Hyperpartisan News Detection

Bachelor Thesis

presented by
Larissa Strauch
Matriculation Number 1518629

submitted to the
Data and Web Science Group
Prof. Dr. Ponzetto
University of Mannheim

Juli 2019

Contents

List of Algorithms

List of Figures

List of Tables

Glossary

CBoW Continous Bag of Words. 3, 4, 16

IDF Inverse Term Frequency. 2, 13, 14

TF Term Frequency. 2, 13, 14

TF-IDF Term Frequency-Inverse Term Frequency. vi, 2, 13, 14, 16, 17

Chapter 1

Introduction

1.1 Problem Statement

1.2 Contribution

1.3 Related Work

Chapter 2

Fundamentals

2.1 Term Frequency-Inverse Document Frequency

Term frequency (TF) is a measure that denotes how frequently a term *t* appears a the document *d*. One way to compute TF is [https://nlp.stanford.edu/IR-book/pdf/06vect.pdf]

$$tf(t, d) = \frac{1 + \log(tf_{t,d})}{1 + \log(ave_{t \in d}(tf_{t,d}))}$$

where $1 + \log(tf_{t,d})$ reflects how many times the term t appears in document d and $1 + \log(ave_{t \in d}(tf_{t,d}))$ is the highest occurrence of any term in document d_i.

Inverse Doucment Frequency (IDF) points to the assumption that the informativeness of the term *t* is inversely proportional to the number of documents in the collection in which the term appears.[https://nlp.stanford.edu/IR-book/pdf/06vect.pdf]

$$idf(t_i) = log\frac{N}{d \in D : t \in d}$$

Where N is the total amount of doucments in a document set and $d \in D : t \in d$ is the amount how many times the term t appears in the document set.

To compute the weight for the term t_i within the document d_j we simply multiply the *TF* and *IDF* components:

$$w_{ij} = tf(t_i, d_j) \cdot idf(t_i)$$

Therefore, TF-IDF indicates how significant a word is to a document in a collection or corpus. It is regularly used as a weighting factor in Information Retrieval and Text Mining. The TF-IDF value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to control the fact that some words are usually more common than others.
TF-IDF is easy to compute. In addition, it is possible to extract the most descriptive

terms, as well as to calculate the similarity between 2 terms. However, TF-IDF is based on the bag-of-words (BoW) model, which is why it disregards aspects such as text position, semantics and co-occurrence.

2.2 Word Embeddings

Word Embeddings are based on the approach of Harris Distributional Hypothesis from 1951, which states, that words that occur in the same contexts tend to have similar meanings.
A Word Embedding provides a word vector for each word. This is done by extracting features from that word within the context in which it appears and assigning it a place within the vector space. Two similar words will occupy locations areas near one another within this vector space, while words that differ will have positions much further apart. This makes it possible to calculate the distance calculation by computing cosine distance.

2.2.1 Word2Vec

Word2Vec is a ”2-Model Architecture for computing continuous vector representations of words from very large dataset” [Efficient Estimation of Word Representations in Vector Space] that creates an n-dimensional vector space in which each word is represented as a vector. Word2Vecs 2 learning models are the CBoW and Skip-Gram-Model (Figure 3.1).

CBoW uses the context word to predict the target word. The input is a one-hot encoded vector. The weights between the input layer and the output layer can be represented by a $V \cdot N$ matrix *W* where each row of *W* is the *N*-dimensional vector representation v_W of the input word [word2vec Parameter Learning Explained]. The hidden-layer *h* is computed by multiplying the one-hot encoding vector of the input word w_I with the weight matrix *W*.

$$h = W^T_{(k,\cdot)} := v^T_{w_I}$$

Next we have another weight matrix $W' = w'_{ij}$ which is an $N \cdot V$ matrix. With these weights we can finally compute a score u_j for each word in the vocabulary

$$u_j = {v'}^T_{w_j} h$$

where v'_{w_j} is the *j*-th column of the matrix *W'*. Afterwards we use *softmax*, which is a log-linear classification model, to obtain the posterior distribution of words.

$$p(w_j|w_I) = y_j = \frac{exp(u_j)}{\sum_{j'=1}^{V} exp(u_{j'})}$$

In contrast to the CBoW model, Skip-Gram uses the target word to predict the context words. The input is still a one-hot encoding vector, the hidden layers definition stays the same as in the CBoW model, each output is still using the same hidden layer to output matrix as in the CBoW model $p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = p(w_j|w_I) = y_j$ and the function for $u_j = u_{c,j}$ stays the same. However in the output layer, we are now outputting *C* multinomial distributions.

2.3 SVM

2.4 Multinomial Naive Bayes Classifier

The Naive Bayes classifier is based on Bayes' theorem, which comes from the probability calculus and describes the calculation of conditional probability. Each object in this classification approach is assigned to the class for which the highest probability was computed or for which the lowest costs arise in this assignment.
The Multinomial Naive Bayes classifiers assumes that the position of the word does not matter, as well as that the feature probabilities $P(x_i|c_j)$ are independent given a class c. "The probability of a class value c given a test document d is computed as

$$P(c|d) = \frac{P(c) \prod_{w \in d} P(w|c)^{n_{wd}}}{P(d)}$$

where n_{wd} is the number of times word w occurs in document d, $P(w|c)$ is the probability of observing word w given class c, $P(c)$ is the prior probability of class c, and $P(d)$ is a constant that makes the probabilities for the different classes sum to one". [https://link.springer.com/content/pdf/10.1007%2F11871637.pdf]

