i=1}^{n} y_i \log(\text{softmax}(BAx_i + b)) \right] \tag{2.31}$$

While training, the weights A and B are repetitively adjusted such that the difference between the two predicted probability distribution $\text{softmax}(BAx_i)$ and the actual probability distribution y_i is minimized.

The FastText classifier is a simple neural network with just one hidden layer and a linear classifier on the top (Joulin et al., 2016; Bhattacharjee, 2018). Its simple architecture makes it faster to train than other deep learning classifiers. Deep learning classifiers in general have the advantage of performing feature engineering by themselves compared to linear classifiers such as Logistic Regression (Shchutskaya, 2018). The FastText classifier cleverly takes word order into account in an computationally economically manner by using bag of n-gram words as features (Joulin et al., 2016). Also the mapping of the n-grams is quick through the so-called hashing trick (Weinberger et al., 2009). When having many class labels or an imbalanced dataset, the FastText classifier reduces the complexity by using the hierarchical softmax and the Huffman tree algorithm respectively (Joulin et al., 2016). The simplicity and quickness of the FastText classifier makes it handle easily large dataset with many class labels (Joulin et al., 2016); however, this also makes it less accurate than other more complex, e.g. Transformer-based, classifier.

2.1.6 Bert as text classification algorithm (the fine-tuned Bert model)

The Bidirection Encoder Representations from Transformers (BERT) (Devlin et al., 2019; Vaswani et al., 2017) is a method that generates word embeddings in the *pre-training* phase as well as performs downstream natural language processing (NLP) tasks such as text classification in the *fine-tuning* phase.

With the word embeddings provided by the pre-training Bert model, we can keep them unchanged as they are and employ them directly to downstream tasks, which is the *feature-based* approach. Or, while training the downstream task with our labeled training dataset, we optimize the pre-trained word embeddings which is the *fine-tuning* approach. Fine-tuning makes the word embeddings more adapted to our specific downstream task (Peters, Ruder, and Smith, 2019). While fine-tuning, only the pre-trained embeddings of words that also appear in our training data are actually updated through backpropagation (Jindrich, 2019). When Bert encounters a word from our labeled training data that doesn't appear in its pre-trained vocabulary, the WordPiece tokenizer splits the word such that the generated subwords occur in its vocabulary (Noe, 2020). However, there exist methods to add new words to the vocabulary as well (Guillou, 2021). Bert can be used for both approaches (Peters, Ruder, and Smith, 2019), but in this section, we describe Bert's fine-tuning approach.

At the beginning of the training, the fine-tuning Bert model is initialized with the vocabulary of pre-trained embeddings. While training, the fine-tuning Bert model is fed by sequences that are single sentences or pairs of sentences from the training data. Considering a classification task as downstream task, let's denote an embedded sequence of size $E \leq 512$ as $T = (C, T_1, T_2, ..., T_{E-1})$. The C and T_i are H-sized hidden vectors of real numbers. The H is the dimension of the embeddings (see section 4.5.1). The C represents the whole sequence and is used for the classification task. Let N be the total number of input sequences. While training, the fine-tuning Bert model aims at minimizing the cross-entropy loss as follows (Devlin et al., 2019; Montantes, 2021)[5]:

$$\arg\min_{W} \left[-\frac{1}{N} \sum_{n=1}^{N} y_n \log(\text{softmax}(C_n \cdot W)) \right] \tag{2.32}$$

The W is the task-specific weight matrix of size $K \times H$ where K is the number of class labels. The softmax$(C_n \cdot W)$ predicts the probability over the K classes, and the y_n here represents the actual class as a K-sized one-hot vector. The weights W are permanently updated such that the difference between softmax$(C_n \cdot W)$ and y_n is minimized.

Bert's fine-tuning model architecture is a Transformer encoder with a classifier layer on the top (like Bert's pre-training model). The original Transformer architecture has an encoder and a decoder part (Vaswani et al., 2017). The fine-tuning Bert adapts the pre-trained embeddings to the specific downstream task while training on the training data. This makes Bert's downstream task very accurate. The fine-tuning Bert model is computationally inexpensive compared to pre-training Bert (Devlin et al., 2019; Google, 2020).

[5]The standard classification loss has been translated into a cross-entropy loss function

Chapter 3

Method

This chapter introduces the significant steps in this thesis work: Data collection, data labeling, fine-tuning, and language-related steps. The data collection and labeling processes correlate with each other. The labeling consists of heuristic rules based on an intuitive analysis of the first labeled data provided by *Softcom*. The rules rely a lot on the data structure; hence, the data must be crawled such that their structure is accessible to the rules. Irregularity within the textual data and their source code can make these steps challenging. Fine-tuning the models' hyperparameters is crucial for modeling a successful prediction tool. A multilingual dataset inevitably brings up the question of how much the language of a dataset has an impact on the classification results.

3.1 Data collection and labeling

To build an accurate prediction model, we need a large amount of labeled data. *Softcom* kindly provided us with a set of first labeled data along with label criteria. These data have been labeled in two steps (i.e., *yes-no* or *yes-yes*), first by an employee and then by a group of experts. To collect more labeled data, we crawled 62'302 textual data from `simap.ch` and labeled them through heuristic rules that are based on our intuitive analysis of the first labeled data with criteria provided by *Softcom*.

The data have been crawled with Python's library *BeautifulSoup*. The crawled data have been saved into `json` format. We crawled data that have been published within a certain period that is determined by a starting and ending date.

The raw data have to be downloaded and saved into a most possible evenly organized form such that the heuristic rules of the labeling algorithm can perform on them adequately. In other words, the labeling algorithm and the raw data's organization have to conform with each other. The textual data on `simap.ch` are multilingual, among the data and sometimes within the data, and their content is divided into only passably equable sections. Moreover, their source-code configuration can differ from one data to another and is in part irregular within a single data.

The heuristic rules of the labeling algorithm consist in detecting particular words or subwords with their particular neighboring (sub-)words within precise areas of the data. These areas are notably accessed through the section titles. The heuristic rules take into account the different languages of the dataset.

