MASTER THESIS

Bc. Jan Bílek

Genres classification by means of machine learning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	Mgr. Roman Neruda Csc.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2018

Title: Genres classification by means of machine learning

Author: Bc. Jan Bílek

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda Csc., Institute of Computer Science, The Czech Academy of Sciences

Abstract: Abstract.

Keywords: key words

Contents

1. Introduction

Two pages wrapping up following 3 sections.

Motivation

- help librarians with classification
- recommender systems - recommends similar book

Goals

- Compare various approaches to text classification on book genres
- Find out how much text is needed to distinguish the genre
- Provide a simple online tool predicting genres given a short excerpt of the book
- Insights (typical words for genre, most similar books)

Outline

2. Background and Related Work

In this chapter, we first define what is a classification problem and introduce few related terms. Next, we describe several classification algorithms that can be used for text classification. Finally, we discuss document representation techniques starting with *bag of words*, explaining the concept behind popular word embedding *word2vec* and paragraph vector up to convolutional neural networks and their usage in text analysis. In the final part of this chapter, we briefly introduce *Project Gutenberg* – an online repository of freely available books which serves as a source of our datasets.

2.1 Classification

- what is a classification (define classification problem)
- two classes vs. multiple classes
- introduce terms such as feature vector, label

2.1.1 Evaluation metrics

To compare two models trained on a given task, we have to define a proper *evaluation metric*. The goal is then to find a model with the best score with respect to this evaluation metric. We introduce two of commonly used metrics applicable to both *binary* and *multiclass* classification – *accuracy* and *F1-score*.

Accuracy

Precision, Recall and F1-score

2.1.2 Similarity metric

It might be interesting to look at the similarities between data points. In the genre classification task, this would describe how similar are two book snippets by comparing their vector representation. Similarity between two objects is usually between 0 and 1 or -1 and 1 with 1 being identical objects and 0 or -1 totally dissimilar objects.

Cosine similarity

Cosine similarity determines similarity between two data points x and y with d dimensions based on the angle between them, in particular:

$$cosine\_similarity(x,y) = cos(\alpha) = \frac{x \cdot y}{||x|| \cdot ||y||} = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2} \cdot \sqrt{\sum_i y_i^2}}$$

This is equivalent to *dot product* of x and y normalized using $l2$ metric. This means the metric is invariant to the scale of individual vectors:

$$\begin{aligned} cosine\_similarity(mx, ny) &= \frac{mn \sum_i x_i y_i}{m\sqrt{\sum_i x_i^2} \cdot n\sqrt{\sum_i y_i^2}} \\ &= \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2} \cdot \sqrt{\sum_i y_i^2}} \\ &= cosine\_similarity(x, y) \end{aligned} \tag{2.1}$$

Cosine similarity is a value between -1 and 1.

2.1.3 Hyperparameter optimization

Regularization strength

- parameter in front of regularization term
- defines how much are complicated models (models with high coefficients) penalized
- value highly problem-dependent - set through cross-validation

K-fold cross-validation

K-fold cross-validation is a technique widely used to choose values of model parameters which are specific to the given task - e.g. the regularization strength.

It first splits the training set randomly into k folds. For the given parameter setting, model is trained on $k-1$ folds and tested on the remaining one. This is done k times so that every fold is used as a test fold once. The score for the parameter setting is then computed as an average of the k scores.

Popular values for k are 3, as it is quick because the model has to be trained only 3 times for the given parameter setting, or 10 which gives more reliable results than using only 3 folds, especially when having little training data.

Grid Search

Frequently, there are multiple parameters that have to be fine-tuned. Grid search accepts discrete list of values for each parameter and runs k-fold cross-validation on every combination to find the best setting.

Even though this method finds the optimal values from the specified set, there are two drawbacks that have to be taken into account:

- It might take long time to run if there are lots of parameter and values to be tried out or if the training set is large.
- As grid search goes only through specified parameter values, it is important to give a list of parameter values in such a granularity that the optimum does not get skipped.[1]

[1]The score could be the same for parameter setting 0.1 and 0.2 but a lot better for 0.15. In that case the granularity of parameter values was not fine enough.

Therefore, good practice is to start with exploring the parameter space with only few roughly sampled values for each parameter and then ”zooming” in to the neighbourhood of the best setting and run grid search again with a bit finer granularity of parameter values.

Random Search

Random search is an alternative to grid search which uses *hill-climbing method* known from optimization problems. It tries out various combinations of parameters from a specified range up to given number of iterations or when the improvement is smaller then specified ϵ.

The advantage of this method over grid search is that it does not have to go through all parameter settings. Also the parameter values are not limited discrete values. The disadvantage is that it is prone to ending up in local optima so multiple runs of random search might be required.

2.2 Classification Algorithms

2.2.1 K Nearest Neighbours

- uses defined distance metric
- finds k nearest training samples
- linear space and time to predict
- approximate neighbours make the prediction time asymptotically constant - e.g. Annoy[1]

2.2.2 Naive Bayes classifier

- formula
- mention Gaussian NB

2.2.3 Logistic Regression

- $x^{(i)}$ - i-th datapoint (vector)
- $y^{(i)}$ - label $\in \{0, 1\}$ of the i-th datapoint
- θ - vector of model coefficients
- $h_\theta(x)$ - prediction for x given a vector of coefficients θ
- m - number of samples
- n - number of features
- $J(\theta)$ - loss function for given θ which is to be minimized

Prediction for vector x:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Loss function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) + (1 - y^{(i)})log(1 - h_\theta(x^{(i)}))] + \frac{\alpha}{2m}\sum_{j=1}^{n}\theta_j^2$$

2.2.4 Feed-forward neural network

Activation Functions

- RelU
- Sigmoid
- Softmax

Dropout Layer

Dropout[2][3] is a regularization technique which prevents complex co-adaptations on the training set and hence reduces overfitting. Dropout layer can be put between two layers of the neural net and it drops (sets to zero) a neuron going into it with a given dropout rate p.

The higher the dropout rate, the more iterations are needed to train the network. If the dropout rate is set too high, the net might underfit the training data.

2.3 Text Analysis

Typical text analysis tasks are classifying texts into given categories or adding tags. Recently, the research moves towards tasks related to human perception of the text. One of those is *sentiment analysis* where the goal is to determine writer's attitude. For example, we might be interested in determining if a review of a product is positive or negative. Another popular task is recognizing fake news.[CITE]

Based on the task, different approaches are needed. For genre classification, both sentences

- He looked at the detective.
- He didn't look at the detective.

probably come from a detective story. The genre does not depend on if the word *look* is in a positive or negative context. The individual word choice is more important. In that case, bag of word approaches[2] perform usually good.

On the other hand, in case of sentiment analysis, these two sentences

- I didn't enjoy the movie.

[2]with tf-idf term weighting – will be introduced later

- I didn't enjoy the movie at first.

capture different sentiment as in the second case, the writer implies they liked the movie after all. To capture these nuances, the model has to keep the context of the whole sentence, which is one of the reasons neural networks with LSTMs or convolution windows win in these tasks.[CITE]

- **glossary:**
- document, term, token, corpus, class (e.g. genre, positive sentiment etc.)

2.3.1 Bag of Words

First representation we explore is *bag of words (BOW)*, which is mostly a decent baseline for text classification tasks. In the binary BOW, each document is represented by a vector of zeros and ones. The length of the vector is the same for all documents and is given by the size of *vocabulary* - list of words. It is common to turn all words to lowercase. The j-th component of the BOW vector d_i corresponding to the i-th document of the corpus is then defined as follows:

$$d_{ij} = \left\{ \begin{array}{ll} 1, & \text{if } j\text{-th word of the vocabulary occurs in the } i\text{-th document} \\ 0, & \text{otherwise} \end{array} \right\}$$

One can also store the number of occurrences instead of 0-1. However, the *tf-idf* approach, which takes not only the word frequencies, but also the uniqueness of words into account, is usually a better choice. The *tf-idf* representation is introduced in the next section.

Let us imagine a corpus consisting of 2 documents D_1 and D_2:

$$D_1 = \texttt{"Roses are red."}$$
$$D_2 = \texttt{"Violets are blue."}$$

The full vocabulary V extracted from this corpus contains the following words:

$$V = \texttt{[are, blue, red, roses, violets]}$$

In the binary *BOW* approach, documents D_1 and D_2 are then represented by vectors d_1 and d_2:

$$d_1 = \texttt{[1, 0, 1, 1, 0]}$$
$$d_2 = \texttt{[1, 1, 0, 0, 1]}$$

To keep the vocabulary meaningful, it is common to drop words with both very high and very low occurrence. Frequent words usually don't carry any meaning nor significance to the predicted classes. These words are also called *stop-words* and it is common practice to filter them out of the texts. Keeping low-occurrence words might introduce noise into the vocabulary. Therefore, only words that occur in more than d documents are added to the vocabulary.

