The Art of Natural Language Processing: Classical, Modern and Contemporary Approaches to Text Document Classification

Andrea Ferrario*, Mara Nägelin*

Prepared for:
Fachgruppe "Data Science"
Swiss Association of Actuaries SAV

Version of March 1, 2020

Abstract

In this tutorial we introduce three approaches to preprocess text data with Natural Language Processing (NLP) and perform text document classification using machine learning. The first approach is based on 'bag-of-' models, the second one on word embeddings, while the third one introduces the two most popular Recurrent Neural Networks (RNNs), i.e. the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures. We apply all approaches on a case study where we classify movie reviews using Python and Tensorflow 2.0. The results of the case study show that extreme gradient boosting algorithms outperform adaptive boosting and random forests on bag-of-words and word embedding models, as well as LSTM and GRU RNNs, but at a steep computational cost. Finally, we provide the reader with comments on NLP applications for the insurance industry.

Keywords. natural language processing, bag-of-words models, word embeddings, machine learning, recurrent neural networks, deep learning, Python, Tensorflow 2.0, Keras.

0 Introduction and overview

This data analytics tutorial has been written for the working group "Data Science" of the Swiss Association of Actuaries SAV, see

`https://www.actuarialdatascience.org/`

The main purpose of this tutorial is to provide an overview of natural language processing methodologies to process text data and to classify text documents with machine learning. To do so, we introduce three distinct approaches. The first approach uses classical NLP pipelines to preprocess text documents, random forests and boosting algorithms to classify them using bag-of-words and bag-of-part-of-speech models. The second approach feeds text document embeddings to machine learning classifiers, using pre-trained word embeddings as input. The third

*Mobiliar Lab for Analytics at ETH and Department of Management, Technology and Economics, ETH Zurich, {aferrario, mnaegelin}@ethz.ch approach reduces text documents preprocessing to the minimum implementing recurrent neural network architectures (e.g. Long Short-Term Memory and Gated Recurrent Unit) for their classification. We apply all approaches to a case study based on data from the Internet Movie Database[1] (IMDb); we write all code in Python and we use recently introduced functionalities in Tensorflow 2.0 for deep learning routines.

0.1 Natural Language Processing

Natural Language Processing (NLP) is a discipline that combines linguistics, computer science and artificial intelligence to study interactions between computer systems and human natural language. Languages are structured systems of communications; they are said to be natural if they are the result of a natural evolution in humans' communities via use and repetition throughout time, as opposed to constructed languages. Examples of constructed languages are programming languages normally used by computer systems.
Among the main examples of NLP tasks one counts information retrieval (i.e. the development of algorithms to retrieve information from document collections), text classification, text generation, language translation, text sentiment analysis, production of taxonomies, entity relation modeling, automated coding of corpora, etc., [3, 27, 32]. NLP researchers aim at the development of quantitative approaches to solve those NLP tasks using stochastic modeling (e.g. machine learning), information theory and linear algebra methodologies [32].

The increased capacity of generating, storing and processing natural language samples in digital format and the recent advances in machine learning research [20, 25, 26] have strongly favoured the application of NLP techniques in a multitude of services and products. Examples are spelling correction in word processors, machine translation engines (e.g. Google Translate) search and speech engines (e.g. Amazon's Alexa), email spam classifiers and news feed engines.
In the insurance industry, the abundance of written evidence like policy contracts or claims notifications, as well as the possibility of recording customers' interactions with corporate conversational assistants ('chatbots'), provide data scientists and actuaries with an ever increasing collection of text documents to be analysed with NLP for the purpose of generating actionable and business-relevant insights.

In this tutorial, we collect selected material on NLP to introduce data pipelines aimed at the processing of text data and the use of machine learning models [37] to classify text documents in digital format. Our goal is to describe the following approaches:

1. **Classical Approach**: we generate bag-of-words and bag-of-POS[2] numerical representations of text documents to be fed into the machine learning classifiers;
2. **Modern Approach**: we use word embedding algorithms to compute real-valued numerical representations for each document to be fed into the machine learning classifiers;
3. **Contemporary Approach**: we leave text preprocessing to the minimum by training recurrent neural networks directly on text documents.

In order to properly describe all three approaches, we start by introducing NLP using Python in Section 1. We then move to the **Classical Approach** in Section 2 and we continue with the

[1] `https://www.imdb.com/`
[2] Part-Of-Speech, see Section 1.5.

Modern Approach in Section 3. In Section 4 we introduce the machine learning classifiers to be used for both approaches from Sections 2 and 3. The **Contemporary Approach** is detailed in Section 5, where different architectures of recurrent neural networks are discussed. We compare all approaches on a case study in Section 6 and we conclude this tutorial. In the Appendix we collect additional material on multilingual NLP.

0.2 On the use of NLP in insurance: opportunities and challenges

In this tutorial we discuss a case study focused on the automated classification of movie reviews, using NLP methodologies to preprocess textual data. However, as mentioned in Section 0.1, the insurance industry offers a wide array of use cases to apply NLP pipelines; examples are the classification of claims [29] with respect to claim type, severity of impacted line-of-business based on their textual descriptions, the classification of emails, policies, contracts, or the identification of fraud cases from textual data.

Although insurance companies are typically rich in unstructured data[3] (e.g. textual data), care has to be taken into account in defining business-relevant use cases. Among the main limitations to the effective use of NLP pipelines in use cases of relevance for the insurance industry we highlight:

1. the presence of multilanguage environments as, for example, in Switzerland (see the Appendix for additional material on NLP in German, French and Italian languages);
2. the intrinsic bias of pre-trained word embeddings (see Section 3), which may not intercept the linguistic specificities of the insurance domain (e.g. life or health insurance) of applicability for the case study at hand;
3. the necessity of designing and implementing complex data pipelines to store, retrieve and preprocess high volumes of textual data for NLP and machine learning on scalable infrastructures;
4. the hiring, training and up-skilling of diversified teams of professionals (e.g. data scientists and actuaries) committed to the identification and development of NLP-driven case studies.

The insurance companies able to conceive, deploy and scale NLP-driven use cases avoiding the aforementioned limitations are expected to gain considerable competitive advantage in the respective industries[4]. In summary, we invite the interested reader to consider the case study from Section 6 as a 'sand-box' to test different NLP and machine learning methodologies, before moving to use cases of relevance for the insurance industry. These can be tackled either for self-study, or to design NLP and machine learning-driven proof-of-concepts, through the active collaboration between business stakeholders, data scientists and actuaries.

1 NLP with Python: a short introduction

We introduce some material on NLP using Python that is relevant for the discussions in Sections 2 and 4, and the case study in Section 6. Our aim is to provide the reader with 1) an overview of text processing pipelines in Python, and 2) a discussion of those steps in the pipelines which traditionally represent a challenge to practitioners.

[3]`https://www.actuaries.org.uk/news-and-insights/news/natural-language-processing-insurance`

[4]Studies show that the market size for NLP is expected to grow; for more details, we refer to `https://www.marketsandmarkets.com/PressReleases/natural-language-processing-nlp.asp`

1.1 Text preprocessing pipelines with Python: an overview

Practitioners working with text data need to introduce a pipeline aimed at preprocessing of strings of text, for the NLP problem at hand. In this tutorial, with the term 'pipeline', we refer to a finite sequence of steps taking 'raw' text as input and returning properly preprocessed text as output. There is no silver bullet in the art of text preprocessing; the design of pipelines depends on the context, the language under consideration and the general aim of the NLP analysis at hand. This said, text preprocessing pipelines typically consist of the following steps:

1. import of raw text and formatting;
2. conversion of text to lowercase;
3. tokenization, i.e. split of all strings of text into tokens;
4. stopwords removal;
5. part-of-speech (POS) tagging of tokenized text;
6. stemming or lemmatization.

Depending on the specific application, one may also remove (too) frequent or rare words. In Python, these removals can be controlled by parameters in the `sklearn TfidfVectorizer()` function; we will return to this point in Section 2. The order of the steps performed in the above text preprocessing pipeline may vary as well, depending on the application at hand. However, after these steps, one typically moves to more advanced text processing activities like n-grams analysis, word embeddings and machine learning modeling. In the following subsections we discuss steps 2. to 6. of the above pipeline using the following Python resources:

1. `nltk`, i.e. the Natural Language Tool Kit[5]: an NLP platform for Python;
2. `SpaCy`[6]: a Python (and Cython) library for multi-language NLP with an eye towards end-to-end NLP applications;
3. `scikit-learn`[7] (often abbreviated as `sklearn`): a Python library for machine learning, which includes functions to compute bag-of-words models.

1.1.1 Test sentence: `text`

We apply steps 2. to 6. of the proposed pipeline to the test sentence `text` presented in Listing 1; `text` refers to a famous short story by Howard Phillips Lovecraft (1890-1937), an American writer from Providence, Rhode Island[8].

```
text = 'In H.P. Lovecraft's short story 'The Call of Cthulhu', the author states that in
    S. Latitude 47° 9', W. Longitude 126° 43' the great Cthulhu dreams in the sea-bottom
    city of R'lyeh.'
```

Code Listing 1: The sentence `text` to be preprocessed along the proposed NLP pipeline.

The sentence `text` comprises lexemes with special characters (e.g. the Celsius degree symbol `°`), hyphens (e.g. in `sea-bottom`), proper nouns with apostrophes (e.g. in `R'lyeh`), numbers and abbreviations (`H.P.`, `W.`, `S.`). It does not contain spelling errors; for the sake of simplicity, we suppose `text` is imported in a Python notebook/file; therefore, we start our analysis by

[5]`http://NLTK.org`

[6]`https://SpaCy.io/`

[7]`https://scikit-learn.org/stable/`

[8]The interested reader may consult [30] for the full text of the short story 'The Call of Cthulhu' and [22] for a work of literary criticism regarding Lovecraft's writings.

addressing step 2., or the conversion to lowercase text. The interested reader can reproduce all results shown in Section 1 using the Jupyter Notebook `NLP_Pipeline_Example.ipynb`, which is stored on the GitHub repository of the Fachgruppe "Data Science" of the Swiss Association of Actuaries (SAV) at

`https://github.com/JSchelldorfer/ActuarialDataScience`

1.2 Lowercase text

Converting all text strings in a document to lowercase is a common procedure in NLP text preprocessing. Python allows us to convert strings to lowercase by using the `.lower()` method. When used before tokenization (see Section 1.3), lowercasing allows us to 'harmonize' strings of text by generating a single token out of lowercase and uppercase variants of the same word. However, by lowercasing one cannot recognize proper nouns from other parts-of-speech like common nouns or adjectives (e.g. 'Bob White' vs. 'white cloud'). Heuristics can be used to limit such challenges; we refer to [32] for a discussion on this point. We lowercase `text` in Listing 2.

```
in h.p. lovecraft's short story 'the call of cthulhu', the author states that in
s. latitude 47° 9', w. longitude 126° 43' the great cthulhu dreams in
the sea-bottom city of r'lyeh.
```

Code Listing 2: Lowercasing `text`.

1.3 Tokenization

A token "is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing" [33]. Therefore, given a string of characters, its tokenization is "the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation" [33]. In Python we can apply different strategies for tokenization; in the following subsections we introduce the most used off-the-shelf tokenizers. It is also important to specify that tokens are more general objects than graphic words, which are "string[s] of continuous alphanumeric characters with space on either side; [they] may include hyphens and apostrophes, but no other punctuation marks" [24]. In this tutorial, graphical words are used a 'proxy' of the more general concept of 'word'. Unfortunately, the original definition of graphical word by Kŭcera and Francis [24] does not include tokens like '$15.10', '.NET' or URLs for web resources in the set of graphic words, although these tokens may be considered as such, depending on the context. Therefore, tokenization procedures have to take care of punctuation (e.g. commas, periods, apostrophes and hyphens), language specific word segmentation rules and stylistic or typographical differences (e.g. those arising when writing telephone numbers). To do so, most tokenizers use regular expressions[9] to define the set of text strings to be retrieved [27]. We now discuss four distinct tokenizers which are available in Python, starting with the easiest one, i.e. the whitespace tokenizer.

[9]Regular expressions in Python are performed using the library regex: `https://docs.python.org/3.7/library/re.html`

1.3.1 Whitespace tokenizer

The whitespace tokenizer splits sequences of characters based on whitespaces [4]; it can be easily implemented using the `.split()` method on strings (we note that the default separator in `split()` is any whitespace). The whitespace tokenizer generates tokens with punctuation (e.g. any word at the end of a sentence is grafted with a punctuation sign) and it splits expressions which should be considered together, like compound geographic locations (e.g. 'Grand Canyon', 'Los Angeles'), language-specific expressions (e.g. *de facto*, *et alia*) and collocations[10]. It splits single phrases like 'in spite of', 'in order to' in multiple graphical words [32]. On the other hand, it preserves hyphens (e.g. in 'step-wise') and apostrophes (e.g. in 'hasn't'). Punctuation characters in Python can be accessed by the command `string.punctuation`; they are shown in Listing 3.