2.5 Logistic Regression Classifier

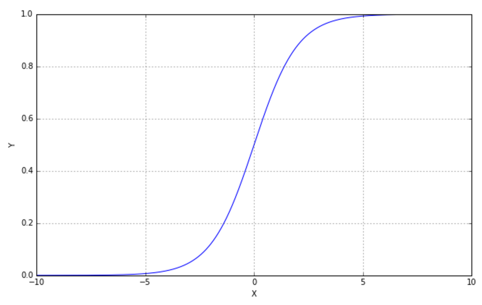


Figure 2.1: sigmoid function[Machine learning algorithms : reference guide for popular algorithms for data science and machine learning]

Logistic Regression is like Naive Bayes, a probabilistic classifier, and thus classifies by estimating the probability $P(Y|X)$ that the object belongs to a particular class. It can be derived analogously to the linear regression hypothesis,

$$h_\Theta(x) = \Theta_0 + \Theta_1 x_1 + ... + \Theta_n x_n = \sum_{i=0}^{n} \Theta_i x_i = \Theta^\tau x$$

where $h_\Theta(x)$ is the predicted value, Θ is the model's parameter vector, Θ_0 is the bias term and x_i is a feature of feature vector. The logistic regression hypothesis generalizes the linear one to use the logistic function

$$h_\Theta(x) = s(\sum_{i=0}^{n} \Theta_i x_i) = s(\Theta^\tau x)$$

and in contrast to linear regression, in logistic regression only values between 0 and 1 are obtained, which can be attributed to the addition of the sigmoid function (Figure 2.1).

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

Combining these two therefore results in

$$h_\Theta(x) = \frac{1}{1 + e^{-\Theta^\tau x}}$$

To find the optimal Θ by which we get an $h_\Theta(x)$ that is close to 0 or 1, we maximize the log-likelihood relying upon the output class, therefore the optimization problem can be expressed, utilizing the indicator notion as the minimization of the loss function [Machine learning algorithms : reference guide for popular algorithms for data science and machine learning]:

$$l(\Theta) = -\sum_{i=1}^{n} y_i \ln(z_i) + (1 - y_i) \ln(1 - z_i) \qquad , \forall i, y_i \in \{0, 1\}$$

where $z = h_\Theta(x)$.
This implies, if $y = 0$, the first term ends up 0 and the second $\ln(1 - z)$, which results in a log-likelihood of 0. In the event that $y = 1$, the second term ends up 0 and the first one corresponds to the log-likelihood of z. Along these lines, both cases are integrated into a solitary articulation.
A major issue in machine learning is the aspect of overfitting. Overfitting tends to adjust the model too much to the training data. This happens when a model learns the details in the training data so that the performance of the model is contrarily influenced by new data. This implies that noise or random fluctuations in the training data are recorded and learned as concepts by the model. The issue is, that these concepts do not apply to new data and negatively impact the generalization of the model. Therefore, there is the process of *regularization*, which is a form of regression that decreases the coefficient estimation to 0 and assumes that smaller weights generate simpler models and thus helps avoid overfitting.

Two of the most commonly used regression techniques are *Lasso Regression (L1)* and *Ridge Regression (L2)*, which differ mainly in the penalty term.
The *L1* loss function minimizes the sum of the absolute differences between the target value and the estimated values. If input characteristics have weights closer to zero, this results in a sparse L1 standard. In the Sparse solution, most of the input features have zero weights and very few features have non-zero weights. The L1 regulation offers a function selection. This is done by associating insignificant input features with zero weights and useful features with a non-zero weight. In the L1 control, we punish the absolute value of the weights. It is defined as:

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

Which we add to the loss function:

$$l_\lambda^R(\Theta) = -\sum_{i=1}^{n} y_i \ln(z_i) + (1-y_i)\ln(1-z_i) + \lambda \sum_{j=1}^{p} |w_j|$$

The L2 loss function is essentially about minimizing the sum of the square of the differences between the nominal value and the estimated values. L2 control forces the weights to small weights, but does not make them zero and leads to a not sparse solution. In addition, L2 is not robust against outliers, since square terms inflate the error differences of the outliers and the regularization term tries to correct them by punishing the weights. It is defines as follows:

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

,which we add as well to the loss function:

$$l_\lambda^L(\Theta) = -\sum_{i=1}^{n} y_i \ln(z_i) + (1-y_i)\ln(1-z_i) + \lambda \sum_{j=1}^{p} |w_i^2|$$

2.6 Decision Trees and Random Forest Classifier

2.6.1 Decision Trees

Starting from the root node of a tree, a feature is evaluated from which a branch is subsequently selected. This process is repeated until the last node in the tree (leaf) is reached, which supplies the corresponding class. Different approaches have been developed in Decision Trees. One of the first is called *Iterative Dichotomizer (ID3)* and required explicit features. This condition led to the development of the *C4.5* formulation, which also manages continuous (but summarized and discretized) values. In addition, C4.5 was also known for its ability to transform a tree into a series

of conditional expressions. The latest development is called *Classification and Regression Trees (CART)*. These kinds of trees can oversee both downright and numeric highlights, can be utilized in either characterization or relapse assignments, and don't utilize a standard set as an inner portrayal.[Machine Learning Algorithms - Second Edition].
The algorithm uses impurity measures to select a particular branch from a leaf. Among others, the two most common measures are "Gini" and "Cross Entropy Index". [Machine Learning Algorithms - Second Edition -Nicht zitiert]. These impurity measures are applied to each candidate subset, and the resulting values are combined (e.g., averaged) to provide a measure of the quality of the split,