In our example, we might exclude the word `are` from the vocabulary as it appears in every document. For the filtered vocabulary V_2

$$V_2 = \texttt{[blue, red, roses, violets]}$$

the document vectors $d1$ and d2 would change to

$$d_1 = \texttt{[0, 1, 1, 0]}$$
$$d_2 = \texttt{[1, 0, 0, 1]}$$

The filtering is highly dependent on the corpus. If all documents are novels, the word *you* probably does not help much in classifying them. However, if the goal is to distinguish novels from news articles, the word *you* could be worth keeping.

Stemming

Depending on the task, it might be not important if the word `writing` or `wrote` appeared in the text – the information that the word `write` occurred in some form could be enough. The goal of *stemming* is to transform a word to its root or infinitive form. The idea behind using stemming is that the model represents the text more robust and with less noise by mapping words derived from the same root to the same token.

The majority of stemmers are based on the initial implementation of Porter Stemmer[4] introduced by Martin Porter in 1980. It uses various heuristics to do the best suffix stripping. M. Porter also presented `Snowball` – a language for stemming algorithms.[5]

N-grams

N-gram is a consecutive sequence of n words as found in the text. They can be added to the bag of words model as additional tokens to keep a bit of information of the word order. The most widely used are *bigrams* (2-gram) and trigrams (3-gram). For $n = 1$, the n-gram is called *unigram* and it corresponds with a single word.

When using n-grams, there might be easily an explosion of tokens as bigrams and trigrams might increase the size of vocabulary V by $|V|^2$ and $|V|^3$ respectively. The vocabulary has to be then filtered in such way to keep meaningful n-grams such as `(young, woman)` or `(too, much)` and drop n-grams such as `(this, is)`, `(is, a)`, `(a, dog)` or `(this, is, a)` which are just grammar structures relating to the language but the conveying any message on type of the document, author's style or attitude.

2.3.2 Tf-Idf

In the previous approach, only the information if a word occurred in the document or not was used and the frequency of the word was ignored. One could instead create a document vector containing number of word occurrences (term[3] frequency). In order to not make the document representation dependent on the number of words in the document, document vector are then normalized. Usual approach is to use the *l2-normalization*.

[3]term in this context relates to a word, but can also be a n-gram or any type of token

Term frequency - Inverse document frequency

- based on BOW
- divides words frequency by the inverse document frequency of that word

  rare words get higher scores when they occur in the document

2.3.3 Word2Vec

Word2vec is an algorithm which learns representations of words as a vector with usually few hundred dimensions. These word vectors capture also the semantics of the words, e.g. words *detective*, *police* and *inspector* would all be represented by similar vectors. These word-embeddings can be then use in text processing as an alternative to one-hot-encoded bag of words.

Word2vec was published[6] in 2013 as the first neural word-embedding approach which gain quickly lots of popularity. Other word-embeddings algorithms followed soon after – *Global Vectors for Word Representation (GloVe)* in 2014 from Stanford University[7] or *fastText* in 2016 by Facebook[8].

The main idea behind word2vec is to train a shallow neural network on a fake task of completing a missing word out of the sequence of words. In fact, the goal is not to train a perfect classifier on ”What word is missing?” but to train a layer which can be then used as word-embedding. So even though the task of word prediction is a supervised task, the real goal of training the word vectors is unsupervised. The word2vec authors[6] propose two architectures – *skip gram* and *continuous bag of words (cbow)* as displayed in Figure 2.1.

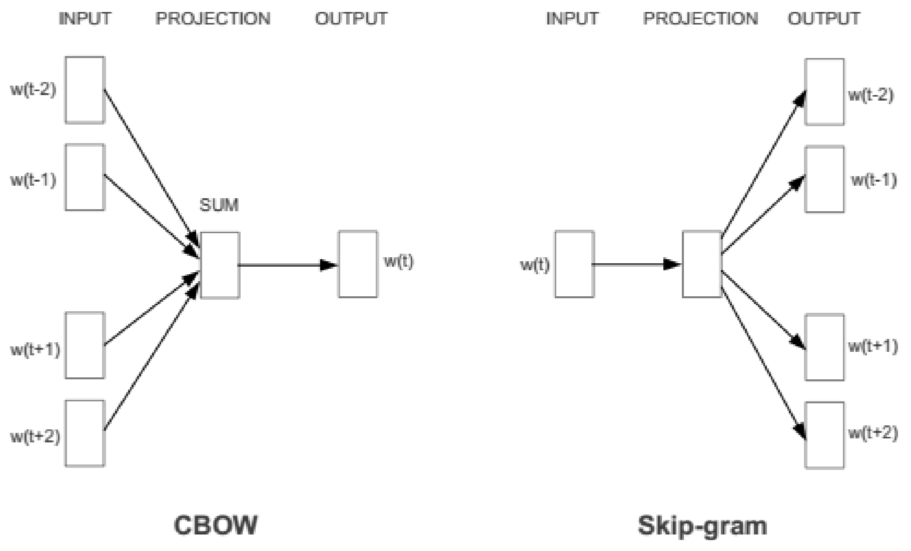


Figure 2.1: Word2vec – cbow and skip gram architecture[6]

For *skip gram*, the goal is to predict surrounding words given one word on the input. This means we want to minimize the following function:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c} log(p(w_{t+j}|w_t))$$

where $t \in T$ is a term from the corpus T, w_i the word on the position i and $(-c, c)$ the context window. As the output layer is large – the size of the whole vocabulary – the word2vec authors suggest in the original paper[6] to use hierarchical softmax[9] instead of classical softmax to compute the probabilistic distribution. Hierarchical softmax represents the output layer as a binary tree and needs only $O(log(W))$ steps instead of softmax's $O(W)$ steps to compute the probability distribution for W words. In the follow-up paper to the original word2vec, Mikolov et al.[10] propose another approach to update the weights – *negative sampling.* In negative sampling, only the positive words (expected output) and n randomly sampled negative words from the vocabulary are selected and updated. Mikolov et al. report typical n for negative sampling being 5 - 20 for smaller datasets and $2 - 5$ words for large datasets.[10]

For *cbow*, the task is the opposite – given the word's context, predict the missing word. The word vectors are projected to the projection layer using *sum* or *mean* aggregation. An example of cbow architecture is shown in Figure 2.2. The *bag of words* in the name of this architecture refers to the order of words not being taken into account. The hierarchical softmax or negative sampling are also applied in the training process.

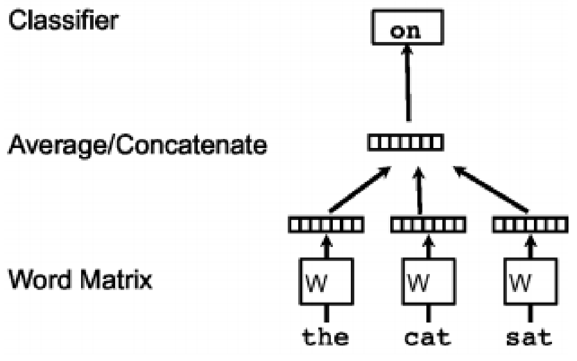


Figure 2.2: Word2vec – cbow architecture[11]

The authors of word2vec report skip-gram architecture being better for texts with infrequent words, however, much slower than cbow. It was already mentioned that semantically similar words end up being encoded by similar vectors. The intuition behind that is – if two words have similar context, well-trained word2vec has to output similar results for both words. Which means both words are encoded similarly. This has some nice application, e.g. one can perform algebraic operations on the vectors and it would correspond to the word semantics – the famous example being:[6]

$$\text{king} + \text{woman} - \text{man} = \text{queen}$$

meaning the closest vector to the one representing the expression is the vector for *queen.* As training the word vectors is an unsupervised task, similar analogies are used to test the quality of word embeddings.[10]

2.3.4 Paragraph Vector (Doc2Vec)

Paragraph vector[11], also known as *doc2vec*, comes from the authors of *word2vec.* It is a technique to represent whole paragraphs or documents as a vector using word-embeddings. Same as word2vec, doc2vec comes with two different architectures – *distributed memory*, which is an extension of the *cbow* architecture of word2vec, and *distributed bag of words*, which is an extension of the *skip-gram* architecture.

In the *distributed memory* architecture, we simply add a new vector to the input representing the document as if it were an extra word. However, unlike word vectors, which are constantly being updated, document vectors are only visible and updated if the training sample is taken from the given document. To aggregate word and document vectors, *average*, where the size of the hidden layer corresponds to the dimensionality of the word vector, or *concatenation* is used. The concatenating result is much bigger as it stacks the document and all words together – so based on the size of the window, these vectors are several times bigger then when using average aggregation. The name *distributed memory* comes from the document vector representing the memory, or better, context of the input text, to predict the missing word. The architecture is shown in Figure 2.3

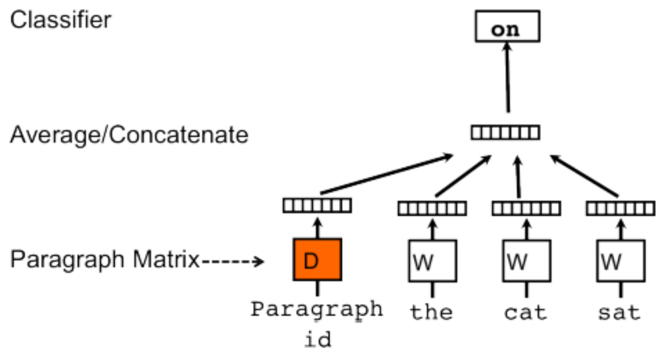


Figure 2.3: Paragraph Vector – distributed memory version[11]

The *distributed bag of words* is a simpler architecture than the previous and does not require so many parameters. However, as the name suggests, it does not utilize the order of words in the text. In the *skip-gram* word2vec architecture, the task was to predict the surrounding words based an a single words. In the *dbow* doc2vec architecture, the task is to predict the surrounding words given the document as an input. The sketch of the architecture is shown in Figure 2.4.