3.2 Fine-tuning and multilingualism

Once we have collected a large set of labeled data, we proceed in two main folds. First, we apply on the labeled data different classification models while tuning their hyperparameters and compare the adjusted models with each other. Each model

is composed of a text representation algorithm and a text classification algorithm. Second, we compare the performance of the tuned models on a multilingual level: We compare between a language-*independent* and language-*dependent* classification methodology. Below we demonstrate more in detail this working procedure in the form of an algorithm that we call the Multilingual Text Classification Problem algorithm. The outputs of this algorithm are the results of both folds. It also contains a second algorithm, the Run Model algorithm, that describes for a particular model the translation from text into features, the training and testing steps, and outputs the classification results.

We composed overall six different classification models, each consisting of a text representation algorithm and a text classification algorithm. The text representation algorithm TF-IDF has been combined with the Logistic Regression classifier, Linear Support Vector classifier, Random Forest classifier, and the Naive Bayes classifier for multinomial models, respectively. The FastText's word embeddings have been combined with the FastText classifier, and Bert's word embeddings have been combined with Bert's fine-tuning model. When classifying language *independently*, the model has been trained on a multilingual dataset. A language-*dependent* classification methodology, however, implies a model that has been trained on a dataset of a single language.

Algorithm 3 Multilingual Text Classification Problem

Input: Set of (unprocessed) labeled simap projects *labeled_data*, Set of models *models* each consisting of a text representation algorithm *repralgo* and text classification algorithm *classalgo*

Output: Tuning results (model, best parameters, results) and Multilingualism results (model, language, language-dependent results, language-independent results)

```
1:  procedure MTCP(labeled_data, models)
2:      labeled_data ← Pre-process labeled_data
3:              *** handle first fold***
4:      models_argscombs_set ← Define tuning arguments for models
5:      for model, argscombs in models_argscombs_set do
6:          trainset_indicies, validationset_indicies ← split labeled_data into train
7:              and validation set indicies
8:          for argscomb in argscombs do
9:              result ← RUN_MODEL(model, argscomb, labeled_data,
10:                     trainset_indicies, validationset_indicies)
11:             argscomb_result_set ← Append argscomb, result
12:         best_argscomb ← Extract arguments with best result
13:                                     from argscomb_result_set
14:         trainset_indicies, validationset_indicies, testset_indicies ←
15:                 split labeled_data into train, validation and test set indicies
16:         final_result ← RUN_MODEL(model, best_argscomb, labeled_data,
17:                 trainset_indicies, validationset_indicies, testset_indicies)
18:         tuning_results ← Append model, best_argscomb, final_result
19:             *** handle second fold***
20:     languages ← Define set of languages
21:     for model, best_argscomb, _ in tuning_results do
22:         for language in languages do
23:             for k times do
24:                 dep_trainset_indicies, testset_indicies ← Take from labeled_data
25:                     only samples in language. Split these samples into train and
26:                     test set indicies.
27:                 indep_trainset_indicies ← Append to dep_trainset_indicies all
28:                     indicies of samples from labeled_data that belong to all
29:                     languages except of language.
30:                 dep_result ← Add RUN_MODEL(model, best_argscomb,
31:                     labeled_data, dep_trainset_indicies, testset_indicies)
32:                 indep_result ← Add RUN_MODEL(model, best_argscomb,
33:                     labeled_data, indep_trainset_indicies, testset_indicies)
34:             multilingualism_results ← Append model, language, dep_result/k,
35:                 indep_result/k
36:     return tuning_results, multilingualism_results
37: end procedure
```

Algorithm 4 The Run Model algorithm

Input: Model *model* consisting of a text representation *repralgo* and a classification algorithm *classalgo*, Single combination of arguments *argscomb*, Set of (pre-processed) labeled data *labeled_data*, Train set indicies *trainset_indicies*, Validation set indicies, Test set indicies *testset_indicies* (if available)

Output: Metric result *result*

```
procedure RUN_MODEL(model, argscomb, labeled_data, trainset_indicies,
    validationset_indicies, testset_indicies)
    repralgo, classalgo ← model
    repralgo_args, classalgo_args ← argscomb
    features ← Transform labeled_data with repralgo and repralgo_args
    if testset_indicies is not available:
        trainset, validationset ← Split features into train and validation set with
            trainset_indicies and validationset_indicies
    else:
        trainset, validationset, testset ← Split features into train, validation
            and test set with trainset_indicies, validationset_indicies and
            testset_indicies
    trained_model ← Train classalgo with classalgo_args and trainset
    predictions_1 ← Evaluate trained_model with validationset based on
        optimal threshold
    if testset_indicies is available:
        predictions_2 ← Test trained_model with testset based on
            same optimal threshold
        result ← Compare predictions_2 with labels of testset by using metrics
    else:
        result ← Compare predictions_1 with labels of validationset by
            using metrics
    return result
end procedure
```

Chapter 4

Experiments

In this chapter, we show the classification results of the experiments. The experiments consist of tuning the classification models for classifying multilingual textual data and evaluate their performance based on two different language-related classification methodologies. We also introduce the properties of the dataset and the metrics that we used for the evaluation of the models. In the end, we present a detailed analysis of the classification results[1].

4.1 Methods

We shortly describe the algorithms that have been used for the experiments (see Chapter 3 for more details):

- *TF-IDF* is a text representation algorithm that represents each document as a vector. Each element of the vector stands for a vocabulary word and assigns a weight that represents the importance of that word within the document.
- *FastText* as a text representation algorithm is a machine learning algorithm that embeds a word as a sum of the word's character n-grams while considering the word's neighboring words during training.
- *Bert* as a text representation algorithm (pre-training Bert) is a Transformer-based machine learning algorithm that learns several embeddings for the exact same word depending on its context. For example, the word *bank* can be representing money deposit or a seat depending on the context.
- *Multinomial Naive Bayes* is a probabilistic classification algorithm that relies on the naive assumption that the features are independent of each other.
- *Random Forest* is an ensemble classification (or regression) algorithm where the class label of data is predicted by training a range of decision trees on bootstrapped training sets while using only a subset of the features.
- *Linear SVC* is a binary classification algorithm of Support Vector Machines that predicts the class label of data through *the* optimal hyperplane classifier that is determined through the best margin.
- *Logistic Regression* is a classification algorithm that predicts for a data a probability distribution over several possible class labels with the help of an optimal hyperplane that is determined through the sum of all training hyperplanes.