$$I_{Gini}(j) = \sum_i p(i|j)(1 - p(i|j))$$

$$I_{Cross-entropy}(j) = -\sum_i p(i|j)log(p(i|j))$$

where j is a certain node, $p(i|j)$ is the probability with $i \in [1, n]$ associated with each class.
The *Gini* Index measures the impurity based on the likelihood of misclassification if a sample is categorized using a label randomly chosen from the node subset distribution. The minimum index (0,0) is achieved when all examples are characterized into a solitary class.
Cross-Entropy depends on information theory, and accepts invalid qualities just when samples having a place with a solitary class are available in a split, while it is greatest when there's a uniform conveyance among classes.

2.6.2 Random Forest Classifier

The Random Forest classifier is a classification technique that creates multiple decision trees from randomly selected subsets of training data. Each tree in this process may make a decision, these votes are then aggregated to determine the final class. According to Breiman [https://www.stat.berkeley.edu/ breiman/randomforest2001.pdf] the Random Forest algorithm is as follows:

Algorithm 1: Random Forest

```
1  Set the number of decision trees $N_c$
2  for $i \leftarrow 1$ to $N_c$ do
3      Create a dataset $D_i$ sampling with replacements from the original
         dataset $X$
4  Set the number of features to consider during each split $N_f$
5  Set an impurity measure
6  Define an optimal maximum for each tree
7  for $i \leftarrow 1$ to $N_c$ do
8      Random Forest: Train the decision tree $d_i(x)$ using the dataset $D_i$ and
         selecting the best split among Nf features randomly sampled
9      Extra-trees: Train the decision tree $d_i(x)$ using the dataset $D_i$
         computing before each split $n$ random thresholds and selecting the
         one that yield the least impurity
10 Define an output function averaging the single outputs or employing a
     majority vote
```

2.7 Cross Validation

Cross Validation is a model validation technique used to survey how the result of measurable statistical analysis generalizes into an independent dataset. The idea is to devide the entire dataset into a moving test and training set. The size of the test set is determined by the quantity of folds, so that at *k* emphasess the test set covers the whole original dataset.
A round of Cross Validation comprises of separating a sample of data into corresponding subsets, performing the analysis on the training set, and validating the analysis on the test set. To decrease fluctuation, most strategies perform several rounds of *Cross Validation* utilizing various partitions and combine the validation result over the rounds to obtain an estimate of the predictive performance of the model.

2.8 Evaluation

In order to evaluate how well a classifier works, several procedures exist. The ones used during the competition are *Accuracy*, *Recall*, *Precision* and *F1-Score*, for which the Confusion Matrix (Table 2.1) forms the basis. Each column of the matrix represents the instances of a predicted class, while each row represents the instances of the actual class.

		Predicted Class	
		positive	**negative**
Actual Class	**positive$'$**	True Positives	False Negatives
	negative$'$	False Positives	True Negatives

Table 2.1: Confusion Matrix

True Positive means that the classifier predicted a class which actually corresponds to it, whereas *False Positive* means that a class was predicted that does not correspond to the actual class. In contrast, there are the *False Negatives*, where the classifier predicted a class as not belonging, although the instance actually belongs to it whereas *True Negative* means that the class was correctly classified as not belonging.
The four evaluation metrics are computed as follows [Machine Learning Algorithms - Second Edition]:

- Accuracy: $\frac{TP+TN}{TP+TN+FP+FN}$
  Defines the correct classification to the total number of cases

- Precision: $\frac{TP}{TP+FP}$
  Defines the correct classification of cases predcited to be positive

- Recall: $\frac{TP}{TP+FN}$
  Defines the correct positive classification of cases that are actually positive

- F1-Score: $2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$
  Defines the average of precision and recall, where an F1 value reaches its best value at 1 and worst at 0

Chapter 3

Data

3.1 Data Description

The given data, on which we want to build our model on, was provided by zenodo.org as part of SemEvals Task 4 [Link zu Task hinzufügen]and consists of 2 independent datasets, which in turn have been divided into a GroundTruth-, Training- and Validation set.
The first dataset, recognizable by the term 'byPublisher', reflects the publisher's general bias set forth by BuzzFeed journalists or MediaBiasFastCheck.com beforehand. It consists of a total of 750,000 items, of which 600,000 belong to the Training- and 150,000 to the Validation set.
In return, the second dataset, recognizable by the term 'byArticle', was scrapped by crowdsourcing at hand and therefore consists of only 645 items without a Validation set.

The GroundTruth dataset was provided as an XML File and consists of the features 'article url', 'labeled-by', 'id' and 'hyperpartisan'. In addition, the GroundTruth dataset scrapped 'byPublisher' contains the feature 'bias'.

- Article-url: Contains the article's url.
- Labeled-by: Reflects whether the respective article was labeled 'byPublisher' or 'byArticle'.
- Id: Allocates each article a unique id.
- Hyperpartisan: Displays whether the particular article has been labeled as hyperpartisan or not.
- Bias: Divides the publisher's bias into 'left', 'left-center', 'least', 'right-center' and 'right'.

The Training dataset was as well provided as an XML file and contains the contents of the website of the respective article. In addition, it consists of the features 'article title' 'published-at' and 'id'.