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{
```

Code Listing 3: Punctuation characters in Python.

Removal of punctuation after whitespace tokenization can be implemented using the functions `maketrans()` and `translate()`, although care has to be exercised in punctuation removal [32], due to the possibility of modifying meaningful tokens, for example by stripping punctuation off 'C#', 'etc.' or '10.50$', or limiting the available information on the macro structure of a text document due to the removal of commas and periods between sentences. We apply the whitespace tokenizer to the test sentence `text`; results are shown in Listing 4.

1.3.2 The Natural Language Tool Kit tokenizer: `word_tokenize()`

The Natural Language Tool Kit provides users with a tokenizer called `word_tokenizer()` by importing the library `NLTK`. The tokenizer uses the original Treebank[11] tokenizer, which tokenizes text using regular expressions. The Python code used in `NLTK` to implement the Treebank tokenizer can be accessed at

`http://www.nltk.org/api/nltk.tokenize.html?highlight=tokenizer`,

In summary, the function `word_tokenizer()` uses regular expressions to 1) split standard English contractions, 2) treat most punctuation as separate tokens, 3) split commas and single quotes when followed by whitespaces, and 4) separate periods that appear at the end of a line. We apply the `word_tokenize()` tokenizer to the test sentence `text`; results are shown in Listing 4.

1.3.3 `SpaCy` tokenizer

Our last example of Python tokenizer is from the `SpaCy` library. Tokenization in `SpaCy` is performed using the `Tokenizer` class, which implements an *ad-hoc* algorithm to tokenize strings of text sequentially.

[10]A collocation is "an expression consisting of two or more words that correspond to some conventional way of saying things"[32]. Examples include noun phrases like 'strong tea', 'the rich and the powerful' or phrasal verbs (see [32] for more details).

[11]The Penn Treebank is a corpus of over 4.5 Million words in American English, which has been annotated during the first three-year phase of the Penn Treebank Project (1989-1992) [34].

The algorithm is explained in detail at

`https://SpaCy.io/usage/linguistic-features#how-tokenizer-works`,

where a Python implementation of the algorithm is presented, as well. `Tokenizer` initially splits the string of characters using whitespaces; the result is a finite set of substrings of characters. The algorithm then searches for 1) special rules to be applied to substrings, and 2) prefixes as well as suffixes. It also checks for 'infixes' (e.g. punctuation characters like `,.\$!-`), in case no prefix or suffix is found. `Tokenizer` stops to loop around substrings when no special rule, as well as prefix or suffix to be removed, can be found. We apply the `SpaCy` tokenizer to the test sentence `text`; results are shown in Listing 4.

1.3.4 Tokenization for more advanced NLP: using `sklearn` `TfidfVectorizer()`

The Python library `sklearn` allows one for flexible NLP pipelines on text data; in particular, the function `TfidfVectorizer()`[12] is used to transform a collection of raw documents into a document-term matrix with real-valued elements: this procedure is commonly called vectorization (see Section 6 for more details). `TfidfVectorizer()` starts processing raw documents by tokenizing them, before computing the document-term matrix elements according to 'bag-of-' representations [13], which will be discussed in Section 2. The `TfidfVectorizer()` default tokenizer implements a regular expression which selects tokens of 2 or more alphanumeric characters (i.e. of the form `[A-Za-z0-9_]`, note the presence of the underscore[14]). Punctuation is ignored and it is always treated as a token separator, instead. Therefore, the default tokenizer in `TfidfVectorizer()` does not retrieve tokens with non-alphanumeric characters or punctuation. We apply the `sklearn` `TfidfVectorizer()` function to the test sentence `text`; results are shown in Listing 4.

```
# whitespace tokenizer
‘In’, ‘H.P.’, "Lovecraft’s", ‘short’, ‘story’, "‘The", ‘Call’, ‘of’, "Cthulhu’,", ‘the’,
‘author’, ‘states’, ‘that’, ‘in’, ‘S.’, ‘Latitude’, ‘47°’, "9’,", ‘W.’, ‘Longitude’,
‘126°’, "43’", ‘the’, ‘great’, ‘Cthulhu’, ‘dreams’, ‘in’, ‘the’, ‘sea-bottom’, ‘city’,
‘of’, "R’lyeh."

# NLTK
‘In’, ‘H.P’, ‘.’, ‘Lovecraft’, "’s", ‘short’, ‘story’, "‘The", ‘Call’, ‘of’, ‘Cthulhu’,
"’", ‘,’, ‘the’, ‘author’, ‘states’, ‘that’, ‘in’, ‘S.’, ‘Latitude’, ‘47°’, ‘9’, "’",
‘,’, ‘W.’, ‘Longitude’, ‘126°’, ‘43’, "’", ‘the’, ‘great’, ‘Cthulhu’, ‘dreams’, ‘in’,
‘the’, ‘sea-bottom’, ‘city’, ‘of’, "R’lyeh", ‘.’

# SpaCy
‘In’, ‘H.P.’, ‘Lovecraft’, "’s", ‘short’, ‘story’, "’", ‘The’, ‘Call’, ‘of’, ‘Cthulhu’,
"’", ‘,’, ‘the’, ‘author’, ‘states’, ‘that’, ‘in’, ‘S.’, ‘Latitude’,  ‘47’, ‘°’, ‘9’,
"’", ‘,’, ‘W.’, ‘Longitude’, ‘126’, ‘°’, ‘43’, "’", ‘the’, ‘great’, ‘Cthulhu’,
‘dreams’, ‘in’, ‘the’, ‘sea’, ‘-’, ‘bottom’, ‘city’, ‘of’, ‘R’lyeh’, ‘.’

# sklearn: TfidfVectorizer()
‘126’, ‘43’, ‘47’, ‘author’, ‘bottom’, ‘call’, ‘city’, ‘cthulhu’, ‘dreams’, ‘great’,
```

[12]`https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html#sklearn.feature_extraction.text.TfidfVectorizer.fit_transform`

[13]`https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction`

[14]On the other hand, the Python method `.isalpha()` considers a string as alphanumeric if it is of the form `[A-Za-z0-9]`.

```
'in', 'latitude', 'longitude', 'lovecraft', 'lyeh', 'of', 'sea', 'short', 'states',
'story', 'that', 'the'
```

Code Listing 4: Applying different tokenizers on `text`. Note that Python uses quotation marks as delimiter of tokens in which apostrophes appear.

The whitespace tokenizer keeps punctuation in tokens, it retrieves all strings of characters including abbreviations, special characters (Celsius degree symbol `°`) and words with hyphens (`sea-bottom`) or apostrophes (`R'Lyeh`). It can be followed by punctuation removal via `.maketrans()` or alphanumeric character retrieval using `.isalpha()`.
Using the `NLTK` tokenizer, punctuation is stripped off words and retrieved as separate tokens. Note that the abbreviation `H.P.` results in `H.P` and `.`, separately. Exceptions to this rule are the abbreviations `S.` and `W.`, where the period is correctly kept; the token `'s` is generated from `Lovecraft's`. The tokenizer correctly retrieves abbreviations, special characters, words with hyphens and apostrophes. Again, punctuation can be removed using `.maketrans()`.
The `SpaCy` tokenizer strips punctuation off words and retrieves it as separate tokens, with exception of periods. In fact, it returns the tokens `H.P.`, `S.` and `W.`. The word `sea-bottom` is split in the three tokens `sea`, `-` and `bottom`. The word `R'lyeh` is correctly retrieved: no stripping due to the apostrophe is performed.
Finally, the tokenizer in `TfidfVectorizer()` starts by performing lowercasing of all characters in `text` and it removes all punctuation; therefore, abbreviations do not appear, and the word `R'lyeh` returns the single token `lyeh`. Special characters are also stripped off. As mentioned above, the minimum (characters) length of tokens is two: therefore, no token is retrieved from the original substring `9'`. In summary, we have shown that different tokenizers generate different collections of tokens; this finding becomes particularly relevant in the case of machine learning modeling performed on the outputs of NLP pipelines, as the results of modeling depend, among others, on the choice of the tokenizer.

1.4 On stopwords in NLP pipelines

The removal of stopwords is typically offered as a step in text preprocessing pipelines. Stopwords are those words in a given language that are not considered to be useful for the analysis at hand, e.g. an information retrieval exercise [32]. Moreover, the removal of words with little semantic weight allows one to reduce dimensionality in 'bag-of-' models: see Section 2. Stopwords are typically the most common used words in a given language (e.g. 'the', 'and', 'from' in English). Unfortunately, the concept of 'importance' or 'usefulness' is context specific; discussions on the use of stopword lists are present in the specialized literature [39]. In what follows, we show the different stopword lists practitioners can access using the Python libraries `NLTK`, `SpaCy` and `sklearn`. We want to provide the readers with an overview of the most common stopword lists in Python and highlight some of the challenges arising from their use.

1.4.1 Stopwords in `NLTK`

The `NLTK` stopword list[15] consists of 179 words. It contains single character words like 'i', 's' and 't', as well as tokens like 'haven', 'haven't', as both could emerge after tokenization

[15] The list that can be accessed at `https://gist.github.com/sebleier/554280` is obsolete; we refer to the Jupyter Notebook `NLP_Pipeline_Example.ipynb` to access the most actual list.

(the NLTK tokenizer word_tokenizer() treats punctuation as separate tokens, as discussed in Section 1.3.2).

1.4.2 Stopwords in SpaCy

SpaCy contains an extensive list of 305 stopwords, as it can be checked by running the following Python code:

```
import spacy
spacy_stop = spacy.lang.en.stop_words.STOP_WORDS
print(list(spacy_stop))
```

The list contains single character words like ‘d’, ‘m’ or ‘a’, as well as adjectives like ‘bottom’, ‘former’ and nouns like ‘five’, ‘fifteen’ and ‘sixty’.

1.4.3 Stopwords in sklearn

The sklearn stopword list (called ENGLISH_STOP_WORDS) is taken from the Glasgow Information Retrieval Group (GIRG)[16] with some modifications (the original list is 319 entries long, while ENGLISH_STOP_WORDS contains 318 entries). It can be accessed with the command

```
from sklearn.feature_extraction.stop_words import ENGLISH_STOP_WORDS
```

The word ‘computer’ from the GIRG list has been removed from ENGLISH_STOP_WORDS, and the word ‘fify’ is corrected to ‘fifty’. The list can be used when calling TfidfVectorizer() by specifying the parameter stop_words:‘english’. However, the use of the sklearn stopword list presents some challenges; in fact the list

- contains words which are shorter than 2 characters; however, these are automatically discarded by the TfidfVectorizer() tokenizer[17];
- does not include enclitics[18] which are normally generated by the TfidfVectorizer() tokenizer;
- does contain ‘hasnt’ but not ‘hadnt’;
- shows omissions among modal verbs (e.g. ‘does’ does not appear), intensifiers (e.g. ‘really’), verbs (e.g. ‘get’ is in the list, but ‘make’ is excluded).

In Listing 5 we collect the results of stopword removal from the different tokenizations of the test sentence text.

```
# NLTK tokenizer + NLTK stopwords
‘the’, ‘of’, ‘that’, ‘in’

# SpaCy tokenizer + SpaCy stopwords
‘The’, ‘of’, ‘that’, ‘Call’, ‘in’, "’s", ‘In’, ‘the’, ‘bottom’

# sklearn: TfidfVectorizer()(stop_word=‘english’)
```

[16]http://ir.dcs.gla.ac.uk/resources/linguistic_utils/stop_words

[17]For more information, we refer to https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

[18]https://www.merriam-webster.com/dictionary/enclitic

```
'bottom', 'call', 'in', 'of', 'that', 'the'
```

Code Listing 5: List of removed stopwords from `text` after tokenization.

Results in Listing 5 show that `SpaCy` performs the most extensive stopwords removal, followed by `TfidfVectorizer()`. The `NLTK` library removes only prepositions (`of` and `in`), a relative pronoun (`that`) and an article (`the`). Both the `SpaCy` tokenizer and `sklearn TfidfVectorizer()` remove the nouns `Call` (or `call` in the case of `TfidfVectorizer()`, as it performs lowercasing by default) and `bottom`. Both words are semantically important, as the first originates from tokenization of the title of Lovecraft's short story 'The Call of Cthulhu', while the second stems from the 'incorrect' tokenization of the hyphenated word `sea-bottom`, as remarked in the comments to Listing 4.

1.5 Part-Of-Speech tagging

A part-of-speech (POS) [32] is a set of words which show similar syntactic behaviour, like nouns, pronouns, adjectives, verbs and propositions. Part-of-speech (POS) tagging is the process of assigning a part-of-speech tag to each word in a given corpus. The algorithm that performs the tagging is called a *POS tagger*; the set of tags that a POS tagger uses is a *tagset*. POS taggers are trained on corpora of documents; in applications, the most used corpora are the Brown[19] and the Penn Treebank ones. Algorithms for POS tagging can be rule-based or stochastic. Natural languages are rich in syntactical and semantical ambiguities; this affects the performance of POS taggers which, on the other hand, can be seen as methods which use existing corpora to resolve those ambiguities. The main applications of POS tagging comprise Named Entity Recognition (NER), sentiment analysis, question answering, and word sense disambiguation. POS tags can be used in 'bag-of' models for machine learning modeling; we will discuss this application in Section 2.
In what follows, we will perform POS tagging using the function `pos_tag()` in the `NLTK` Python library. `pos_tag()` offers 3 distinct POS taggers; the default one is the Penn Treebank[20], which comprises 36 distinct tags (45 including punctuation [27]). Users can specify also the Brown POS tagger or a more recent 'universal' tagger [44], which contains only 12 tags, i.e. `NOUN` (nouns), `VERB` (verbs), `ADJ` (adjectives), `ADV` (adverbs), `PRON` (pronouns), `DET` (determiners and articles), `ADP` (prepositions and postpositions), `NUM` (numerals), `CONJ` (conjunctions), `PRT` (particles), `'.'` (punctuation marks) and `X` (a catch-all for other categories such as abbreviations or foreign words). We apply the default `pos_tag()` to `text` (after its tokenization with `word_tokenize()`) in Listing 6.