All techniques and parameters described for *word2vec*, such as using *context window* or optimizing using *negative sampling*, also translate to doc2vec.

2.3.5 Deep Learning

deep learning[12]

Input - sequence of word embeddings.

Recurrent Neural Network (RNN)

- cite relevant papers
- put an image of the network
- LSTM units

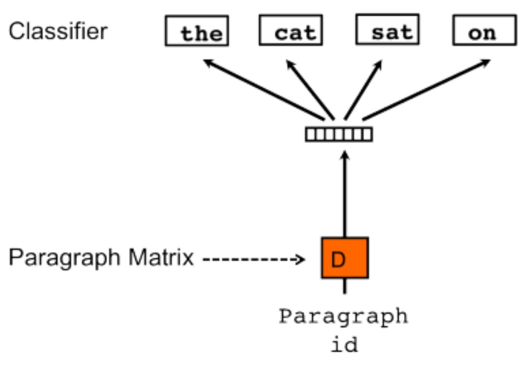


Figure 2.4: Paragraph Vector – distributed bag of words version[11]

Convolutional Neural Networks

- Show Yoon Kim's architecture[13] (Figure 2.5).
- Mention Zhang.[14].

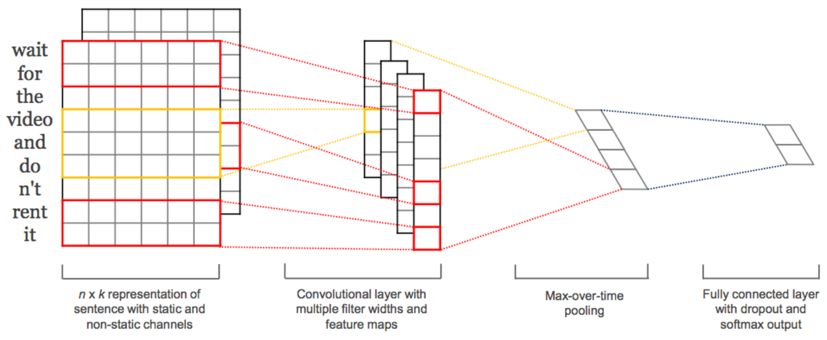

wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 2.5: Convolutional neural network with multiple filter sizes[13]

2.4 Project Gutenberg

- What is Project Gutenberg[15], 1-2 pages in total for this section
- How many books are there in Project Gutenberg
- What types (classes) of books
- Metadata for genre classification.

2.5 Glossary

To recap the terms occurring in the section, we list them all here on one place with their brief explanation of usage throughout the thesis.

- word, token

- document
- short, shorter, middle-length, long, longer documents
- model - might refer based on context to both representation (BOW, paragraph vector) and classification model
- classifier, algorithm - genre classifier
- BOW
- vocabulary, dictionary
- binary BOW
- tfidf

3. Methodology

In this chapter, we introduce the conducted experiment. After describing the dataset creation, we go into detail on the particular usage of text representation techniques and classification algorithms introduced in Section 2.1 and Section 2.3.

3.1 Dataset

There are already some widely known datasets for text classification, such as the IMDB movie review dataset which serves as a benchmark for sentiment analysis[16] or datasets for various tasks in the UCI Machine Learning Repository [17]. Nevertheless, we haven't found any publicly accessible dataset containing short text snippets of books. Therefore, we created a dataset out of the books available in Project Gutenberg. To make the dataset larger, several text snippets are cut out of each book of interest.

Genres

We rely on the *subjects* tag in the Project Gutenberg metadata catalogue to determine genres of the books. After some cleaning (e.g. merging *adventure* and *adventure stories* together), we focused on texts belonging to one of the following genres:

- adventure stories
- biography
- children literature
- detective and mystery stories
- drama
- fantasy literature
- historical fiction
- love stories
- philosophy and ethics
- poetry
- religion and mythology
- science fiction
- short stories
- western stories

We selected 5602 distinct Project Gutenberg books containing one of the above defined genres. We didn't choose books covering multiple genres as the majority of classifiers is suited for a single class predictions. Also we would have to use more complex metrics to compare multiclass classification models.

Out of the 5602 books, we sampled text snippets with the length of 3200 characters. The whole dataset consisting of 225134 documents was then split into train (85 %) and test set (15 %).

The original book texts were first preprocessed to get rid of the Project Gutenberg header and footer. After that, another 10000 characters were stripped out of the beginning and end of the book to avoid book contents, preface or glossary being part of the document. Sequences of whitespace characters were replaced by a single space character as long whitespace sequences would bloat the documents with non-meaningful symbols.

Finally, in case the original book text was split in the middle of a word, the incomplete heading and trailing word of the document is discarded, which makes the documents slightly shorter than 3200 characters.

Document size

As one of the main goals is to find out how much text is needed to distinguish a genre, we created two other datasets containing 800 and 200 characters long documents. The shorter datasets were created by taking first n characters of the original dataset with 3200 characters.[1] These three document lengths approximately represent:

- a snippet of few sentences (200 chars)
- a paragraph (800 chars)
- a couple of pages (3200 chars)

As the sizes are defined in characters and not words, the word count varies between documents.

TODO: show document examples (different lengths)

3.2 Experiment design

In the following, the methodology of the genre classification task is introduced. We represent the documents as vectors using two different methods:

- Bag of words (both binary and tf-idf weighting)
- doc2vec

For each document representation, we train and compare several genre classifiers. The whole experiment is done for 3 datasets – documents with the length of 200 characters, 800, and 3200 characters.

[1]And again, discarding the last word in case the word was not complete.

3.2.1 Tokenization

Bag of words and doc2vec both require different input. Therefore, document texts have to be processed in two different ways based on which representation is used.

Tokens for bag of words

A token for bag of words is usually a sequence of letters split by whitespace characters or punctuation. Punctuation itself is not included as a token. In our implementation, we also don't include numbers or special symbols.

Tokens for doc2vec

The original authors of doc2vec split the whole sentence into tokens without discarding anything, which means punctuation and numbers are also considered as a token. To filter out noise created by numerals inclusion, tokens which appear in fewer documents than a given threshold are skipped.

3.2.2 Bag of Words

First, we represent documents as bag of words. One of the drawbacks of BOW is that it creates vectors in highly-dimensional space. That might cause problems in training as the whole dataset might not fit into memory or it can take very long time until some classifiers, for example SVMs, converge.

To see how many distinct words are needed in the BOW vector for a good prediction, we consider various vocabulary sizes from 1000 to 50000 words and compare the classification scores.

When creating vocabulary with size n, the n most frequent words which occur in less than 50 % of the documents are chosen. At the same time, chosen words most appear at least in 5 documents to be considered at all. Filtering of the frequent words is more or less equivalent to stop word exclusion. By filtering the low occurrence words, we make sure that words such as names very specific to a given book are not included in the dictionary.

For BOW, We train following classifiers:

- naive Bayes (binary vectors)
- logistic regression (binary vectors)
- logistic regression (tf-idf vectors)
- feed-forward neural net (binary vectors)

For the logistic regression classifiers, optimal regularization strength parameter α is found through 3-fold cross-validation.

3.2.3 Doc2Vec Representation

Doc2vec has many parameters which have to be fine-tuned for the model to work good on our domain. The parameter search is done independently for all 3 document lengths.

We use slightly modified version of doc2vec where there is not only a vector for each document, but also a vector for each genre (see Figure 3.1).[2]

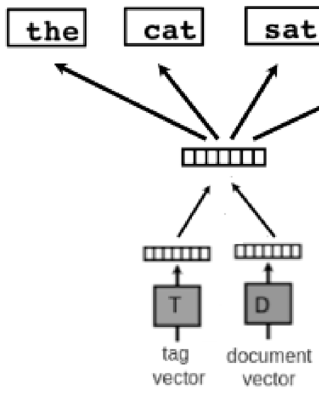


Figure 3.1: Doc2vec (distributed BOW) with tag vector

On the doc2vec representation, we compared following algorithms:

- most similar genre vector
- most similar book vector (KNN)
- Gaussian naive Bayes
- logistic regression
- SVM with linear kernel
- feed-forward neural net

[2]This kind of vector is usually called tag

4. Data Exploration

In the following, we explore some elementary properties of the 225134 documents in our dataset. The linguistic properties are made on the full-length documents (3200 characters).