- Article title: Represents the articles title.
- Published-at: Specifies the published date.
- Id: A unique id, which is the same as the corresponding entry in the GroundTruth dataset.

The given Data has been cleaned in advance, therefore no additional steps were necessary.
The main focus of the datasets is on the Hyperpartisan feature, on which we want to classify the articles as this thesis progresses.

3.1.1 Dataset labelled by Publisher

As mentioned above, this dataset consists of a total of 750,000 articles and is divided into a training record consisting of 600,000 articles and a validation set consisting of 150,000 articles. Summarizing these two sets of data, a total of 375,000 were labelled as 'Hyperpartisan' and 375,000 were not – which corresponds to a 50:50 distribution. But even individually, this distribution does not change.

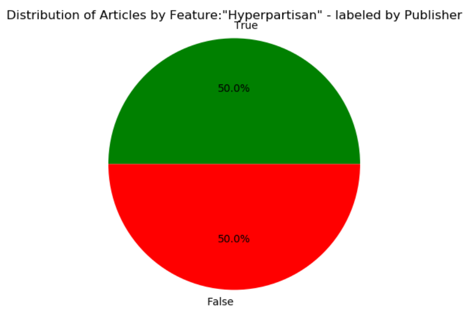
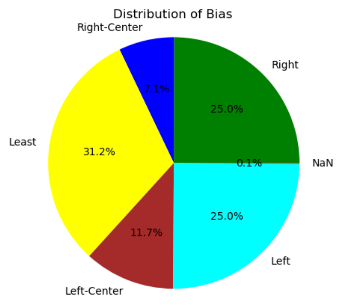


Figure 3.1: Distribution of as Hyperpartisan labelled articles by publisher

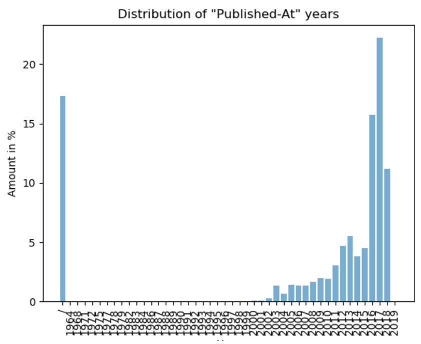
This dataset also includes the feature 'bias', which informs you about the general bias of the publisher. All 375,000 Hyperpartisan labelled all are assigned to either the left or right sectors, but none are right-centre, least or left-centre and are again 50:50 distributed.
The other 50% are split between the remaining bias, with 'Least' owning the largest share at 37%.
The publicity data is distributed over the years 1964-2018, with most of the data coming from 2012-2018.

3.1.2 Dataset labelled by Article

The dataset labelled by Article is a little different to the larger one labelled by publisher. Here the articles were individually labelled by hand. Accordingly, the

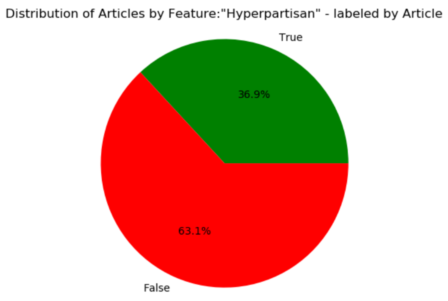


(a) Distribution of Bias

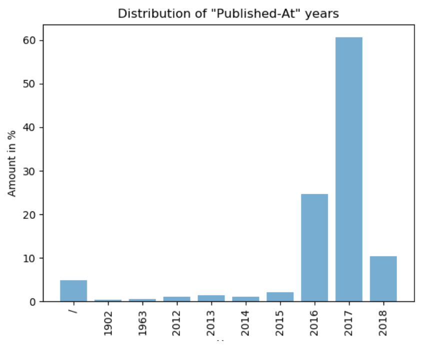


(b) Distribution of publishing years

distributions of this dataset are completely different. This becomes quiet obvious if we look at how the distribution of the Hyperpartisan labelled articles is here. Here



(c) Distribution of Hyperpartisan labelled articles



(d) Distribution of publishing years

we can see that there is no 50:50 distribution left. Only 36.9% were defined here as Hyperpartisan as shown in figure 2.4.
Moreover, in this dataset, the distribution of publication data is not mainly from the years 2012-2018, but in 2016-2018, with the largest number of articles dating back to 2017 at just under 60% as we can see in figure 2.5. Altogether all 645 articles date from the years 1902-2018.

3.2 Data Analysis

Hyperpartisan means "extremely partisan; extremely biased in favor of a political party." [definition]. This often materializes in relation to significant political events. As a result, in the following chapter, I will discuss in detail the direct link between the various features, especially the correlations between publication dates and label, as well as the connections between publisher and label.

Publisher	Amount
The Gateway Pundit	17
OpsLens	14
RealClearPolitics	13
New York Post	10
Salon	8

Table 3.1: Publishers who have published more than 7 Hyperpartisan Articles

Publisher	Bias	Amount
Fox Business	Left	96175
CounterPunch	Left	39832
Mother Jones	Left	36730
Truthdig	Left	25056
Daily Wire	Right	18570

Table 3.2: Publishers who have published more than 10.000 Hyperpartisan articles

The two tables 2.1 and 2.2 list the publisher who produced the highest amount of Hyperpartisan articles in the respective dataset. To keep track, only those publishers who have published more than 7 Hyperpartisan articles in the dataset 'labelled by article' and who have published more than 10.000 hyperpartisan articles in the dataset 'labelled by publisher' are included in the tables. However, a problem here is the dataset 'Labelled by Publisher', since this record, as previously described, has been labelled by the overall bias of the publisher. Meaning, for the further development, we can not include the publishers, since each article of a publishing house has the same label. What I would like to discuss later, however, is the connection between whether or not one of the publishers listed in Table 2.2 published more articles in important political years.