```
# NLTK tokenizer + NLTK POS tagger (default)
('In', 'IN'), ('H.P', 'NNP'), ('.', '.'), ('Lovecraft', 'NNP'), ("'s", 'POS'),
('short', 'JJ'), ('story', 'NN'), ("'The", 'POS'), ('Call', 'NNP'), ('of', 'IN'),
('Cthulhu', 'NNP'), ("'", 'POS'), (',', ','), ('the', 'DT'), ('author', 'NN'),
('states', 'VBZ'), ('that', 'IN'), ('in', 'IN'), ('S.', 'NNP'),
('Latitude', 'NNP'), ('47°', 'CD'), ('9', 'CD'), ("'", "''"), (',', ','),
```

[19]The Brown corpus [24] by Kŭcera and Francis contains text in English from 500 sources, which are categorized in 15 distinct genres. It is the first million-word electronic corpus of English text. For an overview of all categories and genres in the corpus, we refer to the documentation in `http://clu.uni.no/icame/manuals/BROWN/BCMLOS.HTM`.

[20]`https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`

```
('W.', 'NNP'), ('Longitude', 'NNP'), ('126°', 'CD'), ('43', 'CD'),
("'", "''"), ('the', 'DT'), ('great', 'JJ'), ('Cthulhu', 'NNP'),
('dreams', 'NN'), ('in', 'IN'), ('the', 'DT'), ('sea-bottom', 'JJ'), ('city', 'NN'),
('of', 'IN'), ("R'lyeh", 'NNP'), ('.', '.')
```

Code Listing 6: POS tagging of `text` after tokenization in `NLTK`.

As shown in Listing 6, the tagger correctly tags most of the tokens originating from the tokenization of `text`. For example, we note that the word `'states'` is correctly tagged as a verb, third singular person `VBZ` and not as a noun (i.e. the plural of the noun 'state'). However, few tokens are not correctly tagged; for example, the token `'The` originating from the substring `'The Call of Cthulhu'` in `text` is tagged as `POS`, or possessive ending. Both `Latitude` and `Longitude` are correctly tagged as proper nouns, singular `NNP`; however, the verb `'dreams'` is tagged as noun `NN`, instead of `VBZ` as for `'states'`.

1.6 Stemming and lemmatization

1.6.1 Stemming

Stemming and lemmatization are two procedures to reduce inflectional forms in text, e.g. the variability that words present to express different grammatical categories. These categories are expressed by inflections through affixes, which are sequences of letters comprising prefixes, suffixes or infixes. Suffixes decorate verbs in English or nouns in all languages where nouns are declined (e.g. in Latin). Stemming algorithms (also called 'stemmers') strip affixes off words to reduce them to their root forms, which need not to be words in the given language. For the English language, the most used stemming algorithms are the Porter, the 'English Stemmer' or 'Porter2', and Lancaster (Paice-Husk).
The Porter stemmer[21] [45] is a simple and efficient stemming algorithm, which is commonly used in information retrieval. It strips suffixes off words using on a simple set of rules that have been originally implemented in BCPL ('Basic Combined Programming Language'); it is now available in many programming languages. The rules depend on the suffix; they are of the form

$$\texttt{(CONDITION)\ PREFIX} \rightarrow \texttt{RESULT} \tag{1.1}$$

In (1.1), `RESULT` is either the null string (the prefix `PREFIX` is completely stripped) or a sub-string of `RESULT`. Notably, the stemming rules in the Porter algorithm avoid to remove suffixes when the resulting stem is considered to be 'too short'; the length of a stem is encoded in an integer measure m (we refer to [45] for all details). For example, the word `PRELATE` is not stemmed to `PREL` by removing the suffix `-ATE`; on the other hand, `ARCHPRELATE` becomes `ARCHPREL`. The Porter stemmer is available for Python in the `NLTK` library.
The 'English Stemmer' or 'Porter2' stemming algorithm[22] is a derivation of the original Porter stemmer. Porter states that it is conceived as an improvement of the original stripping algorithm[23]. Changes with respect to the original Porter stemmer are not extensive and comprise

[21]The original stemming algorithm paper is available as .txt file at `https://tartarus.org/martin/PorterStemmer/def.txt`

[22]It is often erroneously referred to as 'Snowball' stemmer. In fact, 'Snowball' is the language developed by Porter to write the 'English Stemmer'.

[23]The interested reader can check the English Stemmer documentation at `http://snowball.tartarus.org/algorithms/english/stemmer.html`

few additional rules (e.g. for the suffix `-LY`), among others. The Porter2 stemmer is available in Python in the `stemming 1.0` project[24].
Lastly, the Lancaster (Paice-Husk) stemmer [40] is an iterative algorithm with 115 rules to strip suffixes[25]. It strips prefixes like ‘kilo’, ‘micro’, ‘pseudo’, among others. The Lancaster stemmer is considered to be more ‘aggressive’ than Porter and Porter2, although it is usually quicker than the others due to simpler stripping rules. The Lancaster (Paice-Husk) stemmer is available in Python in the `stemming 1.0` project. In summary, stemming is a procedure that can be leveraged in NLP text processing pipelines. However, the evaluation of the worth of conflating words (i.e. stripping suffixes) is notoriously difficult and it depends on the semantic of the words to be stemmed. As discussed in [45], “the best criterion for removing suffixes from two words `W1` and `W2` to produce a single stem `S`, is to say that we do so if there appears to be no difference between the two statements ‘a document is about `W1`’ and ‘a document is about `W2`’”. Therefore, in [45], Porter continues by stating that “if `W1=‘CONNECTION’` and `W2=‘CONNECTIONS’` it seems very reasonable to conflate them to a single stem. But if `W1=‘RELATE’` and `W2=‘RELATIVITY’` it seems perhaps unreasonable, especially if the document collection is concerned with theoretical physics.”.

1.6.2 Lemmatization

Lemmatization [33] uses vocabularies and morphological analysis of words to remove inflectional endings and return the dictionary form (also called ‘canonical form’) of a word; this latter is referred to as lemma. Lemmata are valid words in the given language (as opposed to stems returned by stemming algorithms). In Python, the `NLTK` library provides the function `WordNetLemmatizer()` that uses the WordNet database[26] to lookup lemmata of words and perform lemmatization on text. Alternatively, practitioners can access the Python `TextBlog` library[27]. The necessity to access a corpus makes lemmatization algorithms usually slower than stemming ones. Moreover, it is necessary to specify which part-of-speech to lemmatize.

We close this short section on stemming and lemmatization by applying the Porter stemming algorithm to `text`, after tokenization with the `NLTK` tokenizer. Results are shown in Listing 7.

```
# NLTK tokenizer + Porter stemming
‘In’, ‘h.p’, ‘.’, ‘lovecraft’, "’s", ‘short’, ‘stori’, "’the", ‘call’, ‘of’, ‘cthulhu’,
"’", ‘,’, ‘the’, ‘author’, ‘state’, ‘that’, ‘in’, ‘S.’, ‘latitud’, ‘47°’,
‘9’, "’", ‘,’, ‘W.’, ‘longitud’, ‘126°’, ‘43’, "’", ‘the’, ‘great’, ‘cthulhu’,
‘dream’, ‘in’, ‘the’, ‘sea-bottom’, ‘citi’, ‘of’, "r’lyeh", ‘.’
```

Code Listing 7: Porter stemming after tokenization of `text` using the `NLTK` tokenizer.

We note that the tokens affected by stemming are the nouns `story`, `city` (see Step 1c in [45]), the nouns `Latitude`, `Longitude` (see Step 5a in [45]) and the verbs `states`, `dreams` (see Step 1a in [45]).

[24]`https://pypi.org/project/stemming/1.0/`

[25]Its Python implementation in `NLTK` can be accessed at `http://www.nltk.org/_modules/nltk/stem/lancaster.html#LancasterStemmer`

[26]The WordNet database is a widely used lexical database of English language. For more details, we refer to the official documentation at `https://wordnet.princeton.edu/`

[27]`https://textblob.readthedocs.io/en/dev/`

2 Classical Approach: document classification with 'bag-of-' models and machine learning

We describe now the **Classical Approach** to the automated classification of text documents with machine learning. We start with some notation to describe the classification problem at hand. Let $\mathcal{D}$ be a dataset $\mathcal{D} = \{(Y_1, x_1), \ldots, (Y_n, x_n)\}$, with $Y_i \in \mathcal{Y}$, and $x_i \in \mathcal{X}$ for all $i = 1, \ldots, n$. $\mathcal{X}$ and $\mathcal{Y}$ are the domain set and the label (finite) set, respectively, of the statistical learning framework at hand, while $\mathcal{D}$ denotes a finite set of data sampled from $\mathcal{Y} \times \mathcal{X}$. In this tutorial, each x_i is a text document, i.e. a finite string of characters with a meaning in a given language. The labels y_i in the set $\mathcal{Y}$ could denote the sentiment extracted from the corresponding document x_i, the name of the author of the text or any other type of discrete label. We introduce two examples of 'bag-of-' modeling: the bag-of-words and bag-of-POS models.

In the case of bag-of-words models, our goal is to introduce a map $\varphi : \mathcal{X} \to \mathbb{R}^N$, which defines a real-valued representation of dimension $N > 0$ for each text document $x_i \in \mathcal{X}$ by the tokenization of the document, and the counting (possibly with a normalization, see below) of the occurrences in x_i of all the tokens retrieved from $\mathcal{X}$[28]. Similarly, in the case of bag-of-POS models, we introduce a map $\varphi : \mathcal{X} \to \mathbb{R}^N$ by tokenizing each document x_i, then POS-tagging the tokens and counting the occurrences of POS-tags (possibly with normalization). As the number of POS tags in a set of documents is generally smaller than the number of words extracted from them, bag-of-POS models can be used to reduce data dimensionality before applying machine learning modeling.
Given the map φ, we train machine learning classifiers $h \in \mathcal{H}$ from a given class of hypotheses[29] on the real-valued representations $\varphi(x_i)$, mapping each document x_i to a label $h(\varphi(x_i)) \in \mathcal{Y}$[30]. In Python, the map φ is constructed using the function `TfidfVectorizer()`, which applies tf-idf normalization to the counts of retrieved tokens by default[31]; we discuss its functioning in Section 2.2. Machine learning classifiers from different hypothesis classes $\mathcal{H}$ and their Python implementations are shown in Section 4 below.

2.1 Machine learning with 'bag-of-' models

The pipeline to train and select the 'best' machine learning model to classify text documents in $\mathcal{D}$ using both types of 'bag-of-' models comprises the following steps:

1. import of the dataset $\mathcal{D}$ (e.g. in Python);
2. partition $\mathcal{D} = \mathcal{D}_{train} \sqcup \mathcal{D}_{test}$ of $\mathcal{D}$ into train and test datasets;
3. run of hyperparameter tuning using k-fold cross-validation on $\mathcal{D}_{train}$ by performing:
   (a) NLP preprocessing ('bag-of-' models) of text documents $x_i \in \mathcal{X}$ using the function `TfidfVectorize()`;

[28]We note that the map φ is not injective, in general. Take, for example, the corpus $\mathcal{X}$ comprising the two documents x_1=`{I am John.}`, and x_2=`{Am I John?}`. Let us consider a tokenizer that performs lowercasing and strips punctuation; the map $\varphi : \mathcal{X} \to \mathbb{R}^3$, $\varphi(x_i) = (x_i^1, x_i^2, x_i^3)$, with x_i^j denoting the counts of the j-th token in the i-th document, is such that $\varphi(x_1) = \varphi(x_2) = (1, 1, 1)$.
[29]The terminology in this section follows the one from the monograph [51].
[30]We are sloppy with notation here; a more rigorous exposition would need the introduction of multi-sets.
[31]For all details on tf-idf normalization, we refer to `https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction`

   (b) machine learning modeling, by selecting different classes $\mathcal{H}$ of classifiers.

4. retrieval of the 'best' combination of NLP preprocessing and machine learning model per each class $\mathcal{H}$, by computing performance measures on validation sets.

We apply this pipeline to a case study in Section 6. We end our exposition of the **Classical Approach** with some additional information on `TfidfVectorize()`.

2.2 'Bag-of-' models with `TfidfVectorizer()`

In the context of the **Classical Approach**, NLP preprocessing of text document is performed using the `sklearn` `TfidfVectorizer()` function; it performs preprocessing steps on text like lowercasing, tokenization and stopword removal. We show it in Listing 8, discussing some parameters that will be of relevance for the case study in Section 6.