[1]The code and the datasets are available here: https://github.com/julul/softcom_simap

- *FastText* as the classifier is a machine learning classifier that takes word order into account by using a bag of n-gram words as features.
- *Bert* as a classifier (fine-tuning Bert) is a Transformer-based machine learning classifier that fine-tunes the vocabulary words while training for a specific classification task.

For the experiments, we combined the TF-IDF text representation algorithm with Multinomial Naive Bayes, Random Forest, Linear SVC, and Logistic Regression. The FastText for word embeddings is combined with the FastText classifier, and the Bert for word embeddings is combined with the Bert classifier. More exactly, we chose a lighter version of Bert, named DistilBert (Sanh et al., 2019) for faster computation with comparable performance.

4.2 Dataset properties

We collected a set of 62'302 labeled textual documents. These documents are written in five different languages: 38'524 have been detected as German (61.83%), 21'203 as French (34.03%), 1729 as Italian (2.78%), 841 as English (1.35%), and only 5 as Spanish (0.01%). Some documents, however, are actually written in multiple languages: A data is divided into sections, and sometimes, for example, the section titles are written in one language and the section contents in another. The section structuring can differ from one data to another, independently from the language. 57'316 documents have been labeled as not interesting (0 or *no*) (92%) and 4986 have been labeled as interesting (1 or *yes*) (8%). 4117 German samples are labeled as interesting (11%) and 34407 are labeled as not interesting (89%). 731 French samples are labeled as interesting (3%) and 20472 French samples are labeled as not interesting (97%). 64 Italian samples are labeled as interesting (4%) and 1665 Italian samples are labeled as not interesting (96%). 74 English samples are labeled as interesting (9%) and 767 English samples are labeled as not interesting (91%). 0 Spanish samples are labeled as interesting (0%) and 5 Spanish samples are labeled as not interesting (100%).

4.3 Experimental Setup:

While fine-tuning, the whole dataset is split up for all models except for Bert into 80% training set and 20% validation set; for the evaluation of Bert, the dataset is split up into 90% training set and 10% validation set to reduce the computational load of Bert (Kumar, 2021). Once the best hyperparameter values for each model are fixed, we re-split the dataset into a balanced training dataset of 7'976 samples (80%), a balanced validation dataset of 996 samples (10%) and a balanced testing dataset of 996 samples (10%). Each model is then trained, validated, and tested. Each model is tuned once, and we evaluate the tuned model's performance based on two different language-related classification methodologies once as well.

4.3.1 Fine-tuning the hyperparameters

Each model goes through a tuning process where we select the hyperparameter values achieving the best classification results. At each iteration during tuning, the dataset is split up into the same training and validation set. The validation set is reduced such that there are equally many data labeled as 0 (not interesting) as data

labeled as 1 (interesting). At each tuning iteration, we apply a different set of hyperparameter values, calculate the optimal decision threshold (i.e., the threshold with the highest f1-score in the precision-recall curve), and predict the labels of the validation set based on the calculated optimal decision threshold. The prediction results are evaluated through a range of different metrics. We choose the AUPRC metric as the tuning metric. For our classification problem, we consider the AUPRC metric as the most representative one for two main reasons: The AUPRC score measures the performance of a classification model independently of the chosen decision threshold and is more indicative of the performance in imbalanced datasets. We are interested in identifying the positive samples, which are underrepresented in our dataset and therefore the AUPRC is appropriate for our case. After the tuning procedure, with the hyperparameter values achieving the best AUPRC values, we train again each model with the balanced training dataset, validate and test and finally report the results.

4.3.2 Language-dependent task

After fine-tuning the hyperparameters, we evaluate the performance of the tuned models on two different language-related classification methodologies: *dependently* and *independently* of the language. For each tuned model and for each language among German, French, English, and Italian, we do the following steps five times, taking German as an example:

- Extract all samples in German from the dataset. Shuffle these German samples randomly and split them into 80% training and 20% validation sets. Then, reduce the validation set such that the class labels are balanced.
- Language-*dependent* set up: Train on the 80% German train set. Evaluate the 20% German validation set.
- Language-*independent* set up: Extract all other samples from the dataset, and train on 100% of Italian, 100% of French, 100% of English, **the same 80%** German set all together at once. Test on **the same 20%** German test set.

After the five runs, we compute the average classification results for both language-related approaches. For Bert, the dataset is split up into 90% training and 10% validation set for faster computation (Kumar, 2021).

4.4 Metrics

In this section, we give an insight into various metrics that we used in our experiments to evaluate the classification results.

4.4.1 Confusion Matrix

A confusion matrix visualizes the performance of a classification model by a table. Each column of the table represents the samples of the predicted label, whereas each row represents the samples of the actual label (or inversely). In our case, since we have two labels, the confusion matrix has two columns and rows, each for positive and negative samples. The positive samples which have been correctly classified by the model as positive are called *true positives* (TP). The positive samples which have been erroneously classified as negative are named *false negatives* (FN). The negative

samples which have been correctly labeled as negative belong to the *true negatives* (TN). Finally, the negative samples which have been wrongly marked as positive are referred to as *false positives* (FP). Table 4.1 demonstrates such a confusion matrix of a binary classification model.

		predicted (+)	**predicted** (−)
actual	(+)	TP	FP
	(−)	FN	TN

TABLE 4.1: Confusion matrix of a binary classification model. P = Positive. N = Negative. TP = True Positive. FN = False Negative. FP = False Positive. TN = True Negative

4.4.2 Accuracy

Accuracy gives information on how close are the classification model's predictions to the actual values. Accuracy is the ratio of the number of correct predictions to the total number of predictions.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4.1}$$

4.4.3 Precision

Precision, also known as Positive Predictive Value (PPV), tells how many of the samples that have been predicted as positive are truly positive and thus have been correctly classified.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.2}$$

4.4.4 Recall

Recall, also known as Sensitivity, True Positive Rate (TPR) or Hit Rate, tells how many of the samples that are actually positive have been correctly classified.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.3}$$