3.3 Data Preparation

In order to be able to work with the existing data in the further course of this project, several preprocessing steps are necessary. In the preprocessation phase of my bachelor's thesis, the data therefore went through the following steps:

1. Read the XML files.
2. Filter out the important information.
3. Merge the Groundtruth- and Training datasets into a csv file.
4. Remove special characters and stop words.
5. Tokenization and stemming of the datasets.

As a result, in the following section, I will go further into detail how I preprocessed my data.

3.3.1 Read the XML files

Since it is difficult to work with the given data in an XML format, it is necessary to bring them into a format with which it'll be easier to work with. The particular challenge hereby is the size of the dataset labelled by publisher. A standard algorithm for reading XML files is provided by pythons DOM library ElementTree, called "ElementTree.parse". This method returns an ElemenTree type, which "is a flexible container object, designed to store hierarchical data structures in memory" [http://effbot.org/zone/element.htm]. Meaning, that this library forms the entire model in the memory which can pose a problem with very large files, such as ours. As a substitute,I therefore use the method "ElemenTree.iterparse", which can process XML documents in a streaming fashion, retaining in memory only the most recently examined parts of the tree.

3.3.2 Filter out the important information

As mentioned in Chapter 2.1 Data Description, the XML files include various features, which will play an important role in the further course of this work. It is now necessary to filter these features out of the XML format so the can be better worked with later. In order to be able to do this, we pass an algorithm the, by the iterparse method read content, which then passes through this algorithm in a double for-loop and checks for each item of an element in the content which "key" is currently processed. I then save this in an array, so that the features which have already been parsed, can be used later.

Algorithm 2: Parse Ground-Truth File

```
  Input: Ground-Truth File
  Output: Trained Classifier
1 for event, elem in content do
2     for key, value in elem.items() do
3         if key == 'id' then
4             id_array.append(str(value))
5         else if key == 'published-at' then
6             published_at_array.append(value)
7         else if key == 'title' then
8             title_array.append(value)
```

3.3.3 Merge the Groundtruth- and Training datasets into a csv file

Since both, the Groundtruth- and Trainingdata contain important information, it is necessary to merge them into one file. I decided to use Python's library pandas to write the two files into one csv file. This, and especially Pandas, allows us to read the file more quickly as well as to access individual rows and columns of the merged file in a targeted manner.

Algorithm 3: Merge Groundtruth- and Trainingdatasets

```
1 feature_extraction.parse_features(xml_training, publisher)
2 groundtruth_parser.parse_groundtruth(xml_gt, publisher)
3 content = content_parser.parse_content(content_training)
4 a_id = feature_extraction.get_id_array(publisher)

5 published = feature_extraction.get_published_at_array(publisher)
6 title = feature_extraction.get_title_array(publisher)
7 bias = groundtruth_parser.get_bias_array(publisher)
8 hyperpartisan = groundtruth_parser.get_hyperpartisan_array(publisher)
9 columns = {"ArticleID": a_id,"PublishedAt": published,"Title":
    title,"Bias": bias,"Content": content "Hyperpartisan": hyperpartisan}

10 tf = Pd.DataFrame(columns)
11 tf = tf[['ArticleID','Published', Title','Bias','Content','Hyperpartisan']]
12 tf.to_csv(titlecsv,encoding='utf-8',index=False)
```

3.3.4 Remove special characters and stop words

Now that we have merged both files and written them into a csv file, we need to remove special characters and stop words. Especially the removal of stopwords is necessary since not all words presented in a document, such as auxiliary verbs, conjunctions and articles [TextClassificationUsingMachineLearning] are useful for training a classifier. It also decreases our corpus which makes it easier to classify later on. (Warum?)

Algorithm 4: Remove special characters and stop words

```
1 stop = stopwords.words('english')

2 df['Content'] = df['Content'].apply(lambda x: ''.join([item for item in
    x.split() if item not in stop]))
3 df['Content'] = df['Content'].map(lambda x: re.sub(r"[â-zA-Z0-9]+", "",
    x))
```

Here it becomes obvious that using pandas was a good choice, as we can now specifically access the 'Content' and 'Title' columns in order to perform this step of preprocessing on only these two and not the whole file.

3.3.5 Tokenization and Stemming of the datasets

After cleaning the dataset, the words are tokenized in order to convert them into numerical vectors so that a classifier is able to work with them. Tokenization is definied as "The process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input".

Stemming is the procedure of reducing the word to its grammatical (morpho-syntatctic) root. The result is not necessarily a valid word of the language. For example, ”recognized” would be converted to ”recogniz”. Still, the basic word almost always contains the very meaning of the word. Stemming is advantageous in that the algorithm used later now only has to fall back on a few different words and not many, all of which have the same meaning.

In order to implement Stemming and Tokenization, only 2 lines of python code are necessary, due to the Pandas dataframe.

```
df["Content"] = df["Content"].apply(nltk.word_tokenize)
df['Content'] = df['Content'].apply(lambda x: [stemmer.stem(y) for y in x
    ])
```

This will transform our output file in the following way:

Chapter 4

Methodology

The main process for Text Classification includes 7 steps of which the first 4 have already been performed in Chapter 2.