4.1 Metadata exploration

4.1.1 Genre Distribution

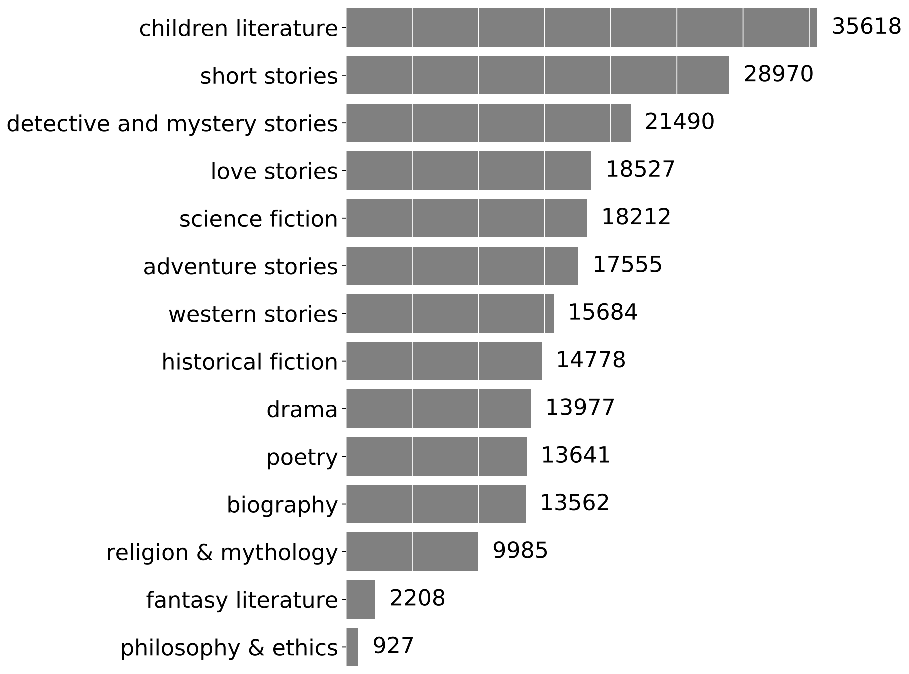


Figure 4.1: Genre distribution among documents.

4.1.2 Author distribution

- How many authors
- Snippets per author distribution
- Main authors

4.1.3 Year

Year defined based on day of birth of the author.[1]

[1]publish date of book not available in metadata catalogue

4.1.4 Most downloaded books

4.2 Text exploration

4.2.1 Average Word Length

- How do we define word
- Distribution per genre

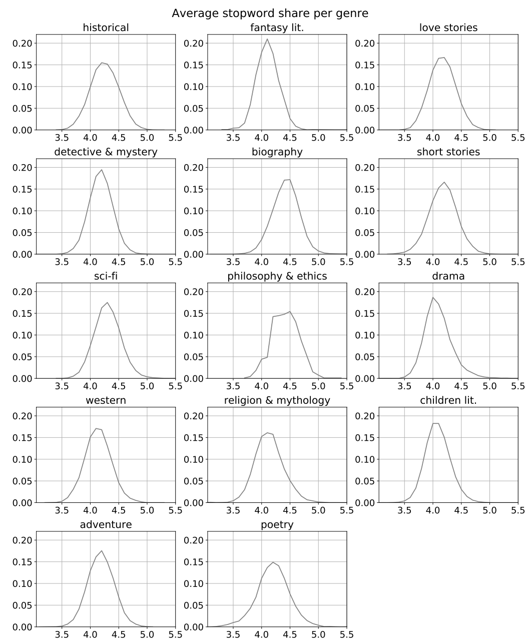
4.2.2 Average Sentence Length

Defined as number of words in a sentence.

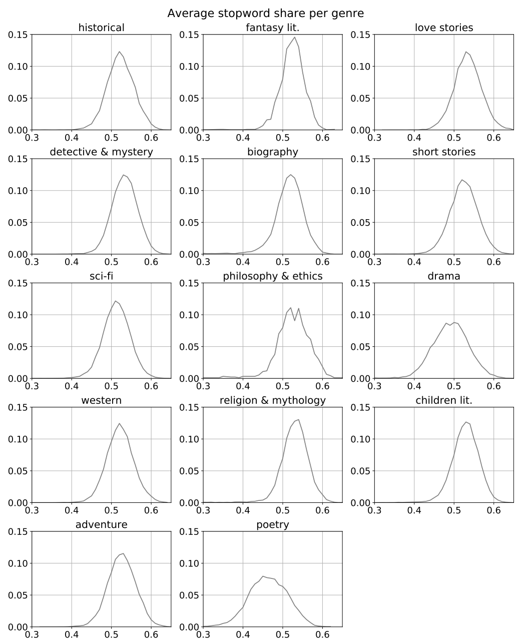
- How do we define sentence
- Distribution per genre

4.2.3 Stop Words Proportion

Stop words definition as in `nltk.corpus.stopwords`. Examples of stopwords.
TODO: Change the plot to two 2x7 plots.



(a) Average word length per genre.



(b) Average stopwords share per genre.

Figure 4.2: Average word length and stop word share distribution per genre.

5. Evaluation

In this chapter, we train various genre classifiers on three different datasets – documents containing 200, 800 and 3200 characters – and compare the prediction performance.

First, we represent documents as *bag of words* and explore the influence of factors such as *size of vocabulary* or *stemming*. Next, we represent documents as vectors in a lower[1] dimensional space using *doc2vec* and discuss the choice of hyperparameters.

For both representations, we explore if the performance for shorter documents improves when using models trained on long documents. Finally, we compare both representations and stack several classifiers to improve the result even further.

Apart from that, we look at some misclassified documents and confusion matrices to understand the models and their wrong predictions a bit better.

5.0.1 Evaluation metric

As the classes are not balanced (there are 35618 *children literature* documents but only 927 *philosophy and ethics* documents), we optimize *F1-macro* score instead of accuracy. The F1-score for multiple classes with macro weighting is defined as follows:

$$F_1 = \frac{2PR}{P+R}$$

where P and R are precision and recall averaged over all classes $C_i \in C$ with equal weight[2].

$$P = \frac{1}{|C|}\sum_{i=1}^{|C|} P_i$$

$$R = \frac{1}{|C|}\sum_{i=1}^{|C|} R_i$$

P_i and R_i are then standard precision and recall defined for single class i:

$$P_i = \frac{TP_i}{TP_i + FP_i}$$

$$R_i = \frac{TP_i}{TP_i + FN_i}$$

where TP_i, FP_i and FN_i are number of true positive, false positive and false negative predictions for class i.

Precision describes how often was the classifier right when predicting class i. *Recall*, on the other hand, captures how often did the classifier predict class i for documents of class i.

[1]By *lower* we mean few hundred dimensions which is a lot smaller than in the *bag of words* representation where each word had its own dimension.

[2]Which means a misclassification in smaller classes changes the score more than a misclassification in bigger ones.

To illustrate the computation of the F1-score with *macro* weighting, we compute the test set score for a baseline class predictor which blindly predicts the majority class for every document:

- 33771 documents in the test set
- 14 genres in total
- the majority class is *children literature* with 5314 occurrences

For all genres g except for *children literature*, the true positive rate TP_g is equal to 0 as the predictor never classifies a document in that class. That means that precision P and recall R for those classes are 0.

For the *children literature* class the precision and recall are

$$\begin{aligned} P_{children\ literature} &= \frac{5314}{33771} = 0.1574 \\ R_{children\ literature} &= \frac{5314}{5314+0} = 1 \end{aligned} \tag{5.1}$$

as there was no misclassified *children literature* document.

With macro weighting, we get overall precision and recall as

$$\begin{aligned} P &= \frac{1}{14}(13 \cdot 0 + 1 \cdot 0.1574) = 0.0112 \\ R &= \frac{1}{14}(13 \cdot 0 + 1 \cdot 1) = 0.0714 \end{aligned} \tag{5.2}$$

Finally, the F1-macro score is then

$$F_{1-macro} = \frac{2PR}{P+R} = \frac{2 \cdot 0.0112 \cdot 0.0714}{0.0112 + 0.0714} = 0.0194$$

The accuracy, at the same time, is $\frac{5314}{33771} = 0.1574$ which is much higher than the F1-score as the metric does not take class sizes into account. To train the classifier to perform well on all 14 classes, we optimize the F1-score and only sometimes mention accuracy for comparison.

5.1 Bag of Words

First, we represent the documents as a *binary* bag of words and explore how does the vocabulary size influence the classification performance. As the time and space complexity of the model training are dependent on the vocabulary size, we want to know if the added complexity brings any boost in performance. We explore vocabulary sizes from 1000 up to 50000 words.

When limiting the vocabulary size to n words, we select the n most frequent words which appear in less than 50 % of all documents. These words must also occur in at least 5 distinct documents to be considered at all. For short documents (200 characters), only score for vocabulary up to 30000 words is shown as the performance stays constant for bigger vocabulary sizes. The reason for that is that if we filter out words occurring in less than 5 documents, there are only ca. 32000 words left.

In the following, we compare the classification performance of naive Bayes classifier, logistic regression and feed-forward neural network with two hidden layers.