```
TfidfVectorizer(input='content', encoding='utf-8', decode_error='strict',
strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None, analyzer='word',
stop_words=None, token_pattern='(?u)\b\w\w+\b', ngram_range=(1, 1), max_df=1.0,
min_df=1, max_features=None, vocabulary=None, binary=False, dtype=<class'numpy.float64'>,
norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)
```

Code Listing 8: `sklearn` function `TfidfVectorizer()`.

Let us suppose to have a corpus of n text documents (for simplicity, we suppose that the encoding `encoding='uft-8'` is correct); we call the corpus via the parameter `input`. The function `TfidfVectorizer()` (by its Python module `fit_transform`) allows us to generate a 'document-term matrix' from the given corpus with term frequency-inverse document frequency (`tf-idf`) normalization, following a 'bag-of-' model to NLP text processing. If the corpus called via `input` consists of raw text documents, then `TfidfVectorizer()` performs bag-of-words modeling. On the other hand, if strings of POS tags (generated from text documents using a given tagger) are called via `input`, then `TfidfVectorizer()` performs a bag-of-POS modeling routine, instead.
Through the parameter `tokenizer` one controls the tokenization of the text documents called via `input`. The default `tokenizer=None` applies tokenization discussed in Section 1.3, which is specified by `token_pattern ='(?u)\b\w\w+\b'`. We note that with the choice `lowercase=True`, the function applies lowercasing before tokenization. Similarly, `stop_words` controls the use of stopword lists, as discussed in Section 1.4. The parameters `ngram_range`, `max_df`, `min_df` and `max_features` are especially relevant in the context of text document classification with machine learning; in fact, `ngram_range` allows one to select the range of ngrams[32] to be produced via tokenization, while `max_df` and `min_df` set the maximum and the minimum number of documents in which a given token (or term) has to appear to be retrieved by `TfidfVectorizer()`; the parameters allow one to control for 'too frequent' and 'too few frequent' tokens in the given corpus of documents. For example, `max_df=0.3` results in ignoring tokens that appear in more than 30% of documents. On the other hand, `max_features` allows one to trim the number of tokens returned by `TfidfVectorizer()` specifying the number of those to be retrieved, based on their frequencies across the whole corpus of documents. For example, `max_features=1000`

[32] An n-gram is a sequence of consecutive tokens of (finite) length n; a single token is an 1-gram, two consecutive tokens are a 2-gram and so on.

results in retrieving only the top 1000 tokens, after sorting them in descending order with respect to their frequency across the corpus. For the description of the other parameters and for additional details on `TfidfVectorizer()` we refer to the official `sklearn` documentation.
The method `.fit_transform()` applied to the result of `TfidfVectorizer()` returns the term-document matrix; the list of terms can be accessed via the module `.get_feature_names()`. We refer to the Jupyter notebook '`NLP_IMDb_Case_Study_ML.ipynb`' for additional details, which can be accessed at

`https://github.com/JSchelldorfer/ActuarialDataScience`

3 Modern Approach: word embeddings

At the core of the **Modern Approach** lie word embeddings, which are maps

$$\mathsf{e} : V \to \mathbb{R}^d, \quad \omega_i \mapsto \mathsf{e}(\omega_i), \tag{3.1}$$

where V denotes a finite set of n words (e.g. a vocabulary) such that $0 < d \leq n$. The idea of embeddings is to associate a real-valued vector to each word in V. Following the distributional hypothesis about languages and words [19], which has been popularized by Firth [14], a word can be recognized by 'the company it keeps'; therefore, words that occur in similar context should be closer to each other in the vector space $\mathbb{R}^d$, under the embedding e.
In other words, endowing $\mathbb{R}^d$ with a metric $\mathrm{d} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_+$, words which are semantically similar should have embeddings that are 'near' in $\mathbb{R}^d$, with respect to $\mathrm{d}(\cdot,\cdot)$.
The most straightforward example of word embedding is the one-hot embedding $\mathsf{e}_{\text{one}} : V \to \mathbb{R}^n$, where

$$\mathsf{e}_{\text{one}}(\omega_i) = \mathsf{e}^i_{\text{one}} := (0, \ldots, 0, \underbrace{1}_{\text{i-th position}}, 0, \ldots, 0) \in \mathbb{R}^n.$$

It has already been used in the bag-of-words model discussed in Section 2.2. The one-hot embedding does not perform dimensionality reduction: the dimension of the embedded vectors is equal to the number of distinct words in the given vocabulary (i.e. $d = n$); this may generate computational issues in presence of extensive vocabularies from rich corpora. Moreover, considering the Euclidean metric space $(\mathbb{R}^n, \mathrm{d})$, with $\mathrm{d}(\mathsf{e}^i_{\text{one}}, \mathsf{e}^j_{\text{one}}) = \sqrt{\langle \mathsf{e}^i_{\text{one}} - \mathsf{e}^j_{\text{one}}, \mathsf{e}^i_{\text{one}} - \mathsf{e}^j_{\text{one}} \rangle}$, the one-hot embedding e_{one} generates orthonormal and equidistant vectors:

$$\langle \mathsf{e}^i_{\text{one}}, \mathsf{e}^j_{\text{one}} \rangle = \delta_{ij},$$
$$\mathrm{d}(\mathsf{e}^i_{\text{one}}, \mathsf{e}^j_{\text{one}}) = \sqrt{2}(1 - \delta_{ij}).$$

This poses limitations to the semantics encoded in the embedding; the word 'cat' is as dissimilar to the word 'dog' than it is from any other word. Moreover, bag-of-words models using one-hot word encodings generate high-dimensional and sparse matrices.
Therefore, researchers have tried to construct embeddings such that $d \ll n$, called *dense* embeddings. In most applications, the dimension d is of the order of few hundreds (while n is typically in the order of millions or even billions); therefore, the use of dense embeddings allows one to reduce substantially computational complexity, hopefully leading to reasonable proximity in terms of meaning of words.

3.1 Brief history of word embeddings

In this section we introduce the three most relevant word embedding models from the literature; we start with the first modern example, i.e. the neural probabilistic language model, and we continue with `word2vec` and `GloVe`. In addition to these models, the interested reader can also access Fasttext [5] and LexVec[33] [50].

3.1.1 Neural probabilistic language modeling

The term 'word embeddings' has been introduced for the first time in [1], where an embedding of a word in a given vocabulary is defined as a distributed representation, which is learned using a shallow neural network. Similarities between words have been studied in the context of clustering [6, 43], where each word is associated with a discrete class; the idea behind clustering is that words in the same cluster show some sort of 'similarity'. However, it is in [1] that the first approach to continuous real-vector representation of words is presented. We describe the model in more detail; to do so we closely follow the exposition in [1].

A statistical language model is a probability distribution $f(w_1, \ldots, w_T)$ over sequences $w_1^T := \{w_1, \ldots, w_T\}$ of words in a given vocabulary, of length $T > 0$. This distribution is estimated by probabilities $\hat{P}(w_1^T)$ represented as conditional probabilities of the next words, given all the previous ones[34], that is,

$$\hat{P}(w_1^T) := \prod_{t=1}^{T} \hat{P}(w_t | w_1^{t-1}). \tag{3.2}$$

The main advantage of this representation is the use of word order to estimate probabilities. Moreover, words closer to the one whose conditional probability has to be estimated are expected to be more dependent. N-grams models follow (3.2) by introducing sub-sequences $w_{t-n+1}^{t-1} := \{w_{t-n+1}, \ldots, w_{t-1}\}$ of length $N = n - 1 > 0$, with

$$\hat{P}(w_t | w_1^{t-1}) \approx \hat{P}(w_t | w_{t-n+1}^{t-1}).$$

Thus, n-grams approximate (3.2) by a Markov property of length N. However, these traditional methods suffer from the curse of dimensionality; a word sequence used as test input is likely to be different from all the word sequences used to train the model, leading to zero joint probability (in [1], this is referred to as the 'generalization problem'). Moreover, traditional statistical language models do not encode the concept of 'word similarity', which should result in sentences made of words with similar grammatical and syntactical roles being as likely.

To overcome these challenges, in [1] the authors introduce the model shown in Figure 1; it is a shallow feedforward neural network model used to learn 1) a distributed representation for each word (i.e. the embedding) in a given vocabulary, and 2) the (parametric) probability function for word sequences, expressed in terms of these representations.

The training dataset for the model consists of sequences $\{w_1, \ldots, w_T\}$ of words from a given vocabulary V of size $|V|$; the goal of the statistical model is to learn the (parametric) N-grams probability function $f(w_{t-1}, \ldots, w_{t-n+1}; \theta)$ of all word sequences, where we set

$$f(w_{t-1}, \ldots, w_{t-n+1}; \theta) := \hat{P}(w_t | w_{t-n+1}^{t-1}; \theta),$$

[33] `https://github.com/alexandres/lexvec#pre-trained-vectors`

[34] In (3.2): $\hat{P}(w_1 | w_1^0) := \hat{P}(w_1)$.

with probability weights for $i = 1, \dots, |V|$ and $w_i \in V$

$$f_i(w_{t-1}, \dots, w_{t-n+1}; \theta) = \hat{P}(w_t = w_i | w_{t-n+1}^{t-1}; \theta).$$

By definition, we have $f \geq 0$ and

$$\sum_{i=1}^{|V|} f_i(w_{t-1}, \dots, w_{t-n+1}; \theta) = 1.$$

We describe the construction of $f_i(w_{t-1}, \dots, w_{t-n+1}; \theta)$. Let $\mathcal{V} \to C \in \mathbb{R}^{|V| \times m}$ denote the embedding of the vocabulary $\mathcal{V}$ to $\mathbb{R}^d$ such that $w_i \mapsto C(w_i) \in \mathbb{R}^m$. The input sequence $(w_{t-1}, \dots, w_{t-n+1})$ can then be described by the concatenation

$$x := (C(w_{t-n+1}), \dots, C(w_{t-1}))^\intercal \in \mathbb{R}^{(n-1)m \times 1}.$$

The vector x will serve as feature to predict the next word w_t. We do this by the following shallow feedforward neural network (see Figure 1):

$$x \mapsto y := b + Wx + \tanh(d + Hx), \quad y \in \mathbb{R}^{|V|},$$

with biases $b \in \mathbb{R}^{|V|}$, $d \in \mathbb{R}^h$, and weight matrices $W \in \mathbb{R}^{|V| \times (n-1)m}$, $H \in \mathbb{R}^{h \times (n-1)m}$, where $h \in \mathbb{N}$ denotes the number of hidden neurons.
The conditional probabilities are then obtained by the softmax output function, i.e. we set

$$f_i(w_{t-1}, \dots, w_{t-n+1}; \theta) := \text{softmax}(y_i) = \frac{\exp(y_i)}{\sum_{j=1}^{|V|} \exp(y_j)}, \quad i = 1, \dots, |V|.$$

The resulting set θ of parameters consists of the 5-tuple

$$\theta := \{b, d, W, H, C\}.$$

The total number of parameters is then $|V|(1 + nm + h) + h(1 + (n-1)m)$. The elements of the set θ are estimated using a stochastic gradient descent algorithm [12] to the regularized log-likelihood

$$\mathcal{L} := \frac{1}{T} \sum_{t=1}^{T} \log f(w_t, w_{t-1}, \dots, w_{t-n+1}; \theta) + R(\theta),$$

where $R(\theta)$ denotes a regularization term. In [1], one has $|V| =$ 17'964, $h = 60$, $m = 100$ and $n = 6$, for a total of 11'904'264 parameters to be trained.
From a computational perspective, the bottleneck for the training of the network in [1] is the use of the softmax output layer, as its dimension scales linearly with the size $|V|$ of the vocabulary V. A first attempt to solve this issue is shown in [10], to which we refer for more details.

3.1.2 Distributed representations of words and phrases and their compositionality: `word2vec`

In 2013, Mikolov et al. [36] introduced `word2vec`, an algorithm to compute word embeddings based on two distinct architectures and using the concept of 'context'.

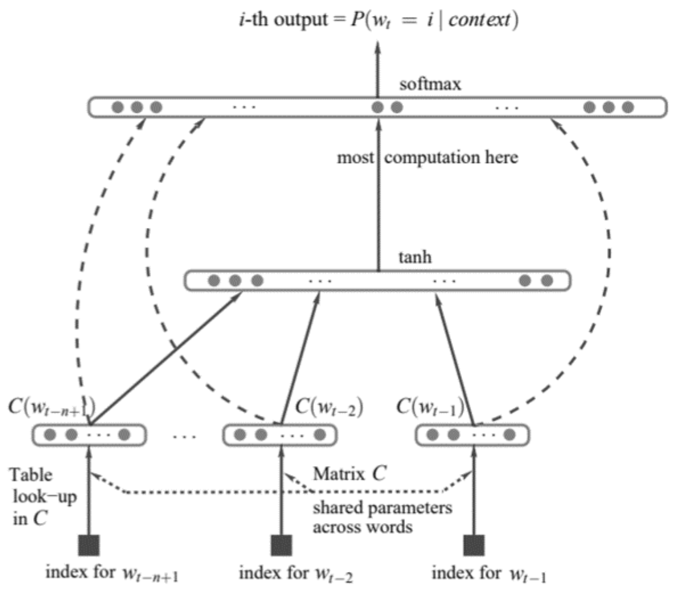


Figure 1: Neural network architecture in [1] to compute word embeddings. The vectors $C(w_{t-n+1}), \ldots, C(w_{t-1})$ are the real-valued embeddings of the words $w_{t-n+1}, \ldots, w_{t-1}$ to be learned during the training of the network. Their dimension m is much smaller than the dimension of the softmax output layer, which is equal to the size $|V|$ of the vocabulary V.

The context $C(w)$ of a word w is a finite set of words before and after w; the latter is said to be the center word of the context $C(w)$. The size of the context is encoded by a non-negative integer c, where $2c = |C(w)|$, for all words w in a given vocabulary V. For example, considering the sentence `'Koalas and platypuses are mammals living in Australia'`, and assuming $c = 2$, then the context $C(w)$[35] of the word $w =$ `'are'` is

$$C(w) = \{\texttt{'and'},\ \texttt{'platypuses'},\ \texttt{'mammals'},\ \texttt{'living'}\}.$$

The first variant of the `word2vec` algorithm uses the so called skip-gram model [35], which tries to predict the context $C(w)$ of a center word w, given that w is known in advance. Computationally, given a sequence $w_1, \ldots, w_t, \ldots, w_T$ of training words, the skip-gram model tries to maximize the average log likelihood

$$\frac{1}{T} \sum_{t=1}^{T} \sum_{-c \le j \le c, j \ne 0} \log p(w_{t+j}|w_t),$$

where $p(w_{t+j}|w_t)$ denotes the probability of 'seeing' word w_{t+j} in the context $C(w_t)$ of word w_t, assuming conditional independence of the probabilities given the center word. Therefore, the skip-gram model tries to capture word-word co-occurrences in one context window at a time. We depict the model in Figure 2. The basic skip-gram model formulation defines the probabilities $p(w_{t+j}|w_t)$ using the softmax function and 'input-output' formulation:

$$p(w_O|w_I) = \frac{\exp(\langle \tilde{v}_O, v_I \rangle)}{\sum_{w \in V} \exp(\langle \tilde{v}_w, v_I \rangle)}, \tag{3.3}$$

[35]Here we assume the context $C(w)$ to be symmetric, as we consider words both before and after w.

where $v_I \in \mathbb{R}^d$ and $\tilde{v}_O \in \mathbb{R}^d$ are the d-dimensional representations of the input word $w_I \in \mathcal{V}$ and the output word $w_O \in \mathcal{V}$, respectively. Each word w has two distinct vector representations: one when it is a center word, or 'input' in (3.3), and a second when it is a context word, or 'output' in (3.3). In summary, the skip-gram model learns $2d|V|$ parameters, where d is the dimensionality of the word embedding vectors and $|V|$ denotes the size of the vocabulary V.

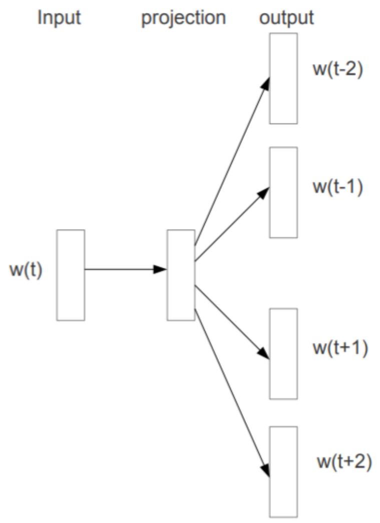


Figure 2: Figure 1 in [36], showing the skip-gram model architecture.

Computing the $2d|V|$ parameters using some variation of the gradient descent algorithm is often impractical, due to the size of the vocabulary V. Therefore, in [36], the authors introduce alternatives to the full softmax formulation like hierarchical softmax and negative sampling; we refer to [36] for all details. In the same paper, the authors consider a large news articles dataset with one billion words to train the skip-gram model. After discarding all words occurring less than 5 times in the dataset, the authors arrived at a training vocabulary of 692K words.
Lastly, in [36] the authors introduce the continuous bag of words (CBOW) model; it is a mirror model with respect to the skip-gram model, as it aims at predicting the center word from a given context by maximizing the log likelihood

$$\sum_{t=1}^{T} \log p(w_t | w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}), \tag{3.4}$$

where $\{w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}\} = C(w_t)$. Again, the conditional probabilities in (3.4) are computed using softmax and averaging along the context $C(w_t)$; we refer to [36, 48] for all details.

The Python library Gensim[36] allows users to either train `word2vec` embeddings on selected corpora, or to access pre-trained ones. We recommend the tutorial at `https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html` for additional information on this topic. Clearly, semantic and domain specificities of a given corpus are encoded by training `word2vec`

[36]`https://radimrehurek.com/gensim/`

on it; however, training word embeddings is a computationally expensive procedure and preprocessing corpora is a rather lengthy task. On the one hand, as remarked in Section 0.2 already, pre-trained embeddings may fail to intercept all linguistic features, including jargon, of the domain of applicability (e.g. health insurance). On the other hand, pre-trained word embeddings are recommended as benchmark or starting point in NLP pipelines. Pre-trained models are usually large in size (usually in the order of few GBs); one example is the Google pre-trained `word2vec` word embedding trained on Google news data (about 100 billion words) which is available at `https://code.google.com/archive/p/word2vec/`; its size is 1.5GB. However, it can be easily imported in Gensim, as shown in Listing 9 below.