1. Read the Document.
2. Tokenize Text.
3. Stemming.
4. Delete Stopwords.
5. Vector Representation of the Text.
6. Feature Selection and/or Feature Transformation.
7. Learning Algorithm.

As already mentioned in Chapter ”Introduction”, we build our classifier model by using BERT-Embeddings. To compare how our model performs, in this chapter, I will discuss the classic methods and algorithms that I have used in order to achieve this comparison.

4.1 Important Algorithms

In order to avoid the repetition of algorithms used in the same context, in this section, I will explain some recurring algorithms.

Algorithm 5: Saving a trained model

```
import pickle
filename = 'finalized_model.sav'
pickle.dump(model, open(filename, 'wb))
```

```
Algorithm 6: Loading a trained model
1 loaded_model = pickle.load(open(filename, 'rb'))
```

4.2 Vector Representation of the Text

In order for our classifier to be able to work with the text, we first need to transform our words into a feature vector representation. A document is a sequence of words [Text Categorization with Support Vector Machines] so a document can be presented by a One-Hot encoded vector, assigning the value 1 if the document contains the feature-word or 0 if the word does not appear in the document. [TextClassificationUsingmachineLearning]. However, using this technique for word representation, resolves in a $V \cdot V$ Matrix, as we have V-diemensional vertor for each out of V words which can lead to huge memory issues. In addition this does not notion similiarity between words. Therefore I will go into further detail for better approaches in the next 2 subchapters.

4.2.1 Term Frequency-Inverse Document Frequency

A comparative approach I used in the course of my Bachelor Thesis is Term Frequency-Inverse Document Frequency. By using TF-IDF, we're able to represent a word as a vector by assigning it weight which is computed through Term-Frequency multiplied with Inverse-Term-Frequency. Table 4.1 shows the 10 terms which have been assigned the highest TF-IDF weights. Python's library *sklearn* provides two ways to implement TF-IDF without having to program TF and IDF by itself. In order to get a generally better overview, I will now explain the 2-step implementation, but also briefly explain how both steps can be combined in one.

Term-Frequency, as mentioned in chapter 1 – Principles, is a measure that denotes how often a term appears in a document. Inverse Document Frequency, on the other hand, reflects the importance of a term throughout a document corpus. To implement the TF-IDF measure, *scikit-learn* provides the classes *CountVectorizer* and *TfidfTransformer* of the submodule *sklearn.feature_extraction.text*. Therefore, in the first step of the TF-IDF-implementation, both classes must be imported. As an input example, I use the first article of the "byArticle" labelled dataset.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
```

In order to calculate our TF measure, we use the method *fit_transform()* of the class *CountVectorizer*, which we pass our document corpus as a parameter.

```
count_vect = CountVectorizer()
content_counts = count_vect.fit_transform(corpus)
```

fit_transform() learns a vocabulary dictionary of all tokens [sklearn Documentation], counts how many times a term *t* occurs in a document *d* and converts the text document into a token matrix, which, in this example, would result in the following output:

```
<1x84 sparse matrix of type '<class␣'numpy.int64'>'
with 84 stored elements in Compressed Sparse Row format>
```

The calculation of the IDF-measure in the second step is similar to the calculation of the TF-measure. Again, the method *fit_transform()* is called. In contrast to the method of the *CountVectorizer*, we no longer pass the text document as a parameter, but the token matrix of *CountVectorizer*.

```
tfidf_transformer = TfidfTransformer()
content_tfidf = tfidf_transformer.fit_transform(content_counts)
```

It should be noted that the shape of the finalized TF-IDF matrix depends on the document corpus. The first time the conversion is carried out, the two objects of the respective classes learn the respective vocabulary through the keyword *"fit_"*. If we want to convert a second text document to TF-IDF vectors afterwards, which is supposed to be used in connection with the already converted text document, we have to make sure to use the same *CountVectorizer* and the same *TfidfTransformer*. This is necessary because otherwise the error message "Dimension Mismatch" would occur in later prediction calls. Let's take the following example:
If we convert the two datasets labelled by article and publisher with the same *CountVectorizer* and *TfidfTransformer* as in algorithm 7, we obtain the following dimensions:

- By Publisher: (600000, 708863)
- By Article: (645, 708863)

The first number in the vectors refers to the number of rows in the document and the second, how many columns the matrix contains. As we can clearly see, the two dimensions match because the documents were converted with the same fitted models.

Algorithm 7: TF-IDF Fitting

```
count_vect = CountVectorizer()
tfidf_transformer = TfidfTransformer()

x_train_counts = count_vect.fit_transform(x_train)
x_train_tfidf = tfidf_transformer.fit_transform(x_train_counts)