5.1.1 Naive Bayes

The naive Bayes classifier turned out to be a very decent model for short documents where it reached the same result as logistic regression while having training time under 1 second[3]. The performance of naive Bayes classifier improved with increasing vocabulary size as shown in Figure 5.1. However, including the low frequent words for short and middle-length documents does not improve the performance too much.

The best F1-score and accuracy for all three document lengths are listed in Table 5.1.

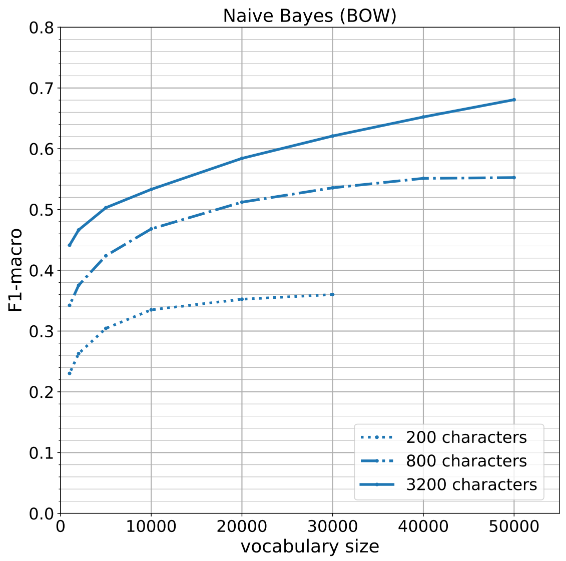


Figure 5.1: F1-macro score comparison for Naive Bayes based on text length and vocabulary size.

Document length	F1-macro score	Accuracy
200 characters	0.360	0.413
800 characters	0.553	0.572
3200 characters	0.681	0.678

Table 5.1: Best performance of **Naive Bayes** on **binary BOW** for each document length.

[3]Training was done on a single core on a computer with 2.5 GHz Intel Core i7 CPU

5.1.2 Logistic Regression

Logistic regression on binary BOW performed better than Naive Bayes for short and medium-length documents. Figure 5.2 shows the F1-scores for all tested vocabulary sizes and Table 5.2 lists best results of the Logistic Regression for each document length.

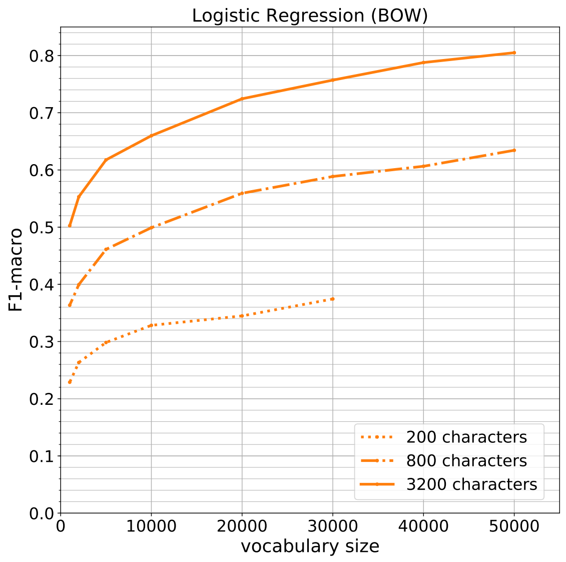


Figure 5.2: F1-macro score comparison for Logistic Regression with binary BOW representation based on text length and vocabulary size.

Document length	F1-macro score	Accuracy
200 characters	0.374	0.398
800 characters	0.634	0.637
3200 characters	0.805	0.802

Table 5.2: Best performance of **Logistic Regression** on **binary BOW** for each document length.

For vocabulary size greater than 10000, training with all 191363 documents does not fit into 16 GB of RAM. Therefore, we used the *stochastic gradient descent* with logistic loss implemented in `sklearn` which corresponds to the logistic regression.

The best logistic regression classifier on binary BOW trained on long documents with vocabulary containing 50000 words reached F1-score 0.805 and accuracy 0.801.

For each vocabulary size, *grid search* was used to find the best regularization strength parameter α. Generally, the optimal α value increased with document size and decreased with the size of vocabulary. The optimal α for the best classifier on long documents was 0.0003.

5.1.3 Feed-forward NN

Next classifier we tried out for the binary BOW representation was a simple feed-forward neural network with 2 hidden layers of 200 and 100 neurons. We used *ReLU* as an activation function for hidden layers, and *softmax* on the output layer.

To decrease overfitting of the net, dropout layers were added between the layers with following dropout rates:

- **0.4** between input and first hidden layer with 200 neurons
- **0.1** between first hidden layer with 200 neurons and second with 100 neurons

Figure 5.3 shows the architecture of the net.

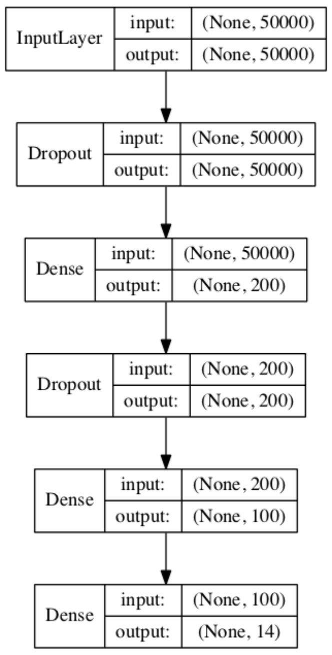


Figure 5.3: Feed-forward NN architecture for 50000 words in vocabulary

In Figure 5.4 we can see that the F1-score of the feed-forward neural network improves with increasing vocabulary size as did the previous two algorithms. Even between the vocabulary size of 40000 and 50000, there is still about 0.02 score difference.

The best F1-score achieved for long documents was 0.849 which is about 0.04 better than the result of logistic regression. That comes as no surprise as logistic regression[4] is equivalent to neural network with softmax output layer activation and no hidden layer.[5] As our net has two hidden layers, it is then computationally stronger than logistic regression. The best results for all document lengths are shown in Table 5.3.

For long documents, the net overfitted the training data massively and reached accuracy score on the training set of 99 %. One way to deal with the overfitting is to increase the dropout rate on the input layer. Another possibility is to decrease

[4]when using one-vs-all approach

[5]except for regularization

the number of neurons in the hidden layer. However, this kind of optimization requires lots of time and computational power and would most likely not bring us more than about 0.5 % improvement on the score.

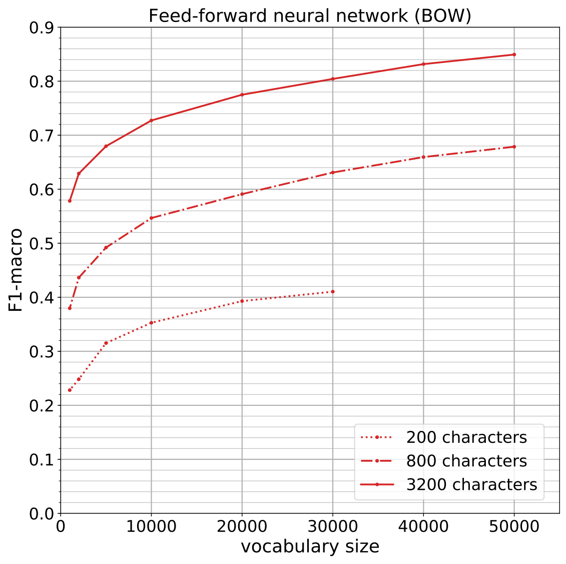


Figure 5.4: BOW with feed-forward NN classifier.

Document length	F1-macro score	Accuracy
200 characters	0.410	0.429
800 characters	0.679	0.680
3200 characters	0.849	0.850

Table 5.3: Best performance of **feed-forward NN on binary BOW** for each document length.

5.1.4 Tf-Idf

Until now, the feature vector for each document was only binary. Nevertheless, it performed quite decently. In the following, we represent documents as tf-idf vectors utilizing both the number of occurrences (*term frequency*) and uniqueness of each word (*document frequency*[6]).

To compare with the binary approach, we train logistic regression on the tf-idf vectors. As shown in Table 5.4, the F1-score for the tf-idf vectors was around 0.03 better than when using binary vectors. Figure 5.5 shows that the classifier can make use of more words in the vocabulary as the score for all three document lengths is increasing with the size of vocabulary.

Similarly to the logistic regression on binary BOW, the regularization parameter α decreased with increasing size of vocabulary. The optimal α value for the best models of each document lengths turned at to be the same – 10^{-6}.

[6]i.e. in how many documents did the word occur

Document length	binary BOW	tf-idf
200 characters	0.374	**0.395**
800 characters	0.634	**0.663**
3200 characters	0.805	**0.836**

Table 5.4: F1-score comparison for logistic regression trained on binary BOW vectors vs. tf-idf vectors.