4.4.5 F1-score

F1-score measures the accuracy of a model, foregrounding the commonly less represented positive class. It is formulated as the harmonic mean of Precision and Recall (see sections 4.4.3 and 4.4.4). F1-score is a metric for expressing the ability of a model

to detect correctly positive samples. F1-score is a good choice when there is an imbalanced class distribution over the dataset (Koehrsen, 2018; Huilgol, 2019).

$$\text{F1-score} = \left(\frac{\text{Recall}^{-1} + \text{Precision}^{-1}}{2} \right)^{-1} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.4}$$

4.4.6 AUROC (or AUC) and the ROC curve

A ROC (Receiver Operating Characteristic) curve is a graph that presents performance results of a classification model based on different classification thresholds. The purpose of such a graph is to find the optimal classification threshold. The y-axis of the graph represents the True Positive Rate (TPR) and the x-axis represents the False Positive Rate (FPR). Any point on the curve represents the performance result, i.e. the ratio between TPR and FPR, at a certain threshold. The optimal classification threshold would maximize the TPR and minimize the FPR. It can be found by computing the Geometric Mean of TPR (also called Sensitivity) and 1-FPR (also called Specificity) at each threshold and picking out the one with the highest G-mean value.

The AUC (area under the curve) or more exactly the AUROC (area under the ROC curve) represents as its name suggests the area under the ROC-curve and is a metric to measure the prediction performance of the classification model on its whole, including all decision thresholds. The bigger the area, the higher the AUC score, the higher the ability of the classification model of classifying the actually positive samples as positive as well as classifying the actually negative samples as negative. The AUC score is helpful to compare the ROC curves of different classification models (e.g. one ROC curve represents the performance of Logistic Regression, another is for Random Forest).

4.4.7 AUPRC and the PR curve

A PR (Precision-Recall) curve is a graph that presents the performance results of a classification model based on different classification thresholds. The purpose of such a graph is to find the optimal classification threshold. The y-axis of the graph represents the Precision and the x-axis represents the Recall. Any point on the curve represents the performance result, i.e. the ratio between Precision and Recall, at a certain threshold. The optimal classification threshold would maximize the Precision as well as maximize the Recall, keeping them balanced. It can be found by computing the Geometric Mean of Precision and Recall or by computing the F1-score of Precision and Recall at each threshold and picking out the one with the highest G-mean value or F1-score respectively.

The AUPRC (area under the PR-curve) represents as its name suggests the area under the PR-curve and is a metric to measure the prediction performance of the classification model on its whole, including all decision thresholds. The bigger the area, the higher the AUPRC score, and the higher the ability of the model of classifying the actually positive samples as positive as well as correctly predicting a sample as positive. The AUPRC score is helpful to compare the PR curves of different classification model (e.g. one PR curve represents the performance of Logistic Regression, another is for Random Forest).

4.5 Results

4.5.1 Hyperparameter fine-tuning results

We present in table 4.2 the classification results after training with the best achieved hyperparameter values on a balanced dataset of 7'976 samples (80%), validating on a balanced dataset of 996 samples (10%) and testing on a balanced dataset of 996 samples (10%).

Model	**Metrics**					
	Accuracy	*Precision*	*Recall*	*F1- score*	*AUC*	*AUPRC*
Multinomial Naive Bayes	0.73	0.74	0.73	0.73	0.82	0.84
Linear SVC	0.85	0.83	0.86	0.85	0.92	0.92
Random Forest	0.85	0.85	0.85	0.85	0.93	0.94
FastText	0.87	0.87	0.88	0.88	0.95	0.95
Logistic Regression	0.90	0.88	0.92	0.90	0.96	0.97
DistilBert	0.93	0.95	0.91	0.93	0.98	0.98

TABLE 4.2: Classification results after training with the best achieved hyperparameter values on a balanced dataset (80%), validating on a balanced dataset (10%) and testing on a balanced dataset (10%).

The hyperparameters have been fine-tuned based on the optimal decision threshold (the threshold with the highest f1-score in the precision-recall curve) and according to the *AUPRC* score. The classification results overall tend to improve from the Multinomial Naive Bayes model to the DistilBert model. The *AUC* and *AUPRC* metrics achieve the best results among the metrics. Overall the results are good for each model, except for the Multinomial Naive Bayes model, notably from *Accuracy* to *F1-score*.

Below we show the values of the most common hyperparameters of the TF-IDF text representation algorithm after fine-tuning in table 4.3; those of the word embedding algorithms are shown in table 4.4 and those of the text classification algorithms are presented in table 4.5.

Model	**TF-IDF** parameters	
	min df	*max df*
MNB	0.001	0.85
RF	0.001	0.95
LR	0.001	0.9
LSVC	0.01	0.8

TABLE 4.3: Values of the most important hyperparameters of the TF-IDF algorithm after fine-tuning. MNB: Multinomial Naive Bayes. RF: Random Forest. LR: Logistic Regression. LSVC: Linear SVC.

Model	**Word Embeddings** parameters				
	dim	*minn*	*maxn*	*epoch*	*lr*
FT	50	2	6	2	0.07
DB	768	-	-	1	0.00004

TABLE 4.4: Values of the most common hyperparameters of the word embedding algorithms. FT: FastText. DB: DistilBert.

The *min df* and *max df* both represent the proportions of the documents. The purpose is to remove the vocabulary words that appear in less than *min* df $\cdot 100$ percent of all documents and those that appear in more than *max df* $\cdot 100$ percent of all documents. The *dim* stands for the dimension of the embeddings. The *minn* is the minimum size of the character n-grams, and *maxn* is the maximum size of the character n-grams. The *epoch* defines the number of times we loop over the training dataset. The *lr* stands for the learning rate, which controls the speed of updating the model's weights while training. Both, the epochs and learning rate, are at risk of overfitting.

Model	**Classifier** parameters			
	C	*penalty*	*epoch*	*lr*
LR	10	l1	-	-
LSVC	0.1	l2	-	-
FT	-	-	38	0.09
DB	-	-	4	0.00005

TABLE 4.5: Tuning results of most common hyperparameters of text classification algorithms. MNB: Multinomial Naive Bayes. RF: Random Forest. LR: Logistic Regression. LSVC: Linear SVC. FT: FastText. DB: DistilBert.

The *C* and the *penalty* have a regularizing role to prevent overfitting (See in sections 2.1.1 for more details).