x_test_counts = count_vect.transform(x_test)
x_test_tfidf = tfidf_transformer.transform(x_test_counts)
```

If we would call the methods *fit_transform()* for both datasets, *CountVectorizer* and *TfidfTransformer* would be initialized each time, which would result in the following dimensions:

- By Publisher: (600000, 708863)
- By Article: (645, 11485)

By Article	TF-IDF Weight	By Publisher	TF-IDF Weight
trump	0.091165	balls	0.057251
bannon	0.065129	said	0.051183
president	0.061829	medicare	0.047450
kimmel	0.052415	martin	0.046519
money	0.051647	school	0.044264
americans	0.049132	university	0.043572
people	0.047342	says	0.042079
obamacare	0.042512	trump	0.041235
wall	0.042117	carrier	0.040903
control	0.039068	degree	0.040704

Table 4.1: Top 10 terms by TF-IDF weight

4.2.2 Word Embeddings

Word2Vec

For a forther comparison and an approximation to the the later on used classification model, I did not only uf TF-IDF as a Vector representation method as part of my Bachelor Thesis, but also Word Embeddings - especially the *Word2Vec* model. Word2Vec represents words as vectors. Unlike the TF-IDF method, however, not only word frequencies and -priorities are considered, but also the connection of individual words to others. Again, several methods of implementation exist. As part of my Bachelor Thesis, I decided to use the library *gensim* to implement my Word2Vec model. With *gensim* it is possible to do unsupervised semantic modelling from plain text. Thsi makes it possible to implement a Word"vec model using only a few lines of code without having to program Skip-Gram of CBoW yourself.

```
model = gensim.models.Word2Vec(vocab, min_count=10, window=10, size
    =300, iter=10)
```

The algorithm above shows that the implementation of the model is straightforward, as it is pretty much the only step we need to program. By default, gensim uses CBoW which can be changed by adding the following parameter to the parameters list: *sg=1*. As for the other parameters, 18 more exist which can be viewed at "https://radimrehurek.com/gensim/models/word2vec.html", but I decided to focus only on the important ones. *Vocab* is our text corpus, which needs to be transformed into a list of tokenized sentences. *Size* determines the dimension of the word vectors, *window* the maximum distance between the current and predicted word within a sentence, *min_count* how often a word must occur to be included in the vocabulary and *iter* how many iterations should be performed on the corpus. What exactly happens here is that we train a neural network with a single hidden layer on which the model is trained to predict the current word based on the con-

By Article	Similarity	By Publisher	Similarity
president	0.9992413520812988	obama	0.5890584588050842
donald	0.9992144107818604	clinton	0.5385712385177612
election	0.9980630874633789	bannon	0.5277745127677917
presidential	0.9978925585746765	priebus	0.5117671489715576
campaign	0.9978247880935669	hillary	0.48596829175949097

Table 4.2: Most similar word to ’trump’

text. The resulting vector consists of several features that describe the target word. After the model has been trained it is possible to find out information like the similarity between 2 words. Table 4.3 shows the most similar words to ”trump”, which was assigned the highest TF-IDF weight.

```
# similarity between identical words
model.wv.similarity(w1="trump", w2="trump")
1,0

# similarity between two unrealted words
model.wv.similarity(w1="trump", w2="explosion")
-0.11690721
```

The functionality here is that the cosine similarity is calculated between 2 words in the vector space. As the range of the cosine similarity can go from [-1 to 1], words that are completely the same are assigned the value *1* and words that are not similar at all are given the value *-1*. Our resulting word vectors now have the dimension defined in the parameter *size*. In order to be able to form features from them, I averaged the Word Embeddings of all words in a sentence.

```
def sent_vectorizer(sent, model):
    sent_vec = []
    numw = 0
    for w in sent:
        if numw == 0:
            sent_vec = model[w]
        else:
            sent_vec = np.add(sent_vec, model[w])
```

In order not to have to re-train the model every time, since this can take some time on large data sets, it is as well possible to save the trained model in order to access it again later.

```
word_vectors = model.wv
fname = "article_vectors.kv"
word_vectors.save(fname)
```

4.3 Classification Algorithms

Classification is about predicting a particular outcome based on given training data. For this prediction, a classification algorithm processes the training data set, which

consists of a set of features and the respective predictions. The algorithm attempts to discover relationships between given features of the instances and the associated classes to learn a function which makes it possible to predict the correct class based on the features of an instance. Thereafter, the algorithm receive a test dataset which it has not seen before. This dataset contains the same features as the training set but not the corresponding class names. With the previously learned function, the algorithm now assigns a class name to each instance of the test record.
Classic classification algorithms include *Multinomial Naive Bayes*, *Support Vector Machines*, *Random Forest* and *Logistic Regression*, which is why, in the following chapter, I will explain the basic procedure for implementing these algorithms. In addition, I will discuss the aspect of *Grid Search*, which gives us the optimal parameter assignment to a given set of data for these algorithms.

4.3.1 SVM

4.3.2 Multinomial Naive Bayes Classifier

The *MultinomialNB* class of the library *scikit-learn* implements the Naive Bayes algorithm for multinomial distributed data and is one ot the two classic Naive Bayes variants used in text classification. Here, the distribution is parameterized by vectors $V_y = (Y_{y1}, ..., Y_{yn})$ for each class y, where n is the size of the vocabulary and V_{yi} is the probability $P(x_i|y)$ of feature i appearing in a sample belonging to class y. The parameter V_y is estimated by a smoothed version of the maximum likelihood

$$\hat{V}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times a feature i appears in a sample of class y in the training set T and $N_y = \sum_{i=1}^{n} N_{yi}$ is the total count of all features for class y. The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations.

The *MultinomialNB* classifier includes the parameters *alpha*,*fit_prior* and *class_prior*. *alpha* specifies α's value with which smoothing should be made. *fit_prior* specifies whether the class probalilities should be leraned in advance and *class_prior* specifies the prior probabilities of the classes.