```
from gensim.models import KeyedVectors
filename = 'GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(filename, binary=True)
```

Code Listing 9: Loading the Google pre-trained `word2vec` embedding in Gensim.

3.1.3 Global vectors for word representation: `Glove`

In 2014, Pennington et. al. [42] introduced `GloVe`, an unsupervised learning methodology to compute word embeddings. `GloVe` uses a weighted least squares log-linear model that is trained on global word-word co-occurrence counts from a given corpus. The adjective 'global' refers to the global corpus statistics captured directly by computing the word-word co-occurrence counts matrix, as opposed to `word2vec`, where word-word co-occurrences are captured in one context window at a time, as remarked in Section 3.1.2.
In [42] it is shown that `GloVe` outperforms other models, for example `word2vec`, on word analogy, word similarity and named entity recognition tasks; let us describe the `GloVe` algorithm in more detail. `GloVe` is essentially a log-bilinear model with a weighted least squares objective function. The main intuition behind it is the use of *ratios* of word-word co-occurrence probabilities to compute word embeddings, as these ratios are expected to encode some meaning of words in the given corpus.

On word-word co-occurrence matrices: an example. Let C be a corpus on a vocabulary V of size $|V|$. The word-word co-occurrence matrix X_C of C is a $|V| \times |V|$ matrix whose elements X_{ij} denote the number of times the word j occurs in the context[37] of word i.
For example, let C be the corpus

$$C = \{\texttt{`Cats and koalas are mammals.'}, \texttt{`Tortoises are not mammals.'}, \\ \texttt{`I like cats, koalas and tortoises.'}\}.$$

Below we show the word-word co-occurrence matrix X_C of C using a context window with $c = 1$. Before computing all co-occurrence counts, we preprocessed C by lowercasing and stripping punctuation after performing tokenization with a whitespace tokenizer, as described in Section 1.3.

[37]As for `word2vec`, the context window size c is a parameter to be fixed in advance; we have already noted that context windows can be symmetric or asymmetric (left or right)Ho. In [42], a symmetric context window with $c = 10$ is used in most empirical studies.

$$X_C = \begin{array}{c} \\ \left(\begin{array}{ccccccccc} \texttt{cats} & \texttt{and} & \texttt{koalas} & \texttt{are} & \texttt{mammals} & \texttt{tortoises} & \texttt{not} & \texttt{I} & \texttt{like} \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right) \end{array} \begin{array}{l} \\ \texttt{cats} \\ \texttt{and} \\ \texttt{koalas} \\ \texttt{are} \\ \texttt{mammals} \\ \texttt{tortoises} \\ \texttt{not} \\ \texttt{I} \\ \texttt{like} \end{array}$$

First of all, we note[38] that $(X_C)_{ii} = 0$, for all i's. Moreover, $(X_C)_{ij} = (X_C)_{ji}$; this results from the very definition of context $C(w)$. Every row of the word-word co-occurrence matrix is an integer-valued vector; therefore, it could be used as embedding of the corresponding word. However, the dimension of this embedding is equal to the size of the vocabulary and such embeddings end up being highly sparse word representations (as shown in our example above). In [42] the authors introduce a novel methodology to overcome this limitation; to explain it, we follow the original example from [42]. We introduce the quantities

$$P_{ij} := \frac{X_{ij}}{\sum_{k=1}^{|V|} X_{ik}},$$

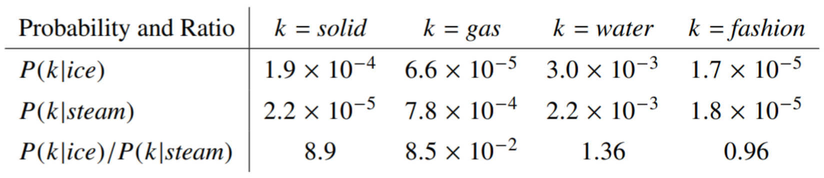
which denote the probability of word j to appear in the context of word i, given the vocabulary V of size $|V|$. Let us consider a corpus in which the words `ice`, `steam`, `water`, `solid` and `fashion` appear (among others). In Figure 3 we report their word-word co-occurrence probabilities and ratios from [42]. As we can see, only in the ratio non-discriminative words like `water` and `fashion` cancel out; so we expect that ratios of probabilities of words that correlate with properties specific to `ice` show large values, and ratios of probabilities of words that correlate strongly with properties specific to `steam` show small values, instead. Compared to word-word co-occurrence counts or probabilities, ratios of probabilities are better at discriminating relevant words (like `solid` and `gas`) from irrelevant ones (like `water` and `fashion`).
Following this observation, in [42] the authors search for a class of functions F, center word embedding vectors $v_i, v_j \in \mathbb{R}^d$ and context word embedding vectors $\tilde{v}_k \in \mathbb{R}^d$ such that

$$F(v_i, v_j, \tilde{v}_k) = \frac{P_{ik}}{P_{jk}}, \quad i, j, k = 1, \dots, |V|.$$

We note the use of distinct embeddings for words and context words as for `word2vec`.

[38]This is not a general property of word-word co-occurrence matrices. Consider, for example, the corpus $C' =$ {`‘ah ah ah’, ‘eh eh eh’`}. Its co-occurrence matrix $X_{C'}$ (again with $c = 1$) reads

$$X_C = \begin{array}{c} \\ \left(\begin{array}{cc} \texttt{ah} & \texttt{eh} \\ 4 & 0 \\ 0 & 4 \end{array}\right) \end{array} \begin{array}{l} \\ \texttt{ah} \\ \texttt{eh} \end{array}$$



Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figure 3: Table 1 in [42], showing the discriminative power of ratios of word-word co-occurrence probabilities. In our notation, $P(k|ice) = P_{ice,k}$ and $P(k|steam) = P_{steam,k}$.

The authors require: F 1) to be defined on the scalar products $\langle v_i - v_j, \tilde{v}_k \rangle$ (to encode the information in $\frac{P_{ik}}{P_{jk}}$ in a vector space and avoid 'mixing' of vector embedding dimensions), and 2) to be a group homomorphism[39] $F : (\mathbb{R}, +) \to (\mathbb{R}_{>0}, \cdot)$. Moreover, as for word-word co-occurrence matrices the distinction between center and context words is arbitrary, the equation for training the word embeddings would need to reflect this arbitrariness (called 'symmetry' in [42]). Finally, taking into consideration these desiderata, one arrives at equation (6) in [42], i.e.

$$\langle v_i, \tilde{v}_k \rangle + b_i + \tilde{b}_k = \log(X_{ik}), \tag{3.5}$$

where b_i and $\tilde{b}_k$ are scalar biases introduced to enforce 'symmetry', for all $i, k = 1, \ldots, |V|$. Equation (3.5) is encoded in a weighted least squares problem

$$J = \sum_{i,k=1}^{|V|} g(X_{ik}) \Big(\langle v_i, \tilde{v}_k \rangle + b_i + \tilde{b}_k - \log(X_{ik}) \Big)^2, \tag{3.6}$$

where $g(\cdot)$ is a weighting function; it takes care of sparsity in the word-word co-occurrence matrix, among others. In [42] the authors used

$$g(x) = \begin{cases} (\frac{x}{x_{\max}})^{\alpha} & x < x_{\max}, \\ 1 & \text{otherwise}, \end{cases}$$

with parameter $\alpha = \frac{3}{4}$ and cut-off $x_{\max} = 100$. The formulation in (3.6) shows that `GloVe` aims at minimizing the difference between the dot product of the embedding vectors of two words (including scalar biases) and the logarithm of the number of their co-occurrences. We note that the matrix X of word word co-occurrences is typically highly sparse and comprises $|V|^2$ elements. `GloVe` has been originally trained by stochastic sampling from the word-word co-occurrence matrix X, using the AdaGrad gradient descent algorithm [11] on five different corpora:

1. a 2010 Wikipedia dump with 1B tokens;
2. a 2014 Wikipedia dump with 1.6B tokens;
3. Gigaword 5[40] comprising 4.3B tokens;
4. the combination Gigaword5 + Wikipedia2014, for a total of 6B tokens;
5. Common Crawl, for a total of 42B tokens.

[39] I.e. $F(a + b) = F(a) \cdot F(b)$, for all $a, b \in \mathbb{R}$.
[40] `https://catalog.ldc.upenn.edu/LDC2011T07`

To train `GloVe`, in [42] the authors start with the tokenization and the lowercasing of the selected corpus; then a vocabulary of the 400'000 most frequent words is built, together with the matrix of word-word co-occurrence counts X. For each word w_i in the corpus, the final embedding vector is given by the sum $v_i + \tilde{v}_i \in \mathbb{R}^d$ of the corresponding trained vectors. Run times for training are competitive, as discussed in [42], Section 4.6. In their empirical studies, the authors consider embedding vectors of different dimensions and multiple context window sizes for the purpose of testing `GloVe`'s accuracy; we refer to Sections 4.1 and 4.4, as well as Figure 2 in [42], for all details. Here, we mention that `GloVe` performs significantly better than all other state-of-the-art algorithms (including `word2vec`) for both semantic and syntactic analogy tasks. Again, we refer to [42], for all details. Four pre-trained `GloVe` embeddings are available at `https://nlp.stanford.edu/projects/glove/`; the corpora used for training are:

- Wikipedia 2014 + Gigaword 5, resulting in 6B tokens, and vectors with 50, 100, 200 and 300 dimensions;
- Common Crawl[41], with 42B tokens, uncased text, and vectors with 300 dimensions;
- Common Crawl, with 840B tokens, cased text, and vectors with 300 dimensions;
- Twitter, with 2B tweets and 27B tokens, uncased text, and vectors with 25, 50, 100 and 200 dimensions.

The pre-trained embeddings are stored in .txt files; therefore, they can easily be imported in Python using the `numpy` and `pandas` libraries. However, we warn the interested reader: each pre-trained `GloVe` embedding file is rather large; for example, the one trained on Twitter data has a size of 1.42GB. Similarly to `word2vec`, a Python library— called Glove[42]—allows users to train `GloVe` embeddings on selected corpora. The same caveats highlighted for the training of `word2vec` embeddings apply here; we refer to the official `GloVe` documentation for all details.

3.1.4 Pre-trained word embeddings in `SpaCy`

We now turn our attention to the pre-trained word embedding models which are available in `SpaCy`. Pre-trained word embedding models in `SpaCy` are named after three main features: 1) type, or the model capability, 2) genre, or the type of text the model is trained on, and 3) size, which denotes the size of the model. For example, the model `en_core_web_md`[43] is a core (e.g. a general purpose model with word vectors, among others), medium sized model trained on written text (blogs, news, comments) from the World Wide Web. More precisely, this model is a "multi-task CNN trained on OntoNotes[44], with GloVe vectors trained on Common Crawl. [The model] assigns word vectors, context-specific token vectors, POS tags, dependency parse and named entities."[45]. According to the official documentation, all the `SpaCy` models for English language have 300 components, i.e. $d = 300$ in (3.1). For example, in Listing 10 we retrieve the pre-trained embeddings for all the tokens in the `text` sentence using `en_core_web_md`; by definition, the embedding of the whole sentence (`doc.vector`) is given by the sum of the embeddings of all tokens generated from the sentence.

[41]`https://commoncrawl.org/`

[42]`https://pypi.org/project/glove/`

[43]`https://SpaCy.io/models/en#en_core_web_md`

[44]`https://catalog.ldc.upenn.edu/LDC2013T19`

[45]`https://SpaCy.io/models/en`

```
# importing spacy
import spacy

# load the spacy word embedding for English Language
nlp = spacy.load('en_core_web_md')

# we use nlp() on the reference sentence ‘text’
doc = nlp("In H.P. Lovecraft's short story 'The Call of Cthulhu', the author states
that in S. Latitude 47° 9', W. Longitude 126° 43' the great Cthulhu dreams
in the sea-bottom city of R'lyeh.")