4.5.2 Language-dependent results

After fine-tuning the models, we evaluate their performance on two different language-related classification methodologies (*dependently* and *independently* of the language) for each language. Figure 4.1 demonstrates the averaged AUPRC results of the two methodologies[2].

[2]The results might show overfitting tendencies

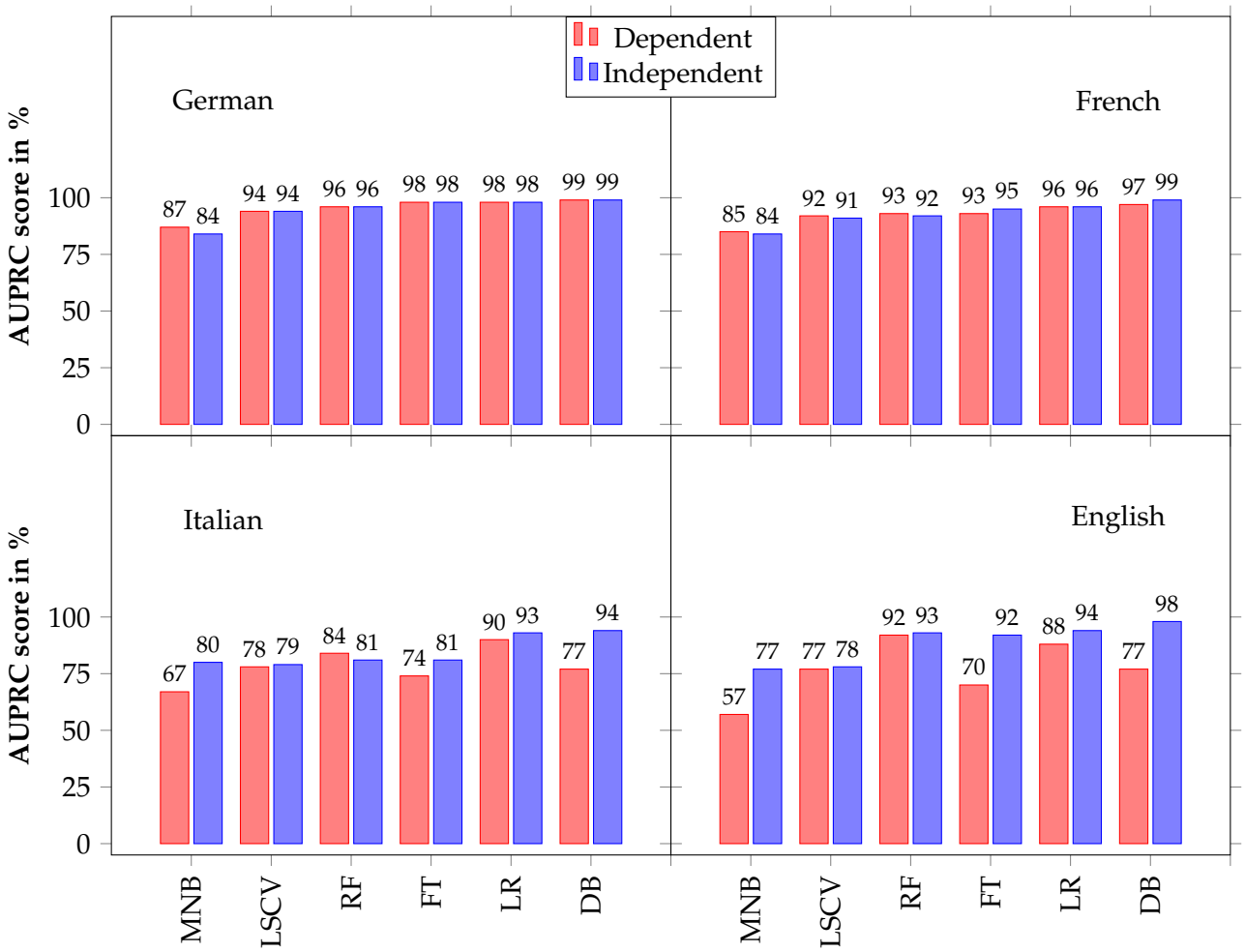

FIGURE 4.1: AUPRC scores after German-dependent and -independent, after French-dependent and -independent, after Italian-dependent and -independent, and after English-dependent and -independent classification. MNB: Multinomial Naive Bayes. LSCV: Linear SCV. RF: Random Forest. FT: FastText. LR: Logistic Regression. DB: DistilBert

4.5.3 Analysis of the results

Reading the fine-tuning results from the table 4.2, classification results overall tend to improve from the Multinomial Naive Bayes model to the DistilBert model. The *AUC* and *AUPRC* metrics achieve the best results among the metrics. The results of the other metrics (Accuracy, Precision, Recall, F1-score) tend to be overall close per model. Altogether the results are good (≥ 0.8) for each model, except for the Multinomial Naive Bayes model, notably from *Accuracy* to *F1-score*.

Bert obtained the best results. It produces several embeddings for the exact same word depending on the word's context that are fine-tuned while training for a specific classification task, which leads to very accurate embeddings for a particular task. Additionally, we applied the multilingual Bert, which learns cross-lingual word embeddings for several languages; the German, French, English, and Italian languages are included.

Logistic Regression and Linear SVC have similar training objectives and therefore are supposed to achieve similar classification results. Logistic Regression fits

more for simple rule-based classifications, whereas the Linear SVC better fits the classifications based on more complex connections. Our labeling algorithm is rule-based, which might explain why Logistic Regression achieves better results than Linear SVC.

The FastText performance is comparable with Bert's performance. The FastText model has a simple architecture compared to Bert, which usually leads to less accurate results. Deep learning algorithms, in general, tend to outperform other machine learning algorithms when a large amount of data is available due to its self-learning capabilities (Pham et al., 2021). Logistic Regression is competing with them in our classification problem, probably due to our rule-based labeled dataset.

Random Forest also achieves good classification results through the training of *multiple* decision trees. However, each decision tree trains on a bootstrapped and feature bagged training set: Only a subset of training samples are chosen. From which, only a subset of training features are considered while training a decision tree. These properties tend to prevent overfitting but are disadvantageous in case the features are highly correlated. The features of our classification problem have some correlation, which explains the low performance of Random Forest compared to FastText, Logistic Regression, Distilbert. Random Forest and Linear SVC have overall very close classification results.