```
from sklearn.naive_bayes import MultinomialNB
MultinomialNB(alpha=1.0, fit_prior=True, class_prior=None)
```

4.3.3 Random Forest Classifier

As described in Chapter 2.6.2, the Random Forest Classifier is a classification technique that creates multiple decision trees from randomly selected subsets of training data. For implementing the Random Forest Classifier, *scikit-learn* provides the class *sklearn.ensemble.RandomForestClassifier*. Unlike the original publication

[Random Forests", Machine Learning, 45(1), 5-32, 200], the *scikit-learn* classifier determines the final class by averaging the probabilistic forecasts as opposed to having each classifier vote in favour of a solitary class.
The Random Forest classifier includes 17 parameters, of which I have included 6 in my classification.

```
from sklearn.ensemble import RandomForestClassifier
RandomForestClassifier(min_samples_leaf=1, n_estimators=500, criterion='
    gini', max_depth=15, max_features='auto', bootstrap=True)
```

where *n_estimators* determines the number of trees in the forest, *criterion* which impurity measure to utilize, *max_depth* determines the maximum depth of a tree, *min_samples_leaf* determines the base number of samples to be at a leaf node, *max_features* the quantity of features that must be considered in the search for the best partition and *bootstrap* indicates whether bootstrap models are utilized when making trees.

4.3.4 Logistic Regression Classifier

The Logistic Regression classifier is a model for classification, where the probabilities depicting the potential results of a solitary class are modelled utilizing a logistic function. The provided classifier of *scikit-learn* offers the possibility not only to classify binary, but also One-vs-Rest and multinomial. There are several solvers available for this, yet since our classification problem only refers to binary classification, I will only discuss those aspects of the logistic regression classifier in the following section.
Scikit learns Logistic Regression Classifier provides L1, L2 and Elastic-Net Regularization.[https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression]
L1 solves the regularized logistic regression by minimizing the following cost function:

$$\min_{w,c} \quad \|w\|_1 + C\sum_{i=1}^{n} \log(\exp(1 + e^{-y_i x^T x_i})$$

whereas *L2* solves it the following way:

$$\min_{w,c} \quad \frac{1}{2}w^T w + C\sum_{i=1}^{n} \log(1 + e^{-y_i w^T x_i})$$

,where $C > 0$ is penalty parameter, w is the vector, w^T is an additional dimension, x_i is an instance of the vector and y_i is an instance label pair $\in \{-1, +1\}$ [https://www.csie.ntu.edu.tw/ cjlin/papers/liblinear.pdf].
A solvers, *liblinear*, *newton-cg*, *lbgfs*, *sag* and *saga* are available. *Scikit-learn* points out [https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression], that *liblinear* is a good algorithm for small datasets, whereas *sag* and *saga* are faster for large datasets. Also, *newton-cg*, *lbgfs* and *sag* only handle *L2* penalty, whereas *liblinear* also handles *L1* penalty and *saga* in addition *elasticnet*.
Liblinear uses a coordinate descent algorithm, the *sag* solver a Stochastic Average

Gradient descent, *saga* is a variant of *sag* and therefor supports the non-smooth penalty *L1* and *elasticnet* and the *lbfgs* solver is an optimization algorithm that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm.

4.3.5 Grid Search

Parameter	Result by Article	Result by Publisher
alpha	0.7	0.5
fit_prior	False	True

Table 4.3: Grid Search Result of the Multinomial NB Classifier

A significant perspective in machine learning is the aspect of hyperparameters. These are parameters that are not determined by the learning algorithm, but those that must be determined beforehand. In our classification algorithms, for example, theses are the parameters to be passed to the classifier. For instance, which size *n_estimators* in the *Random Forest* classifier should assume. To solve this issue the algorithm *GridSearch* exists. This algorithm is used to find the optimal hyperparameters of a model, which then leads to the most accurate predictions.
The procedure is as follows: We define a set of parameters of which GridSearch trains the given classifier for all possible combinations and measures the performance by cross-validation. This ensures that our trained model has received the most samples from our training data set.
A simple implementation of GridSearch is provided by *scikit* learn through the class *sklearn.model_selection.GridSearchCV*. This class evaluates all possible combinations of parameters when calling the method fit() and keeps the best combination.
The parameters that can be passed to the class include *estimator*, which specifies the classifier, *param_grid*, which is our parameter set, *scoring*, which specifies which measure we want to use to evaluate our test set, and *cv* how many cross-validation splits we want to perform. Algorithm 8 shows a practical example of how to use *GridSearchCV* while table 4.4 shows the corresponding results.

Algorithm 8: Grid Search for Multinomial NB classifier

```
from sklearn.model_selection import GridSearchCV
from sklearn.naive_bayes import MultinomialNB
parameter_candidates = {'alpha': np.linspace(0.5, 1.5, 6), 'fit_prior':
  [True, False]}
clf = GridSearchCV(estimator=MultinomialNB(),
  param_grid=parameter_candidates, scoring='accuracy', cv=10)
clf.fit(features, labels)
```

4.3.6 Training and Predicting

Algorithm 9: Classifier fitting

```
Input: Classifier model, Features, Labels
Output: Trained Classifier
1 classifier = model
2 classifier.fit(features, labels)
```

Chapter 5

Bidirectional Encoder Representations from Transformers

Chapter 6

Evaluation

Chapter 7

Conclusion

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Master-/Bachelorarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Er- klärung rechtliche Folgen haben wird.

Mannheim, den 31.08.2014 Unterschrift