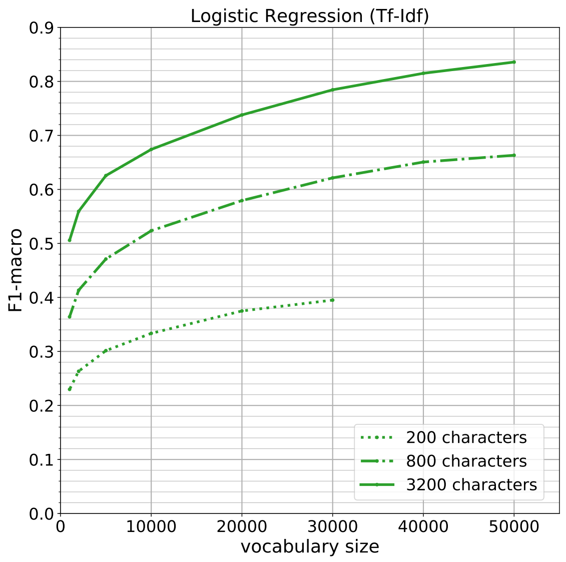


Figure 5.5: F1-macro score comparison for Logistic Regression on tf-idf vectors.

5.1.5 Stemming

To make the vocabulary more robust and less noisy, we tried out *stemming* the words using `SnowballStemmer` provided in `nltk`. As shown in Table 5.5, stemming didn't significantly improve performance for any of the models.

	200 ch.	200 ch. stemming	800 ch.	800 ch. stemming
Multinomial NB	0.362	0.361	0.553	0.552
logistic reg. BOW	0.365	0.362	0.622	0.626
logistic reg. tf-idf	0.404	0.395	0.663	0.663

Table 5.5: F1-score comparison for shorter documents with and without using Snowball Stemmer.

5.1.6 Using models trained on long documents

So far, to classify documents with a given length, we trained a model on documents with that same length. The question is if we can get better score for shorter documents using models trained on longer documents. First, we use the vocabulary from the 3200-character documents and train the genre classifiers for 200 and 800-character documents on this vocabulary. After that, we explore if also the classifiers trained on long documents also work on shorter texts.

Vocabulary

When using vocabulary extracted from long texts, the expectation was to obtain better results as the vocabulary should contain more relevant words and less noise.

However, as shown in Table 5.6, we cannot see any improvement for neither of the classifiers – on documents with both 200 and 800 characters.

The cause for this result is that when vocabulary chooses top n words based on frequency in the corpus, words with higher frequency in the corpus of long documents are preferred to those with high frequency in the training set of short documents. So it might happen that a word that is somewhat frequent in the training set and would be a good feature ends up not being in the vocabulary because it didn't occur often enough in the much bigger corpus. Instead, a word very rare in the training set (but somewhat frequent in the corpus) is picked for the vocabulary.

	200 ch.	200 ch. w vocab	800 ch.	800 ch. w vocab
Multinomial NB	0.362	0.364	0.553	0.557
logistic reg. BOW	0.365	0.364	0.622	0.626
logistic reg. tf-idf	0.404	0.403	0.663	0.654

Table 5.6: Comparison of classification model performance when trained on the vocabulary extracted from the 200 or 800-character documents vs. when trained on the vocabulary of long documents (*w vocab*).

Models trained on long texts

Next, we explore the performance when classifying shorter documents using models trained on 3200-character documents. Some models might be invariant to document length and hence work also on the short documents.

In Table 5.7, we can see that the F1-score deteriorated for logistic regression on both binary BOW and tf-idf. However, multinomial naive Bayes classifier trained on long documents proved to be working also on shorter ones with 0.09 improvement on 200-character documents and 0.04 on 800-character documents. For short documents, this approach beats all other BOW approaches.

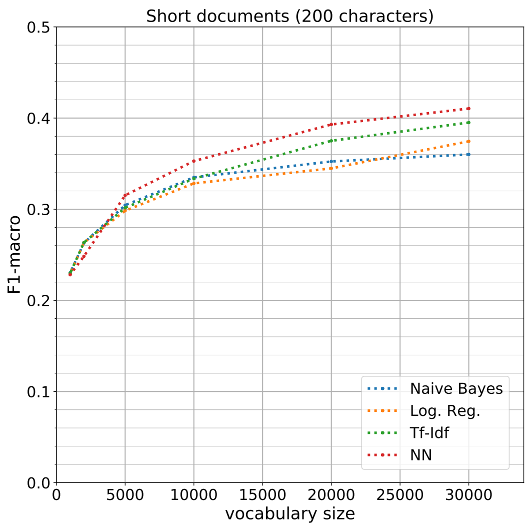
	200 ch.	200 ch. w model	800 ch.	800 ch. w model
Multinomial NB	0.362	**0.451**	0.553	**0.593**
logistic reg. BOW	0.365	0.280	0.622	0.581
logistic reg. tf-idf	0.404	0.351	0.663	0.647

Table 5.7: F1-score when predicting for shorter documents using models trained on long documents.

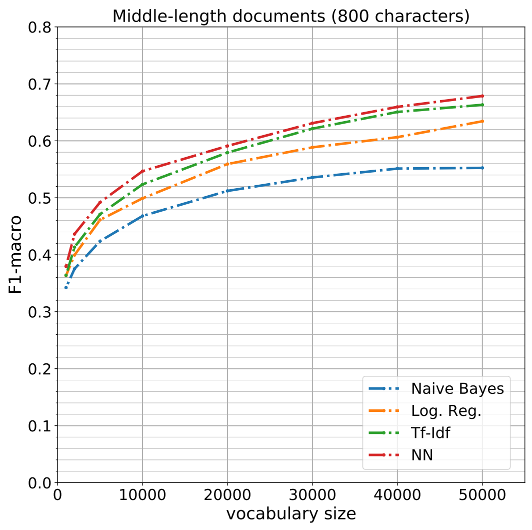
5.1.7 Summary BOW

All in all, the score improved with growing vocabulary size for all algorithms and document lengths. Figure 5.6 shows comparisons of all BOW algorithms on each document length. The neural network performed the best for both 800 and 3200-character documents, only for the short documents, naive Bayes classifier trained on the long documents performed better than the neural network.

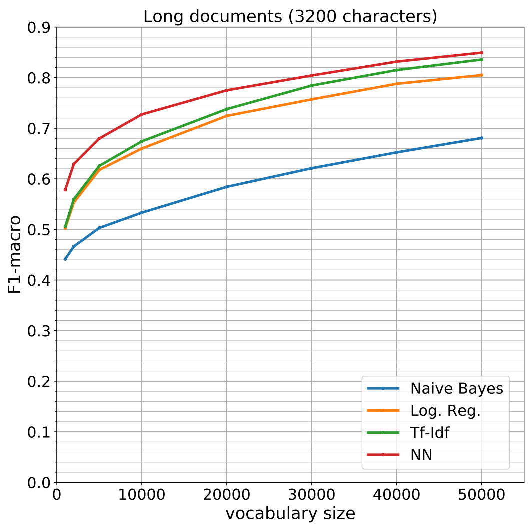
Logistic regression with tf-idf vectors was just slightly worse than the neural networks. Logistic regression performed better on tf-idf than on binary BOW and the gap increased with growing size of vocabulary. It is probably caused by tf-idf boosting the rare words[7] that are typical for a given genre.[8] For longer documents, naive Bayes classifier performed the worst.



(a) 200 characters



(b) 800 characters



(c) 3200 characters

Figure 5.6: F1-macro score comparison for all BOW algorithms.

[7]words not in top 10000 most common words in the corpus

[8]For example the word *asteroid* occurred 54 times in science fiction genre and never in any other genre.

Table 5.8 shows the comparison of all algorithms and document lengths. The neural net reached F1-score of 0.849 and accuracy 0.850. The logistic regression on tf-idf performed was worse only by 0.013 which is negligible given a lot higher training complexity and space needed to fit and store the neural net.

Document length	Naive Bayes	Log. reg.	Log. reg. (Tf-Idf)	NN
200 chars	0.360 (**0.451**)*	0.374	0.395	0.410
800 chars	0.552 (0.593)*	0.634	0.663	**0.679**
3200 chars	0.681	0.805	0.836	**0.849**

Table 5.8: F1-score comparison of classifiers on BOW document representation for each document length. The symbol * marks algorithms trained on 3200-character documents.

5.2 Doc2vec

In the next approach, documents are embedded into a space of several hundred dimensions using *doc2vec*. This representation is a lot smaller and compacter than the previous BOW approach where documents were represented by vectors with up to 50000 dimensions.

For the document classification, we used similarity metrics to find most similar documents (kNN) or most similar genre vector. Apart from those, logistic regression and simple neural network with one hidden layer containing 50 neurons were used.

The training of doc2vec was done using the python module `gensim`[18] and a big hyperparameter search had to be done to make the approach work for our task. The main parameters we had to tune were:

- choosing between *distributed bow* and *distributed memory* architecture
- *dimension* of the vectors
- *window size*

5.2.1 Hyperparameter tuning

For the initial parameter setting, we adopted the parameters from J. H. Lau and T. Baldwin - *An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation*[19]. They also report improvement when initializing the doc2vec model with word embeddings trained on bigger corpus.

In order to choose the right hyperparameters, we have to find a way to compare trained doc2vec models. As we included also the genre itself as a tag when training doc2vec (see Section 3.2.3), doc2vec trains *genre vectors* as a by-product. The quality of the doc2vec model can be estimated by comparing the inferred documents from the validation set with the vectors for the genre tags and computing how often was the document vector closest to the vector representing its genre out of all 14 genre vectors using cosine similarity.