The Naive Bayes for multinomial models is based on the naive assumption that the features are completely independent from each other. As with Random Forest, the low performance of the Naive Bayes model let us suggest that the features of our classification problem are correlated.

Reading the results of the two language-related methodologies from figure 4.1, following observations are summarized below:

- *German*: Among all models, there is no significant difference between classifying German samples in a dependently and independently manner, according to the AUPRC score. The Multinomial Naive Bayes model shows only 3% higher results for the language-dependent approach. The classification results of both approaches in German are good ($\geq$ 80%) for all models; the results improve from Multinomial Naive Bayes to DistilBert.

- *French*: Among all models, there is no significant difference between classifying French samples in a dependently and independently manner, according to the AUPRC score. Multinomial Naive Bayes, Linear SVC, and Random Forest are only 1% better in each language-dependent approach. FastText and DistilBert are better in the language-independent approach for only 2% each. Same as for German, the classification results of both approaches in French are suitable for all models; the results improve from Multinomial Naive Bayes to DistilBert.

- *Italian*: Multinomial Naive Bayes and Distilbert show a significant difference between classifying Italian samples in a dependently and independently manner, according to the AUPRC score, where the independent manner leads to better outcomes (13% and 17% of differences respectively). FastText too works better for 7% language-independently. Logistic Regression and the Linear SVC are also better in the language-independent approach, but for 3% and 1% only respectively, and the Random Forest works better language-dependently for 3% only. The Logistic Regression model performs well in both approaches. The DistilBert model performs well in the language-independent approach

and satisfying ($\geq$ 70% and $<$ 80%) in the language-dependent approach. Random Forest performs well in both approaches. FastText performs satisfying in the language-dependent approach and good in the language-independent approach. The Linear SVC performs satisfying in both approaches. The Multinomial Naive Bayes model performs poorly ($<$ 70%) in the language-dependent approach and just good (80%) in the language-independent approach.

- *English*: Multinomial Naive Bayes, FastText, and Distilbert show a significant difference between classifying English samples in a dependently and independently manner, according to the AUPRC score, where the independent manner achieves better results (20%, 22%, and 21% of differences respectively). Same for Logistic Regression with 6% better performance in the language-independently approach. Linear SCV and Random Forest work also better language-independently but only for 1% each. The Logistic Regression model performs well in both approaches. The DistilBert model performs well in the language-independent approach and satisfying ($\geq$ 70% and $<$ 80%) in the language-dependent approach. Random Forest performs well in both approaches. FastText performs satisfying in the language-dependent approach and good in the language-independent approach. The Linear SVC performs satisfying in both approaches. The Multinomial Naive Bayes model performs poorly ($<$ 70%) in the language-dependent approach and satisfying in the language-independent approach.

Recall that 61.83% of the samples are German, 34.03% are French, 2.78% are Italian, and 1.35% are English (0.01% are Spanish). 4117 German samples are labeled as interesting (11%), and 34407 are labeled as not interesting (89%). 731 French samples are labeled as interesting (3%), and 20472 French samples are labeled as not interesting (97%). 64 Italian samples are labeled as interesting (4%), and 1665 Italian samples are labeled as not interesting (96%). 74 English samples are labeled as interesting (9%), and 767 English samples are labeled as not interesting (91%).

The performance tendencies of the two language-related approaches in German are similar to those in French. Likewise, the performance results of the two language-related approaches in Italian adopt similar tendencies over the models as in English. There are more differences between both approaches in Italian and English than in German and French. It seems that languages that are less well represented in a dataset (Italian and English in our case) tend to show more differences between both approaches than languages that are well represented (German and French in our case). It seems evident the difference in performance tends to be higher between a model that learned from a dataset of 1.35% English samples (and been tested on 100% English samples) and another model which learned from a dataset of 100% English samples (and been tested on the same 100% English samples), than between a model that learned from a dataset of 62% English samples and a model that has learned from a dataset of 100% English samples.

In German and French, the performance improves from the Multinomial Naive Bayes model to the DistilBert model based on the same assumptions mentioned at the top of this section. E.g., the low performance of Multinomial Naive Bayes compared with the other models can be explained by its naive assumption that the features are completely independent of each other; the features of our classification problem, however, are correlated. The dependent approach in German and French achieves slightly better results for the Multinomial Naive Bayes model, probably also due to the independence principle; features of different languages may rely more on the correlation between them than the features of the same language. In

Italian and English, it is the inverse; the Multinomial Naive Bayes performs poorly in the language-dependent approach compared to the language-independent approach and other models in the language-dependent approach. The reason here might be the small training dataset in the language-dependent approach; the Multinomial Naive Bayes model probably needs particularly more training data than other models. Since it doesn't consider the correlations between the features, even though the features are actually correlated, the model needs more training data to achieve better performance. The training datasets in the language-dependent approach in Italian and English are very small. These small training datasets lower the performance results also for the other models but are less substantial than for Multinomial Naive Bayes model.

A model that has been trained on a large dataset but is tested on a language that is underrepresented in the training set clearly achieves low performance compared with a model that has been trained on a large dataset and is tested on a language that is well represented in the training set. However, a model trained on a training set of one language but of small size (and tested on that same language) seems to have a lower performance.

Probably a model that is trained on a large dataset of one language (and tested on that same language) would outperform a model that is trained on a comparable large dataset of different languages (and tested on the same set). This assumption is based on the results for German and French, where there is almost no difference between the two methodologies even though the training dataset in the language-dependent approach is still much smaller than the training dataset in the language-independent approach.

The language-independent approach works better when the training dataset of the language-dependent approach is too small, i.e., for languages that are underrepresented over the entire dataset.

Chapter 5

Conclusion

5.1 Summary

During the realization of the project within this master thesis, we went through several challenges: The data collection where we collected data from simap.ch and labeled it. The implementation of text representation and text classification algorithms, as well as other machine learning methods, evaluating the performance of the tuned classification models in two different aspects.