# we print the first 5 components of the ‘text’ embedding
print(doc.vector[:5])

[0.10901878  0.25519025  0.09448721 -0.05869794  0.20765139]
```

Code Listing 10: Using `SpaCy` pre-trained embeddings on `text`.

```
# defining pairs of words of interest and computing their cosine similarity
a=round(nlp('Latitude').similarity(nlp('Longitude')), 2)
b=round(nlp('love').similarity(nlp('hate')), 2)
c=round(nlp('mathematics').similarity(nlp('physics')), 2)
d=round(nlp('dogs').similarity(nlp('dog')), 2)
e=round(nlp('supernova').similarity(nlp('mouse')), 2)
print(a, b, c, d, e)

0.45 0.64 0.73 0.88 0.05
```

Code Listing 11: Computing some examples of cosine similarity between pre-trained `SpaCy` word embeddings.

In Listing 11 we compute the cosine similarities of few word embeddings using `en_core_web_md`. The `SpaCy` model suggests a moderate similarity between the words ‘Latitude’ and ‘Longitude’, considering the (web-based) training corpora. On the other hand, it returns a cosine similarity of 0.73 for the words ‘mathematics’ and ‘physics’, as well as of 0.88 for the pair ‘dogs’ and ‘dog’. Quite unsurprisingly, the embeddings of the words ‘supernova’ and ‘mouse’ are close to orthogonality. In the Appendix, we collect additional resources to perform multilingual NLP, with a focus on pre-trained word embeddings in the German, French and Italian languages.

3.2 Machine learning with word embeddings

We conclude this overview of word embeddings describing the pipeline we will use in Section 6 to train machine learning models for the classification of text documents. The pipeline comprises the steps (we follow the same notation from Section 2):

1. import of the dataset $\mathcal{D}$ (e.g. in Python);
2. partition $\mathcal{D} = \mathcal{D}_{train} \sqcup \mathcal{D}_{test}$ of $\mathcal{D}$ into train and test datasets;
3. run of hyperparameter tuning using k-fold cross-validation on $\mathcal{D}_{train}$ by performing:
   (a) the computation of the word embeddings of each text document $x_i \in \mathcal{X}$ as result of the tokenization of the document x_i, and the averaging of the token embeddings;
   (b) machine learning modeling, selecting different classes $\mathcal{H}$.
4. retrieval of the ‘best’ combination of NLP preprocessing and machine learning model per each class $\mathcal{H}$ by computing performance measures on validation sets.

4 Classical and Modern Approaches: machine learning classifiers

In this section, we discuss the Python implementation of the machine learning classifiers we use in Section 6 to automatically classify text documents, after 'bag-of-' modeling described in Section 2 or the use of word embeddings, which we discussed in Section 3.
In this tutorial we use 1) random forests, 2) adaptive boosting, and 3) extreme gradient boosting algorithms to classify text documents. We refer to [12] for boosting methods. In Python, random forests are implemented in the `sklearn` function `RandomForestClassifier()`[46], which is shown in Listing 12.

```
RandomForestClassifier(n_estimators='warn', criterion='gini', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None,
random_state=None, verbose=0, warm_start=False, class_weight=None)
```

Code Listing 12: sklearn function `RandomForestClassifier()`.

For the case study in Section 6 we will perform hyperparameter tuning on `n_estimators` and `max_depth`; the first parameter controls the number of trees trained in the ensemble, while the second allows to specify their depth. For classification problems, `RandomForestClassifier()` uses the function `DecisionTreeClassifier()` to train each decision tree in the ensemble. Trees are grown either using either Gini impurity or information gain; these values are controlled by the `criterion` parameter. The function `DecisionTreeClassifier()` does not support categorical variables[47], as discussed in [12]. This means that `RandomForestClassifier()` tries to convert categorical variables into numerical ones, before running tree growing routines. However, this does not happen when considering NLP preprocessing with `TfidfVectorizer()` or computing word embeddings, as in these cases all variables are numerical, by construction.

Adaptive and extreme gradient boosting algorithms are described in detail in [12], to which we refer for all details. In Python, the `sklearn` function `AdaBoostClassifier()`[48] provides the practitioner with an implementation of adaptive boosting algorithms; its interface is shown in Listing 13.

```
AdaBoostClassifier(base_estimator=None, n_estimators=50, learning_rate=1.0,
algorithm='SAMME.R', random_state=None)
```

Code Listing 13: The sklearn function `AdaBoostClassifier()`.

`AdaBoostClassifier()` can use different classes of base learners (if their training algorithms support re-weighting of data points); they are specified controlling the parameter `base_estimator`. It uses decision tree stumps as base learners, per default. In fact, `base_estimator=None` in

[46] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[47] https://github.com/scikit-learn/scikit-learn/issues/12398

[48] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#id2

Listing 13 is equivalent to use `DecisionTreeClassifier(max_depth=1)`. However, as remarked in [13], it is recommended to increase tree depth in hyperparameter tuning routines to capture nonlinearities in data already with base learners. In this tutorial, we will consider only decision trees as base learners. Finally, the parameter `learning_rate` allows us to control the learning rate (see Section 3.13 in [12]) during training of the adaptive boosting ensemble.

XGBoost, i.e. eXtreme Gradient Boosting, is a boosting algorithm introduced by Chen and Guestrin in [7]. It is one of the most used and best performing boosting algorithms; it tackles both regression and binary classification problems in an unified framework, by performing a sequential Newton approximation of a regularized functional (loss function + regularization term) to boost base learners (e.g. tree stumps). It accepts only trees as base learners, and the choice of the loss function to be optimized depends on the learning problem at hand.
`XGBClassifier()`[49] is the default `sklearn` Python API function for classification problems with XGBoost; we show its interface in Listing 14.

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1, colsample_bytree=1,
gamma=0, learning_rate=0.1, max_delta_step=0, max_depth=3,  min_child_weight=1,
missing=None, n_estimators=100, n_jobs=1, nthread=None, objective='binary:logistic',
random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None, silent=True,
    subsample=1)
```

Code Listing 14: The `sklearn` function `XGBClassifier()`.

In Section 6 we tune `XGBClassifier()`'s parameters `learning_rate` and `n_estimators`, which denote the learning rate and the number of boosting iterations, similarly to what we discussed in the case of `AdaBoostClassifier()`. However, we note that the default formulation of `XGBClassifier()` proposes a stronger shrinkage and a higher number of estimators with respect to the default `AdaBoostClassifier()`. In fact, in `XGBClassifier()` the default depth of trees is set to `max_depth=3`. Similarly to `RandomForestClassifier()` and `AdaBoostClassifier()`, `XGBClassifier()` can deal only with numerical data columns; therefore, encoding of categorical variables is needed.

5 Contemporary Approach: deep learning with recurrent neural networks

Both the classical and modern approaches described in Sections 2 and 3 rely on extensive preprocessing of the raw text data before any machine learning classifier can be applied, and the performance of the models depends highly on the quality of the performed preprocessing. In other words, the task of extracting and engineering suitable representations of the raw text data for the machine learning algorithm to work falls to the data scientist or actuary (although this task can be facilitated by the use of NLP libraries, e.g. `sklearn` and its `TfidfVectorizer()` function).
As discussed in the **Classical Approach**, bag-of-word models fail to encode word order, unless n-gram models are implemented. Moreover, bag-of-word models suffer from high dimensionality and sparseness of tf-idf word-document matrices. A more advanced **Modern Approach** to the

[49] `https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBClassifier`

task of NLP preprocessing and classification of text documents with machine learning has been taken in Section 3, where an initial data representation (within a low dimensional real-valued vector space) is learned by an *ad-hoc* algorithm, before the application of a traditional machine learning classifier.

The **Contemporary Approach** outlined in this section also uses this key idea that the learning algorithm should automatically discover the representations needed for the classification task, and takes it one step further by using a single neural network to fulfill most of the NLP preprocessing task and the machine learning classification simultaneously. This simultaneous approach may tailor the preprocessing more specifically to the classification task, and limits the preprocessing steps the data scientist or actuary has to perform by hand (we note that tokenization and vectorization of the raw text data will still need to be carried out beforehand).
Deep learning architectures such as neural networks excel at discovering intricate structures in high-dimensional data by using multiple layers of data representations [28]. However, the additional task of automatic representation learning unsurprisingly comes at a steep computational price. Recent developments in deep learning (from 2010 on) and the advent of graphical processing units (GPUs), have allowed practitioners to improve the performance achieved by deep learning approaches considerably, for NLP tasks as well as in other domains.

A Recurrent Neural Network (RNN) is a neural network architecture suited to deal with sequential data (such as texts, audio and video [17, 49]). RNNs are a type of neural networks that allow for cycles in their architecture; in contrast to the classical feedforward neural networks, at every step RNNs take into account the activations of the previous step. Therefore, RNNs are said to possess a sort of 'memory' of the structure of the training data. This memory-related functionality is, of course, highly valuable when dealing with sequential data. In what follows, we start introducing some theory behind RNNs, and continue with the description of two of the most used RNN architectures.

5.1 Introducing recurrent neural networks

We introduce the definition of recurrent neural networks, starting with a short section on sequential data and moving to the discussion of single layer architectures. In our exposition we follow the notation in [13, 47], to which we refer for further details.

Sequential data. We recall some definitions from Section 2, for the sake of readability. Let $\mathcal{D}$ be a dataset $\mathcal{D} = \{(Y_1, x_1), \dots, (Y_n, x_n)\}$, with responses $Y_i \in \mathcal{Y} \subset \mathbb{R}$, and features $x_i \in \mathcal{X}$ for all $i = 1, \dots, n$. In what follows, we assume that any input $x \in \mathcal{X}$ is a finite sequence of real-valued vectors, i.e. $x = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_T)$, for some fixed $T \in \mathbb{N}$, with $\boldsymbol{x}_t \in \mathbb{R}^{\tau_0}$ for all $t = 1, \dots, T$ and $\tau_0 \in \mathbb{N}$. For example, in the context of NLP, $\mathcal{X}$ could represent a corpus, x a text document, $(\boldsymbol{x}_t)_{t=1,\dots,T}$ the sequence of embeddings of words in x, and $\mathcal{Y} = \{0, 1\}$[50] a (binary) sentiment

[50]Note that we could generalize to $\mathcal{Y} \subset \mathbb{R}^k$, $k \in \mathbb{N}$, i.e. modeling an input series to an output series is also possible with RNN architectures. However, in this tutorial we are only interested in the final output after the whole input series of length T has been processed through the network. Hence we have a one-dimensional response $Y = Y_T \in \mathbb{R}$ (i.e. we consider only 'many-to-one' RNN architectures, see `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks`). The extension of the given definitions to a higher

of each document $x \in \mathcal{X}$. Therefore, given a corpus $\mathcal{X}$ consisting of text documents x, we can fulfill the above assumptions by

1. tokenizing each document x (e.g. with `TfidfVectorizer()`);
2. slicing and/or padding each $x \in \mathcal{X}$ with 0's to ensure that the resulting sequences of words are all of equal length T;
3. performing a word embedding (see Section 3) to map each word into $\mathbb{R}^{\tau_0}$, $\tau_0 \in \mathbb{N}$, for all $x \in \mathcal{X}$.