Distributed BOW (dbow) vs. Distributed Memory (dm)

Le & Mikolov, the original authors of the Paragraph vector[11], propose two architectures.

The first one is *distributed memory (dm)* where the task is to predict a missing word from the window given the context (surrounding words) and the paragraph vector.

The second architecture is *distributed bag of words (dbow)* where the net is trained to predict words in a small window given the document vector.

Le & Mikolov report distributed memory version to perform better.[11] However, Lau and Baldwin[19] as well as the creators of gensim[18] observed the distributed BOW version to obtain better results.

In our experiments, we join the latter as the distributed BOW version reaches 0.05 to 0.1 better score on the genre classification task than the distributed memory architecture.

Following hyperparameter discussion focus then on the **dbow** architecture of doc2vec.

Vector dimension

Le & Mikolov used vectors with 400 dimensions in their original work[11], Lau and Baldwin[19] chose 300 dimensions.

For our task, number of dimensions between 300 and 400 worked the best as well. The performance did not improve for vector sizes bigger than 500 and for 200 dimensions or less, the performance started decreasing.

Window size

For all document lengths, we reached best performance with the window size 1. Larger windows had marginally worse results and needed longer time to train the doc2vec model.

Including genre vectors

As already mentioned, the doc2vec model was trained using two kinds of tags:

- document tag unique for each document
- genre tag corresponding to the document (14 in total)

When using this architecture compared to using only document vectors, the F1-score improved a lot – from about 0.05 for short documents to 0.1 for longer documents.

Not only did the score improve, but now we also have genre vectors that can be used themselves as a classifier – *nearest genre vector* classifier.

To build on success of *genre tags*, we tried also including *book tags* as multiple documents were sampled from the same book. However, the improvements on F1-score for both *nearest genre vector* classifier and *logistic regression* were negligible.

Pre-trained word embeddings

Lau and Baldwin report improvement when using pre-trained word vectors.[19] For our task, using *GloVe* vectors with 300 dimensions trained on Wikipedia improved the score as well. The improvement was more significant for predictions based on short documents. That comes as no surprise as short documents contain less data to train a good word-embedding than longer documents.

Learning rate α

The default learning rate α in gensim is 0.025 which turned out to be too large for our setting. We observed α between 0.0075 and 0.015 and multiplying α by 0.8 at the end of each epoch delivering the best performance.

Document reshuffling

Another thing that improves the document vector quality is reshuffling of training samples at the beginning of each epoch. Although considered as standard for neural net training, reshuffling is not automatically supported by gensim[9] and has to be done manually. Doing so constantly improves the score by couple of percent points.

Vector inference for new documents

When inferring a vector for a given document, the accuracy was slightly better when using only 3 infer steps instead of default 5 in gensim.

Roundup

To sum up, Table 5.9 gives on overview of the best parameter setting used in the experiment. Better performance was reached when using pretrained *GloVe* vectors, adding tag for every genre and shuffling the data after each epoch.

Hyperparameter	value
architecture	distributed BOW
vector dimension	300
window size	1
infer steps	3
learning rate α	short docs: 0.015
	rest: 0.0075

Table 5.9: Optimal setting of hyperparameters for doc2vec model.

5.2.2 Cosine similarity with genre vectors

After having trained the doc2vec model, we first check what is the relation between the inferred document vector and genre vectors learned during training. The classifier *nearest genre vector* chooses the nearest genre vector to the document vector using cosine similarity.

The big advantage of this approach is that it is very efficient in terms of speed and storage. It requires storing only 14 genre vectors additional to the doc2vec model and the prediction is done by computing 14 cosine similarities with no training required (except for the doc2vec model, of course). Given the simplicity, the *nearest genre vector* classifier already achieves a decent score. As inferring the document vector is not a deterministic task, we can get n different vectors for a document by inferring the document vector n times. Then we can find the nearest genre vector to each of the n document vectors. Finally, we predict the genre occurring most frequently. The F1-score for both single infer and multiple infers can be seen in Table 5.10.

The improvement by multiple inferring is more significant for the short documents where it improved by 0.032 whereas for the long ones only by 0.017.

[9]At least not at the time of writing this text – June 2018

Document length	1 nearest genre	1 nearest genre (20 infers)
200 chars	0.342	0.374
800 chars	0.587	0.609
3200 chars	0.748	0.765

Table 5.10: Nearest genre vector based on cosine similarity between genre and document vectors.

5.2.3 K most similar documents (KNN)

One can also look at the similarities between the given document and other documents in the training set. The genre of the document can be than predicted as the most common class among the *k* most similar documents. This is basically nothing else than *k nearest neighbours* classifier using cosine similarity.

The optimal *k* for each document length was chosen using cross-validation and can be seen in Table 5.11. The *k* decreases with growing document length. This signals that for long documents, model can rely on 9 most similar documents being relevant (i.e. having the same genre) whereas for shorter documents, the most similar documents are not that relevant and larger neighbourhood of 32 documents is needed.

Unlike the previous classifier, where only 14 genre vectors had to be stored and compared with, this approach requires documents of the whole training set to be stored and compared with.

Document length	k	F1-macro score
200 chars	32	0.301
800 chars	14	0.512
3200 chars	9	0.817

Table 5.11: K nearest documents using cosine similarity.

5.2.4 Logistic Regression

Similarly to the BOW representation, we train logistic regression on the document vectors. As shown in Table 5.12, logistic regression on doc2vec vectors performed slightly worse than when using tf-idf representation. The gap is bigger for short documents.

Document length	doc2vec	BOW (50k words)	BOW tf-idf (50k words)
200 chars	0.362	0.374	0.395
800 chars	0.640	0.634	0.663
3200 chars	0.829	0.805	0.836

Table 5.12: F1-score comparison using logistic regression.

5.2.5 Feed-forward NN

In the original doc2vec paper[11], the authors report feed-forward neural net with 50 neurons performing better than logistic regression on doc2vec vectors.[10]

It holds also for our task – one hidden layer with 50 neurons and *RelU* activation function performed the best. To reduce overfitting, 0.2 dropout probability was added between the input and hidden layer. Larger hidden layers increased overfitting on the training set and didn't improve the performance for validation nor test set.

Table 5.13 compares the F1-score of the neural network with the score of neural network on BOW and logistic regression on doc2vec representation. The neural net on doc2vec performs slightly worse than neural net on BOW with bigger gap for shorter documents, which is the same behaviour we observed for logistic regression.

Document length	doc2vec - NN	doc2vec - logistic reg.	BOW - NN
200 chars	0.373	0.362	0.410
800 chars	0.652	0.640	0.679
3200 chars	0.841	0.829	0.849

Table 5.13: Comparison of F1-score for feed-forward nets trained on BOW and doc2vec as well as logistic regression trained on doc2vec vectors.

5.2.6 Using doc2vec trained on long documents

For the *BOW* representation, we tried improving the performance for *shorter documents* by using vocabulary and models trained on *long documents*. Except for multinomial naive Bayes classifier, it did not improve the performance significantly. For doc2vec, we try something similar.

The idea is to use the doc2vec model *trained on long documents* to *infer* document *vectors for shorter documents*. On the inferred document vectors, we can again train all the classification models discussed above and compare the performance.

Table 5.14 shows the difference in score for genre prediction on shorter documents when using doc2vec model trained on 3200-character document compared to doc2vec model trained only on 800 and 200-character documents respectively. All classifiers performed better when doc2vec was trained on more text. Average improvement for middle-length documents was slightly above 0.03 whereas improvement for short texts was 0.08. The neural net keeps performing the best.

When directly using the *classification models* trained on long documents for genre prediction on shorter documents, the performance was worse for all of them.

5.2.7 Summary doc2vec

All in all, doc2vec had slightly worse performance than BOW. However, when we used the doc2vec model trained on the long documents also for shorter documents, doc2vec mostly outperformed BOW. This indicates that even though the training

[10]The classification task in that article was *sentiment analysis* on IMDB film reviews.[16]

algorithm / document length	200 ch.	200 ch. (long)
1 nearest genre	0.316	0.404
1 nearest genre (20 infers)	0.374	0.439
k nearest documents (k=41)	0.274	0.337
logistic regression	0.355	0.430
neural network	0.357	**0.446**
	800 ch.	800 ch. (long)
1 nearest genre	0.587	0.610
1 nearest genre (20 infers)	0.609	0.635
k nearest documents (k=17)	0.512	0.561
logistic regression	0.640	0.671
neural network	0.652	**0.691**

Table 5.14: Comparison of classification models for shorter documents when doc2vec trained on shorted documents (*first column*) versus when trained on 3200-character documents (*second column*).

set contained almost 200000 documents, doc2vec could still benefit from extra text.