This master thesis conveys a comparison of six different classification models as well as a performance evaluation on a language-oriented level. We conclude that language-independent classifiers perform overall better than the language-dependent ones for underrepresented languages; this is probably due to their too small training dataset. Language-dependent classifiers with large training dataset might outperform the language-independent classifiers with training dataset of comparable size. Linear SVC, Random Forest, FastText, Logistic Regression and DistilBert are well-performing classification models, whereas Multinomial Naive Bayes achieves only satisfying performance results. DistilBert performs best.

A future work can be the collection of more labeled data with the help of more classification examples from *Softcom* or downloading more data from simap.ch to label them. More labeled data could be collected through existing weak supervision methods such as few-shot learning techniques (Ozsubasi, 2021).

Bibliography

Banerjee, Writuparna (2020). *Train/Test Complexity and Space Complexity of Logistic Regression*. https://levelup.gitconnected.com/train-test-complexity-and-space-complexity-of-logistic-regression-2cb3de762054.

Bayes, T. (1763). "An essay towards solving a problem in the doctrine of chances". In: *Phil. Trans. of the Royal Soc. of London* 53, pp. 370–418.

Bhattacharjee, Joydeep (2018). *fastText Quick Start Guide*. Packt Publishing Ltd.

Bojanowski, Piotr et al. (2016). *Enriching Word Vectors with Subword Information*. cite arxiv:1607.04606Comment: Accepted to TACL. The two first authors contributed equally. URL: http://arxiv.org/abs/1607.04606.

Breiman, L. et al. (1984). *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks.

Breiman, Leo (2001). "Random forests". In: *Machine learning* 45.1, pp. 5–32.

Cortes, C. and V. Vapnik (1995). "Support Vector Networks". In: *Machine Learning* 20, pp. 273–297. URL: http://image.diku.dk/imagecanon/material/cortes_vapnik95.pdf.

Cox, David R. (1958). "The Regression Analysis of Binary Sequences (with Discussion)". In: *J Roy Stat Soc B* 20, pp. 215–242.

Devlin, Jacob et al. (June 2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://www.aclweb.org/anthology/N19-1423.

Di, Wei, Anurag Bhardwaj, and Jianing Wei (2018). In: *Deep Learning Essentials*. Chap. 5, p. 141. ISBN: 9781785880360.

Du, Haitao (2020). *SVM vs Logistic Regression [duplicate]*. https://stats.stackexchange.com/questions/443351/svm-vs-logistic-regression.

Fan, Shuzhan (2018). *Understanding the mathematics behind Naive Bayes*. https://shuzhanfan.github.io/2018/06/understanding-mathematics-behind-naive-bayes/.

Google (2020). *What is Bert?* https://github.com/google-research/bert.

— (2021). *Logistic Regression: Calculating a Probability*. https://developers.google.com/machine-learning/crash-course/logistic-regression/calculating-a-probability.

— (nd). *Text Classification with fastText*. https://colab.research.google.com/github/NaiveNeuron/nlp-excercises/blob/master/tutorial2-fasttext/Text_Classification_fastText.ipynb.

Grover, Khushnuma (n.d.). *Advantages and Disadvantages of Logistic Regression*. https://iq.opengenus.org/advantages-and-disadvantages-of-logistic-regression/.

Guillou, Pierre (2021). *NLP | How to add a domain-specific vocabulary (new tokens) to a subword tokenizer already trained like BERT WordPiece*. https://medium.com/@pierre_guillou/nlp-how-to-add-a-domain-specific-vocabulary-new-tokens-to-a-subword-tokenizer-already-trained-33ab15613a41.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc. Chap. 15, pp. 587–604.

Ho, Tin Kam (1995). "Random Decision Forests". In: *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*. ICDAR '95. M: IEEE Computer Society, pp. 278–282. ISBN: 0-8186-7128-9.

Horev, Rani (2018). *BERT Explained: State of the art language model for NLP*. https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270.

Huilgol, Purva (2019). *Accuracy vs. F1-score*. https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2.

Jindrich (2019). *How are the TokenEmbeddings in BERT created?* https://stackoverflow.com/questions/57960995/how-are-the-tokenembeddings-in-bert-created.

Jones, Karen Spärck (1972). "A statistical interpretation of term specificity and its application in retrieval". In: *Journal of Documentation* 28, pp. 11–21.

Joulin, Armand et al. (2016). *Bag of Tricks for Efficient Text Classification*. arXiv: 1607.01759 [cs.CL].

Khan, Suleiman (2019). *BERT Technology introduced in 3-minutes*. https://towardsdatascience.com/bert-technology-introduced-in-3-minutes-2c2f9968268c.

Kim, Augustine Yongwhi et al. (2017). *Automated Text Analysis Based on Skip-Gram Model for Food Evaluation in Predicting Consumer Acceptance*. Tech. rep.

Kim, Eric (2019a). *Demystifying Neural Network in Skip-Gram Language Modeling*. https://aegis4048.github.io/demystifying_neural_network_in_skip_gram_language_modeling.

— (2019b). *Optimize Computational Efficiency of Skip-Gram with Negative Sampling*. https://aegis4048.github.io/optimize_computational_efficiency_of_skip-gram_with_negative_sampling.

Koehrsen, Will (2018). *Beyond Accuracy: Precision and Recall*. https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c.

Kumar, Ajitesh (2021). *Machine Learning – Training, Validation and Test Data Set*. https://vitalflux.com/machine-learning-training-validation-test-data-set/.

Liu, Chi-Liang et al. (2020). "What makes multilingual BERT multilingual?" In: *CoRR* abs/2010.10938. arXiv: 2010.10938. URL: https://arxiv.org/abs/2010.10938.

Luhn, H. P. (Apr. 1958). "The Automatic Creation of Literature Abstracts". In: *IBM J. Res. Dev.* 2.2, 159–165. ISSN: 0018-8646. DOI: 10.1147/rd.22.0159. URL: https://doi.org/10.1147/rd.22.0159.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze (2008a). *Introduction to Information Retrieval*. Cambridge University Press. Chap. 15.

— (2008b). *Introduction to Information Retrieval*. Cambridge University Press. Chap. Text classification & Naive Bayes.

Maron, M.E. (1961). *Automatic indexing: an experimental inquiry*.