Single layer RNN architecture. Let $\tau_1 \in \mathbb{N}$ and $\phi : \mathbb{R} \to \mathbb{R}$ be an activation function, see [13]. With $\langle \cdot, \cdot \rangle$ we denote the standard scalar product in $\mathbb{R}^n$. A single layer RNN architecture with τ_1 hidden neurons and activation function ϕ is defined by the mapping

$$\boldsymbol{z}^{(1)} : \mathbb{R}^{\tau_0 \times \tau_1} \to \mathbb{R}^{\tau_1}, \quad (\boldsymbol{x}_t, \boldsymbol{z}_{t-1}) \mapsto \boldsymbol{z}_t^{(1)} := \boldsymbol{z}^{(1)}(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}), \tag{5.1}$$

where the hidden layer reads

$$\begin{aligned} \boldsymbol{z}_t^{(1)} &= \left(\phi\left(\langle w_1^{(1)}, \boldsymbol{x}_t \rangle + \langle u_1^{(1)}, \boldsymbol{x}_t \rangle \right), \ldots, \phi\left(\langle w_{\tau_1}^{(1)}, \boldsymbol{x}_t \rangle + \langle u_{\tau_1}^{(1)}, \boldsymbol{x}_t \rangle \right) \right)^\top \\ &:= \phi\left(\left\langle W^{(1)}, \boldsymbol{x}_t \right\rangle + \left\langle U^{(1)}, \boldsymbol{z}_{t-1}^{(1)} \right\rangle \right) \in \mathbb{R}^{\tau_1}, \end{aligned}$$

and with neurons

$$\phi\left(\langle w_j^{(1)}, \boldsymbol{x}_t \rangle + \langle u_j^{(1)}, \boldsymbol{x}_t \rangle \right) := \phi\left(w_{j,0}^{(1)} + \sum_{k=1}^{\tau_0} w_{j,k}^{(1)} \boldsymbol{x}_{t,k} + \sum_{k=1}^{\tau_0} u_{j,k}^{(1)} \boldsymbol{z}_{t-1,k} \right), \tag{5.2}$$

for $t \in \{1, \ldots, T\}$ and $j \in \{1, \ldots, \tau_1\}$. We set $\boldsymbol{z}_0^{(1)} := 0 \in \mathbb{R}^{\tau_1}$, which indicates our lack of previous knowledge about the input sequence at time $t = 0$. The network parameters are collected in the matrices $W^{(1)} \in \mathbb{R}^{\tau_1 \times (\tau_0 + 1)}$ and $U^{(1)} \in \mathbb{R}^{\tau_1 \times \tau_1}$. We note that the bias is included in $W^{(1)}$ as the 0-th column (therefore the elements $w_{j,0}^{(1)}$ in (5.2)), but no such inclusion is present in $U^{(1)}$. We also note that all parameters in (5.2) are independent of time; they are shared between all timesteps (or loops). Hence, the same hidden layer is revisited at each timestep (recurrent visits): this is called the folded representation of the RNN, see Figure 4. Alternative but equivalent formulations for single layer RNNs can be found in [17] by the interested reader. In summary, the RNN architecture above computes a sequence $(\boldsymbol{z}_1^{(1)}, \ldots, \boldsymbol{z}_T^{(1)})$ of outputs, where each output is a function of the feature information at the same timestep and the output information from the previous timestep, i.e. $\boldsymbol{z}_{j+1}^{(1)} = \boldsymbol{z}^{(1)}(\boldsymbol{x}_{j+1}, \boldsymbol{z}_j^{(1)}) \in \mathbb{R}^{\tau_1}$, for all $j = 1, \ldots, T-1$. It follows that the latest output $\boldsymbol{z}_T^{(1)}$ is a function of all feature information $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ of an input $x \in \mathcal{X}$; therefore, the output $\hat{Y}_T$ for the input sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ computed by the network is written as

$$\hat{Y}_T := \varphi\langle v, \boldsymbol{z}_T^{(1)} \rangle \in \mathcal{Y},$$

where $v \in \mathbb{R}^{\tau_1 + 1}$ are (time independent) output weights (again, including bias), and $\varphi : \mathbb{R} \to \mathcal{Y}$ is an appropriate output activation function.

Multi-layer RNN architectures. In case of more complex RNN architectures, there are different ways to stack and connect hidden layers. For simplicity, let us consider the case of an RNN

dimensional output is straightforward.

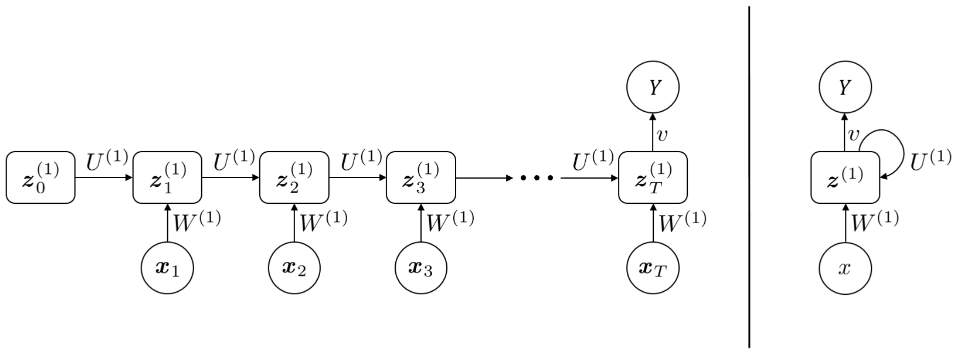


Figure 4: Unfolded single layer RNN architecture for a many-to-one mapping (left); folded graph of the same single layer RNN (right). Here $\boldsymbol{z}^{(1)}$ denotes the vector of hidden states, $W^{(1)}$, $U^{(1)}$ and v are the time independent matrices of parameters. Adapted from [46].

with two hidden layers, i.e. two mappings $\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}$ as in (5.1). One way of connecting layers is to let $\boldsymbol{z}_t^{(1)}$ feed only on input $\boldsymbol{x}_t$ and $\boldsymbol{z}_{t-1}^{(1)}$, while $\boldsymbol{z}_t^{(2)}$ feeds on $\boldsymbol{z}_t^{(1)}$ and $\boldsymbol{z}_{t-1}^{(2)}$. A second variant could consider also the contribution of $\boldsymbol{z}_{t-1}^{(2)}$ in the computation of $\boldsymbol{z}_t^{(1)}$; we refer to [17, 47] for additional details on connecting the layers in deep RNNs.

Training RNNs. Training an RNN does not essentially differ from training a feedforward neural network. Depending on the sequence modeling context at hand (we remind the reader that we discuss only the 'many-to-one' case in these notes), one has to select a loss function $\mathcal{L}$ and a parameter updating algorithm. The most commonly used gradient-based learning algorithm to update RNN parameters is Backpropagation Through Time (BPTT) [17], which is applied to the unfolded network in order to compute the gradients at each timestep [41]. However, the necessary computations for these gradients are recursive and thus often lead to either *exploding* or *vanishing gradients* with respect to time [2]. This means that weights may oscillate wildly, or they are unable to retain any time-distant activations, i.e. making it impossible to learn any long-term correlations in the data [41]. The problem of exploding or vanishing gradients has been tackled via the use of different learning algorithms (e.g. truncated BPTT, [52]) as well as the design of specialized RNN architectures, called gated RNNs [17]. The most common (and successful) implementations pf such gated RNNs are the Long Short-Term Memory (LSTM) unit [21] and its simplified variant, the Gated Recurrent Unit (GRU) [8]. The LSTM and GRU architectures are described in the following two sections.

5.2 On long short-term memory units

By using LSTM units one replaces hidden neurons in RNN architectures by more complex memory cells. Each memory cell consists of three gates that control the flow of information through the cell: they are named the forget, the input and the output gate. We note that the forget gate does not appear in the original paper [21]; it has been added in [15]. A memory cell at timestep $t \in \{1, \ldots, T\}$ takes three inputs: the input features $\boldsymbol{x}_t$, the neural activations $\boldsymbol{z}_{t-1}^{(1)}$, and a so-called cell state $\boldsymbol{c}_{t-1}^{(1)}$ from the previous timestep. After processing the inputs through its three gates, the memory cell outputs a new cell state $\boldsymbol{c}_t^{(1)}$ as well as the new neural activations $\boldsymbol{z}_t^{(1)}$. The unfolded structure of a single LSTM memory cell is shown in Figure 5.

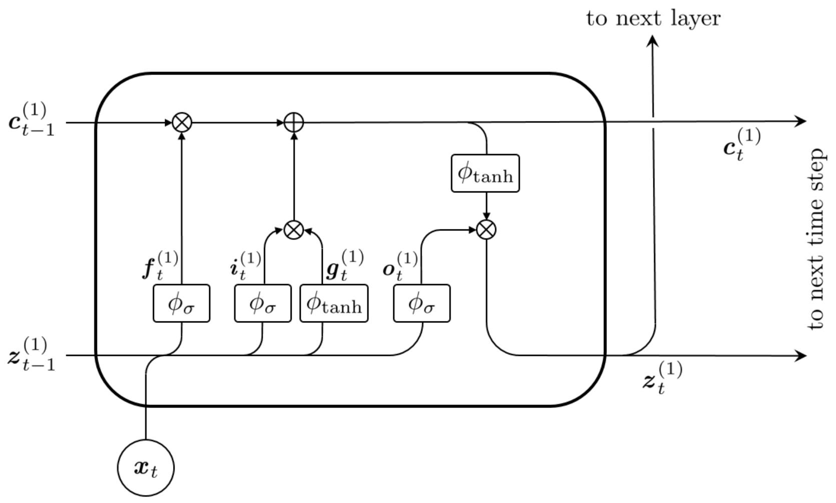


Figure 5: LSTM memory cell, $\oplus$ and $\otimes$ denote element-wise summation and multiplication, respectively. Adapted from [46].

We now proceed to describe the three gates of the memory cell and the LSTM computations in some detail. We start by introducing the sigmoid and hyperbolic tangent activation functions:

$$\begin{aligned}\phi_\sigma(x) &= \frac{1}{1+e^{-x}} \in (0,1),\\ \phi_{\tanh}(x) &= 2\phi_\sigma(2x) - 1 \in (-1,1).\end{aligned}$$

At each timestep $t = 1, \dots, T$, the LSTM memory cell processes the inputs $\boldsymbol{x}_t \in \mathbb{R}^{\tau_0}$, $\boldsymbol{z}_{t-1}^{(1)}, \boldsymbol{c}_{t-1}^{(1)} \in \mathbb{R}^{\tau_1}$ through the following gates (all functions are understood element-wise):

- *Forget gate* (loss of memory gate):

$$\boldsymbol{f}_t^{(1)} = \phi_\sigma\left(\left\langle W_f, \boldsymbol{x}_t\right\rangle + \left\langle U_f, \boldsymbol{z}_{t-1}^{(1)}\right\rangle\right) \in (0,1)^{\tau_1}, \tag{5.3}$$

- *Input gate* (memory update gate):

$$\boldsymbol{i}_t^{(1)} = \phi_\sigma\left(\left\langle W_i, \boldsymbol{x}_t\right\rangle + \left\langle U_i, \boldsymbol{z}_{t-1}^{(1)}\right\rangle\right) \in (0,1)^{\tau_1}, \tag{5.4}$$

- *Output gate* (release of information memory gate):

$$\boldsymbol{o}_t^{(1)} = \phi_\sigma\left(\left\langle W_o, \boldsymbol{x}_t\right\rangle + \left\langle U_o, \boldsymbol{z}_{t-1}^{(1)}\right\rangle\right) \in (0,1)^{\tau_1}. \tag{5.5}$$

The gate variables $\boldsymbol{f}_t^{(1)}$, $\boldsymbol{i}_t^{(1)}$ and $\boldsymbol{o}_t^{(1)}$ in (5.3), (5.4) and (5.5) are combined together to compute the state $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$ at time t. To do so, let $\circ$ denote the Hadamart product of vectors, i.e. $(v \circ w)_i = v_i w_i$ for any $v, w \in \mathbb{R}^m$, $m \in \mathbb{N}$; and we compute the memory cell state $\boldsymbol{c}_t^{(1)}$ at time t as follows:

$$\boldsymbol{c}_t^{(1)} = \boldsymbol{c}^{(1)}(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}, \boldsymbol{c}_{t-1}^{(1)}) := \boldsymbol{f}_t^{(1)} \circ \boldsymbol{c}_{t-1}^{(1)} + \boldsymbol{i}_t^{(1)} \circ \boldsymbol{g}_t^{(1)} \in \mathbb{R}^{\tau_1}, \tag{5.6}$$

where

$$\boldsymbol{g}_t^{(1)} = \phi_{\tanh}\left(\langle W_c, \boldsymbol{x}_t\rangle + \langle U_c, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in (-1,1)^{\tau_1},$$

with $W_c \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ and $U_c \in \mathbb{R}^{\tau_1 \times \tau_1}$. Finally, we arrive at

$$\boldsymbol{z}_t^{(1)} = \boldsymbol{z}^{(1)}(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}, \boldsymbol{c}_{t-1}^{(1)}) := \boldsymbol{o}_t^{(1)} \circ \phi(\boldsymbol{c}_t^{(1)}) \in \mathbb{R}^{\tau_1},$$

where $\phi = \phi_{\tanh}$ in [21], although other activation functions can be used. The (time independent) parameters of the LSTM architecture are $W_f, W_i, W_o, W_c \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ and $U_f, U_i, U_o, U_c \in \mathbb{R}^{\tau_1 \times \tau_1}$; an LSTM layer involves $4((\tau_0+1)\tau_1 + \tau_1^2)$ parameters.

Let us now discuss the above formulae in some detail. The *forget gate* output $\boldsymbol{f}_t^{(1)}$ at time t (5.3) has components in $(0,1)$ and updates the memory cell state (5.6) by rescaling the components of the memory cell state $\boldsymbol{c}_{t-1}^{(1)}$ at time $t-1$; thus, the forget gate controls how much information from the memory cell state $\boldsymbol{c}_{t-1}^{(1)}$ flows in the update of the hidden neuron activations $\boldsymbol{z}_t^{(1)}$ at time t.
The *input gate* $\boldsymbol{i}_t^{(1)}$ at time t is defined in (5.4); it has components in $(0,1)$, as well. The idea is that the input gate $\boldsymbol{i}_t^{(1)}$ 'protects' the cell state $\boldsymbol{c}_t^{(1)}$ from perturbation by irrelevant inputs [21]; this is achieved by scaling of the candidate values $\boldsymbol{g}_t^{(1)}$ with the components of $\boldsymbol{i}_t^{(1)}$; the result is added to the new cell state in (5.6). We note that the candidate values $\boldsymbol{g}_t^{(1)}$ at time t are derived from input variable $\boldsymbol{x}_t$ and the neuron activations $\boldsymbol{z}_{t-1}^{(1)}$ only.
The *output gate* $\boldsymbol{o}_t^{(1)}$ at time t shown in (5.5) controls the update of the neuron activations by rescaling the components of the memory cell state $\phi(\boldsymbol{c}_t^{(1)})$. Therefore, the output gate 'protects' the neuron activations from perturbation by currently (i.e. at time t) irrelevant content stored in the memory cell [21].