Table 5.15 lists all classifiers trained for each document length. For doc2vec on 200 and 800-character documents, doc2vec trained on long documents is used. The same applies to the multinomial naive Bayes classifier which was trained on the long documents. For 1 nearest genre classifier, we show only the version with multiple infers abbreviated as *1NG (20)*.

representation	algorithm	200 chars	800 chars	3200 chars
BOW binary	Multinomial NB	**0.451**	0.593	0.681
BOW binary	logistic regression	0.365	0.622	0.805
BOW binary	neural network	0.410	0.679	**0.849**
BOW tf-idf	logistic regression	0.404	0.663	0.836
doc2vec	1NG (20)	0.439	0.635	0.765
doc2vec	*k* nearest docs	0.337	0.561	0.817
doc2vec	logistic regression	0.430	0.671	0.829
doc2vec	neural network	0.446	**0.691**	0.841

Table 5.15: F1-score comparison of all BOW and doc2vec classifiers.

5.3 Stacking

Finally, we can make use of the already trained classifiers to achieve the best result using the stacking approach – training another classifier on the predictions of those classifiers. In this case, we don't use the single class prediction of the classifiers but the probabilities[11] for each genre to utilize the confidence of each classifier in their prediction.

The classifiers to be stacked are:

- multinomial naive Bayes (BOW binary)[12]
- logistic regression (BOW binary)
- logistic regression (tf-idf)
- 1 nearest genre (doc2vec)
- logistic regression (doc2vec)

For each document in the training set, we use these 5 classifiers to predict genre probabilities. This converts each document into a vector with 70 dimensions[13]. Then we train a neural network using this 70-dimensional vector as input. To make predictions for the documents in the test set, we have to first make predictions using the 5 algorithms to create the input vector for the stacking classifier. Final prediction is then the prediction made by the neural network with this vector as the input.

The best architecture differed a bit based on the length of the documents as shown in Table 5.16. For the 200 and 800-character documents, the best performance was reached using 14 hidden neurons. For long documents, however, the neural network without any hidden neurons performed marginally better.

document length	hidden neurons	dropout rate on input layer
200 char	14	0.4
800 char	14	0.3
3200 char	0	0.2

Table 5.16: The best parameter overview for the stacked neural network.

The neural network with stacking several classifiers performs according to our expectations better than single classifiers. Table 5.17 shows the comparison of the stacked F1-score with the best result of a single classifier. The score improved by 0.03 for shorter documents, for 3200-character documents, the improvement was only 0.013.

[11]i.e. the method `predict_proba()` in most APIs

[12]The multinomial naive Bayes trained on 3200-character documents is taken.

[13]Each of the 5 classifiers outputs a probability score for each of the 14 genres.

document length	best classifier	F1-score	stacking NN F1-score
200 char	Multinomial NB	0.451	**0.479**
800 char	NN (doc2vec)	0.691	**0.721**
3200 char	NN (BOW)	0.849	**0.862**

Table 5.17: Comparison of the best F1-score reached by a single classifier and the stacked neural net classifier.

5.4 Error analysis

Select few documents for logistic reg. tf-idf and logistic reg./cosine sim for doc2vec that were confidently assigned to another genre and look at their text. Does it make sense for a human that they were misclassified?

- Put confusion matrix and discuss

6. Insights

6.1 Typical Words

6.1.1 Tf-Idf

How did we define typical words for tf-idf:

- define tf-idf genre vectors by averaging all document vectors of given genre (on training set)
- compute an average tf-idf vector of the whole training set
- subtract the average vector from each genre vector
- The most important words are those with highest scores
- Based on desired output, filter out too short words (having only two characters) and rare words (e.g. keep only those which occurred at least in 0.5 % of documents)
- investigate words with highest and lowest variance in tf-idf coefficient among the 14 genre vectors

6.1.2 Doc2Vec

- look at words most similar to the trained genre vectors

As we trained also word embeddings for the doc2vec model, we can look at similarities between a document and a word.

Next, we compute similarities between the genre vector and all words in the vocabulary. As we want to get representative words of the genre, we filter out uncommon words which occurred in less than 0.5 % of all documents. We also focus only on words consisting of at least 4 letters, as shorter words seem to have higher similarity to all documents in general when computed based on dot product. Figure 6.1 shows a word cloud of most typical words for *detective and mystery stories*, *science fiction* and *western stories*.

When using different similarity metrics, the typical words didn't seem too meaningful (would require further filtering).

6.2 Document similarity

Choose a document and find most similar documents. Look at accuracy @ 1, 3, 5, 10 documents for the following:

- Are they part of the same book?
- Same author?
- Same genre?

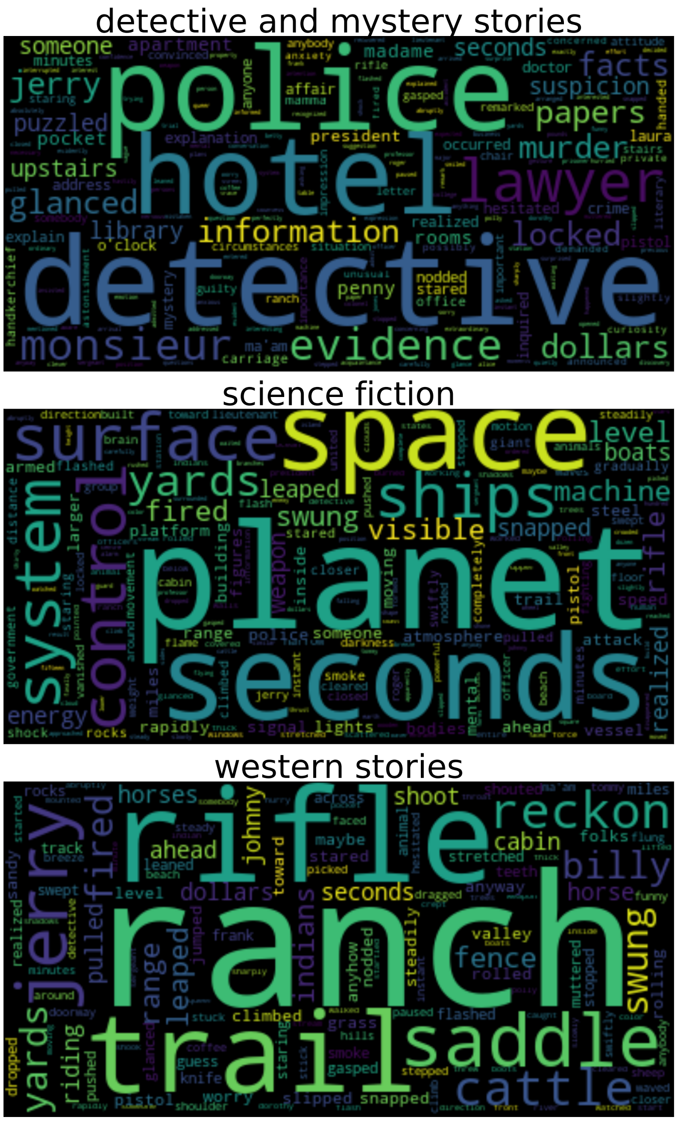


Figure 6.1: Typical words for *detective and mystery stories*, *science fiction* and *western stories* based on doc2vec and dot product document-word similarity.

6.3 Visualizing document vectors

To visualize documents in 2D, we transformed the document vectors to 2 dimensions using dimensionality reduction techniques. The final plot is made by t-SNE algorithm. However, as the convergence takes very long for large datasets, we first run PCA and transformed the data to 50 dimensions and run t-SNE on those.

7. Implementation

Implementation was done in python. Notebooks available in github repo.

7.1 Used modules

7.1.1 gensim

Gensim[18]. provides implementation of various models for text processing and analysis.

- tf-idf
- word2vec
- doc2vec

7.1.2 sklearn

Central python module for machine learning. We used:

- classification algorithms & feature engineering
- dimensionality reduction – PCA and TSNE

7.1.3 keras & tensorflow

7.2 Live demo

- Text classifiers deployed to heroku

  https://book-genres-prediction.herokuapp.com
- Which algorithm is used?
- Deploy / implementation details on technologies etc. $\cdots$
- Show screenshot

8. Summary

8.1 Summary and Conclusions

- Looked at text genre classification into 14 classes of documents of sizes 200, 800 and 3200 characters
- The classification accuracy grew with increasing length of the documents indicating the algorithms had difficulties to recognize genre from only a short snippet.
- Bigger gap in performance between documents with 200 and 800 characters than between 800 and 3200 characters.
- Feed-forward neural net on feature vectors outperforms logistic regression but not by much…
- Provided insights into what is typical for each genre (typical words)
- Deployed couple of classifiers to heroku with web interface

8.2 Future Work

Bibliography

[1] Erik Bernhardsson. Annoy. 2013.

[2] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[3] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[4] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[5] Martin F Porter. Snowball: A language for stemming algorithms, 2001.

[6] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[7] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[8] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *CoRR*, abs/1607.01759, 2016.

[9] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[11] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[13] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[14] Ye Zhang and Byron C. Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *CoRR*, abs/1510.03820, 2015.

[15] Michael Hart. Free ebooks by Project Gutenberg. `http://www.gutenberg.org/`.

[16] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[17] Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017.

[18] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[19] Jey Han Lau and Timothy Baldwin. An empirical evaluation of doc2vec with practical insights into document embedding generation. *CoRR*, abs/1607.05368, 2016.