Melcher, Kathrin (2018). *Regularization for Logistic Regression: L1, L2, Gauss, or Laplace?* https://dzone.com/articles/regularization-for-logistic-regression-l1-l2-gauss.

Mikolov, Tomas et al. (2013a). "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems*. Ed. by C. J. C. Burges et al. Vol. 26. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.

Mikolov, Tomas et al. (2013b). *Efficient Estimation of Word Representations in Vector Space*. cite arxiv:1301.3781. URL: https://arxiv.org/pdf/1301.3781.pdf.

miyaRanjanRout (2020). *Advantages and Disadvantages of Logistic Regression*. https://www.geeksforgeeks.org/advantages-and-disadvantages-of-logistic-regression/.

Montantes, James (2021). *BERT Transformers — How Do They Work?* https://becominghuman.ai/bert-transformers-how-do-they-work-cd44e8e31359.

Nalisnick, Eric and Sachin Ravi (2017). *Learning the Dimensionality of Word Embeddings*. arXiv: 1511.05392 [stat.ML].

Noe (2020). *How pre-trained BERT model generates word embeddings for out of vocabulary words?* https://datascience.stackexchange.com/questions/85566/how-pre-trained-bert-model-generates-word-embeddings-for-out-of-vocabulary-words.

Ozsubasi, Izgi Arda (2021). *Few-Shot Learning (FSL): What it is and its Applications*. https://research.aimultiple.com/few-shot-learning/.

Paul, Michael (2018). *Regularization*. https://cmci.colorado.edu/classes/INFO-4604/files/slides-6_regularization.pdf.

Peters, Matthew E., Sebastian Ruder, and Noah A. Smith (Aug. 2019). "To Tune or Not to Tune? Adapting Pretrained Representations to Diverse Tasks". In: *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*. Florence, Italy: Association for Computational Linguistics, pp. 7–14. DOI: 10.18653/v1/W19-4302. URL: https://www.aclweb.org/anthology/W19-4302.

Pham, Binh Thai et al. (2021). "Can deep learning algorithms outperform benchmark machine learning algorithms in flood susceptibility modeling?" In: *Journal of Hydrology* 592, p. 125615. ISSN: 0022-1694. DOI: https://doi.org/10.1016/j.jhydrol.2020.125615. URL: https://www.sciencedirect.com/science/article/pii/S0022169420310763.

Pires, Telmo, Eva Schlinger, and Dan Garrette (2019). "How multilingual is Multilingual BERT?" In: *CoRR* abs/1906.01502. arXiv: 1906.01502. URL: http://arxiv.org/abs/1906.01502.

Pothabattula, Santosh Kumar (2019). *A complete understanding of how the Logistic Regression can perform classification?* https://medium.com/analytics-vidhya/a-complete-understanding-of-how-the-logistic-regression-can-perform-classification-a8e951d31c76.

Pradhan, Dilip, Sunakshi Mamgain, and Profile photo for Colleen Farrelly Colleen Farrelly (2020). *What are the advantages and disadvantages for a random forest algorithm?* https://www.quora.com/What-are-the-advantages-and-disadvantages-for-a-random-forest-algorithm.

Ray, Sunil (2017a). *Understanding Support Vector Machine(SVM) algorithm from examples (along with code)*. https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/.

— (2017b). *What are the advantages and disadvantages of Naive Bayes for classification?* https://www.quora.com/What-are-the-advantages-and-disadvantages-of-Naive-Bayes-for-classification.

Rennie, Jason D et al. (2003). "Tackling the poor assumptions of naive bayes text classifiers". In: *Proceedings of the 20th international conference on machine learning (ICML-03)*, pp. 616–623.

Sahil (2021). *Word Embedding: CBOW and Skip-gram*. https://medium.datadriveninvestor.com/word-embedding-cbow-skip-gram-8262e22fa7c.

"TF–IDF" (2010). In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, pp. 986–987. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_832. URL: https://doi.org/10.1007/978-0-387-30164-8_832.

Sanh, Victor et al. (2019). "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *ArXiv* abs/1910.01108.

Schölkopf, Bernhard and Alex Smola (2002). *Learning with Kernels. Support Vector Machines, Regularization, Optimization and Beyond*. Cambridge, MA: MIT Press.

Shchutskaya, Valeryia (2018). *Deep Learning: Strengths and Challenges*. https://indatalabs.com/blog/deep-learning-strengths-challenges?cli_action=1622636755.034.

Sontag, David (2014). *Support vector machines (SVMs), Lecture 2*. https://people.csail.mit.edu/dsontag/courses/ml14/slides/lecture2.pdf.

Tanskanen, Aapo (2020). *How to classify text in 100 languages with a single NLP model*. https://gofore.com/en/how-to-classify-text-in-100-languages-with-a-single-nlp-model/.

Team, Great Learning (2020). *Random Forest Algorithm- An Overview*. https://www.mygreatlearning.com/blog/random-forest-algorithm/.

TensorFlow (2021). *Tokenization*. https://hub.tensorflow.google.cn/google/experts/bert/wiki_books/1.

Teufel, Simone (2014). *Lecture 7: Text Classification and Naive Bayes*. https://www.cl.cam.ac.uk/teaching/1314/InfoRtrv/lecture7.pdf.

Tyagi, Neelam (2021). *L2 and L1 Regularization in Machine Learning*. https://www.analyticssteps.com/blogs/l2-and-l1-regularization-machine-learning.

Vaswani, Ashish et al. (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc.

Weinberger, Kilian (2017). *Bayes Classifier and Naive Bayes*. https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote05.html.

Weinberger, Kilian et al. (2009). "Feature Hashing for Large Scale Multitask Learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: Association for Computing Machinery, 1113–1120. ISBN: 9781605585161. DOI: 10.1145/1553374.1553516. URL: https://doi.org/10.1145/1553374.1553516.

Wikipedia (2021a). *BERT (language model)*. https://en.wikipedia.org/wiki/BERT_(language_model).

— (2021b). *Bootstrap aggregating*. https://en.wikipedia.org/wiki/Bootstrap_aggregating.

— (2021c). *Random Forest*. https://en.wikipedia.org/wiki/Random_forest.

— (2021d). *Support-vector machine*. https://en.wikipedia.org/wiki/Support-vector_machine.