Variants of the presented LSTM architecture can be introduced, e.g. by defining so-called peephole connections between the cell state and one or more of the internal gates [16]; however, the different variants seem achieve similar performance [18].

5.3 On gated recurrent units

One drawback of the LSTM architecture is represented by its complexity. In 2014, Cho et al. [8] introduced the GRU network, which uses a similar but simpler memory cell architecture than the LSTM while achieving comparable training results. In essence, the GRU memory cell combines the input and forget gates of the LSTM cell into a single update gate, and merges the cell state with the hidden state of the cell, see Figure 6.
At each timestep $t = 1, \ldots, T$, the GRU architecture processes the inputs $\boldsymbol{x}_t \in \mathbb{R}^{\tau_0}$, $\boldsymbol{z}_{t-1}^{(1)} \in \mathbb{R}^{\tau_1}$ through the following gates:

- *Reset gate*:

$$\boldsymbol{r}_t^{(1)} = \phi_\sigma\left(\langle W_r, \boldsymbol{x}_t\rangle + \langle U_r, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in (0,1)^{\tau_1}, \tag{5.7}$$

- *Update gate*:

$$\boldsymbol{u}_t^{(1)} = \phi_\sigma\left(\langle W_u, \boldsymbol{x}_t\rangle + \langle U_u, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in (0,1)^{\tau_1}, \tag{5.8}$$

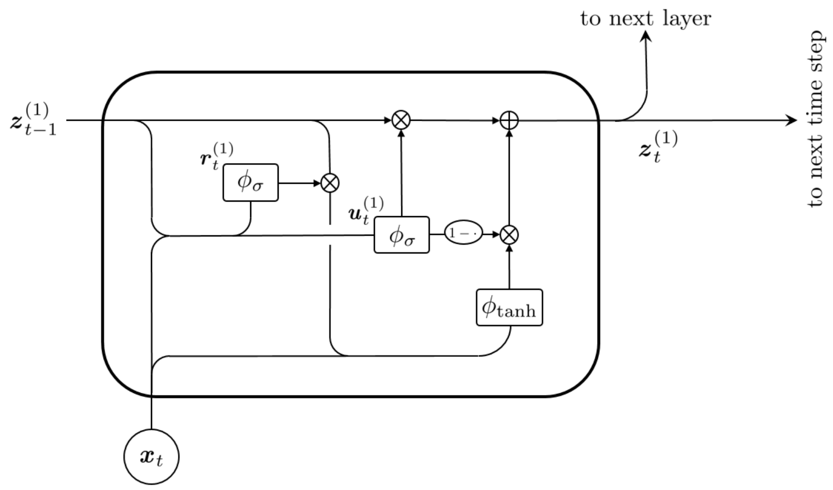


Figure 6: GRU memory cell, $\oplus$ and $\otimes$ denote element-wise summation and multiplication, respectively.

The gate variables $\boldsymbol{r}_t^{(1)}$ in (5.7) and $\boldsymbol{u}_t^{(1)}$ in (5.8) are combined together to compute the neuron activation $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$ at time t; explicitly:

$$\boldsymbol{z}_t^{(1)} = \boldsymbol{z}^{(1)}(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}^{(1)}) = \boldsymbol{u}_t^{(1)} \circ \boldsymbol{z}_{t-1}^{(1)} + (\boldsymbol{1} - \boldsymbol{u}_t^{(1)}) \circ \phi_{\tanh}\left(\langle W, \boldsymbol{x}_t\rangle + \boldsymbol{r}_{t-1}^{(1)} \circ \langle U, \boldsymbol{z}_{t-1}^{(1)}\rangle\right) \in \mathbb{R}^{\tau_1},$$

where $W_r, W_u, W \in \mathbb{R}^{\tau_1 \times (\tau_0+1)}$ and $U_r, U_u, U \in \mathbb{R}^{\tau_1\times\tau_1}$ contain the $3((\tau_0+1)\tau_1 + \tau_1^2)$ parameters to be estimated during training.
In summary, the update gate $\boldsymbol{u}_t^{(1)}$ controls the update of the neuron activations like conditional leaky integrators that "can linearly gate any dimension[51], thus choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme)" [17]. On the other hand, the reset gate $\boldsymbol{r}_{t-1}^{(1)}$ adds a nonlinear effect in the relationship between $\boldsymbol{z}_{t-1}^{(1)}$ and $\boldsymbol{z}_t^{(1)}$ by rescaling the components of $\langle U, \boldsymbol{z}_{t-1}^{(1)}\rangle$. We refer to the literature [8, 9] for additional details and remarks on GRUs.

5.4 RNNs in TensorFlow 2.0 with Keras

In this section, we discuss how to implement RNNs in Python using the Keras[52] API of TensorFlow 2.0[53]. We will apply multiple RNN implementations in the case study presented in Section 6.
TensorFlow is an end-to-end open source platform for high performance numerical computation, in particular, for machine and deep learning tasks. Computations can easily be deployed across a variety of processors (CPUs, GPUs, TPUs) using its APIs for different computing languages, in particular for Python. The Python library `tensorflow` allows the user to develop, train and evaluate machine learning models directly or by using wrapper libraries. A user-friendly wrapper

[51]i.e. dimensions of $\boldsymbol{z}_{t-1}^{(1)}$
[52]`https://keras.io/`
[53]`https://tensorflow.org/`. Tensorflow 2.0 has been released on 30.09.2019, see `https://www.tensorflow.org/guide/effective_tf2/`

library for deep learning models is Keras with its `Sequential` API. As of TensorFlow 2.0, the Keras API is included in the `tensorflow` library as `tensorflow.keras`, or `tf.keras` for short. To design a neural network in `tf.keras`, one starts by creating an empty model and stacking one layer onto the next sequentially. Let us discuss an easy example with a single hidden RNN layer, for sake of readability.

```
# importing keras from tensorflow
from tensorflow import tensorflow.keras

# creating a new model
model = Sequential()

# adding an input (embedding) layer
model.add(layers.Embedding(input_dim, output_dim, embeddings_initializer='uniform',
    embeddings_regularizer=None, activity_regularizer=None, embeddings_constraint=None,
    mask_zero=False, input_length=None))

# adding a recurrent (simple RNN) layer
model.add(layers.SimpleRNN(units, activation='tanh', use_bias=True, kernel_initializer='
    glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros',
    kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None,
    activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None,
    bias_constraint=None, dropout=0.0, recurrent_dropout=0.0, return_sequences=False,
    return_state=False, go_backwards=False, stateful=False, unroll=False))

# adding an ouput (dense) layer
model.add(layers.Dense(units, activation=None, use_bias=True, kernel_initializer='
    glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=
    None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None))
```

Code Listing 15: Building a simple RNN model with the Keras `Sequential` API

Let $\mathcal{X}$ be a corpus of text documents (following, once again, the notation from Section 2); in Listing 15 we show how to define a single layer RNN to classify the documents in $\mathcal{X}$. The RNN model takes input data as an array of integers; each row in the array corresponds to a text document in $\mathcal{X}$. Each document is represented by a sequence of integers (of fixed length T), where each integer uniquely identifies the words extracted from the whole corpus $\mathcal{X}$: this is the starting representation of input data. The `Embedding` layer is used to compute an embedding of each word in $\mathbb{R}^{\tau_0}$, with `output_dim=`τ_0. The parameter `input_dim` denotes the size of the vocabulary of $\mathcal{X}$, while the parameter `input_length` corresponds to the fixed length of sequences of words, i.e. T. We note that if no `input_length` is specified, this value is inferred automatically from the input data. As next step, the recurrent layer `SimpleRNN` is stacked on top of the embedding layer. The parameter `units` controls the number of hidden neurons (τ_1 in Section 5.1). The other input parameters can be used to customize the RNN layer, e.g. by choosing different activation functions. We note that, by default, `return_sequences=False`. Hence, only the last timestep (i.e. $\boldsymbol{z}_T^{(1)}$) is returned by the RNN model. If the whole sequence of neural activations has to be returned, for example when the first recurrent layer is followed by a second (recurrent) one or if we consider the case of a many-to-many architecture, `return_sequences` needs to be set to `True`. To use different RNN architectures such as an LSTMs or GRUs, one can simply replace `layers.SimpleRNN()` with `layers.LSTM()` or `layers.GRU()`, respectively. Finally, a densely-connected output layer `Dense` is added to the architecture to compute the output of each sequence at timestep T; the parameter `units` controls the dimensionality of the output, while `activation` allows one to choose an output activation function.

```
# compiling the model
model.compile(optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=
    None, weighted_metrics=None, target_tensors=None)
```

Code Listing 16: Compiling a model in Keras

Next, the `compile` method is used to configure the RNN training process; we show it in Listing 16. A commonly used optimizer is the Adam optimizer [23] which can be specified with `optimizer="adam"`. The choice of the loss function is controlled by `loss`: a list of all available loss functions can be found in the Keras documentation[54]. With `metrics` one can select which metrics to use to evaluate the model performance during training, validation and testing (e.g. accuracy or mean squared error).

```
# training the model
model.fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
    validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
    sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None,
    validation_freq=1, max_queue_size=10, workers=1, use_multiprocessing=False)
```

Code Listing 17: Training a model in Keras

After model compilation with `compile`, one can move to model training (or fitting), as shown in Listing 17. Input data and their outcomes (e.g. discrete labels) are specified by `x` and `y`. The length of the training process is controlled by the `batch_size` (number of samples per gradient update) and `epochs` (number of iterations over the entire dataset) parameters. A fraction `validation_split` of the training data can be used as validation data, i.e. to evaluate the loss and all model metrics at the end of each epoch. Depending on the number of parameters to be estimated, the size of the input data and the computational power of the machine under use, the training of an RNN might take a considerable amount of time.

```
# evaluating the model
results = model.evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None,
    steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False)
```

Code Listing 18: Evaluating a model in Keras

Finally, with `model.evaluate` one retrieves the loss and all the metrics values for the trained model on testing data, see Listing 18.

6 Case study: classification of movie reviews from the Internet Movie Database (IMDb)

In this section we provide a case study that compares the performance of different machine learning algorithms using data from the Internet Movie Database (IMDb), for all three approaches we discussed in these notes. The results of the case study can be reproduced with the following Jupyter Notebooks:

[54]`https://keras.io/`

1. `NLP_IMDb_Case_Study_ML.ipynb`: machine learning pipelines from Sections 2.1 and 3.2;
2. `NLP_IMDb_Case_Study_RNN.ipynb`: deep learning with RNNs from Section 5.

Both Jupyter notebooks are stored on the GitHub repository of the Fachgruppe "Data Science" of the Swiss Association of Actuaries (SAV), which is available at

`https://github.com/JSchelldorfer/ActuarialDataScience`

We run all the machine learning routines presented in this section on the ETH High Performance Computing (HPC) infrastructure Euler[55], by submitting all jobs to a virtual machine consisting of 32 cores with 3072 MB RAM per core (total RAM: 98.304 GB). All presented results are fully reproducible by the interested reader; however, due to the high number of computations, we recommend the reader to launch the training routines on adequate computing infrastructures, or to improve run time by sub-sampling training data, for example.

6.1 Getting started

6.1.1 Python and Jupyter Notebook

First of all, Python 3.X.Y has to be installed on the working machine. We recommend to install it *via* the Anaconda platform[56]. Secondly, we need to access the Jupyter Notebooks from the Fachgruppe "Data Science" GitHub by downloading them into a folder of preference. We continue by creating Anaconda environments (e.g. called `NLP_ML` and `NLP_DL`) comprising the Python packages used in the notebooks (it is recommended to install them all together for a better management of dependences) and activate them. Tensorflow 2.0—including its Keras API—can be installed with `pip install --upgrade tensorflow`.

6.2 Data import

The IMDb data [31] used in this case study are available at `http://ai.stanford.edu/~amaas/data/sentiment/` in a tar.gz archive called `aclImdb_v1.tar.gz`. The interested reader can extract the archive using, for example, 7zip[57] (the extraction is performed twice: from tar.gz to .tar, and from .tar to text files); this procedure creates a folder called `acImdb`, with subfolders `train` and `test`. Each subfolder contains 25'000 .txt movie review files, which are equally distributed among the subfolders `neg` and `pos` of negative and positive reviews.

6.3 Movie reviews analysis

In the Jupyter notebook the 50'000 files are joined together and movie reviews are saved in a `pandas` data frame. In summary, we have 50'000 movie reviews and their sentiment stored as rows of a data frame with columns named `review` and `sentiment`. A negative movie review shows `sentiment==0`, while a positive movie review has `sentiment==1`. Let us start by providing the reader with some examples of movie reviews. In Listing 19 we show an excerpt from a movie

[55] `https://scicomp.ethz.ch/wiki/Euler`
[56] `https://www.anaconda.com/`
[57] `https://www.7-zip.org/`

review with positive sentiment; the text comprises words in both lower and uppercase, HTML markups like < br/ > and a typo (i.e. ‘Bristish’), as well.

```
Actor turned director Bill Paxton follows up his promising debut, the Gothic-horror
"Frailty", with this family friendly sports drama about the 1913 U.S. Open where a
young American caddy rises from his humble background to play against his Bristish idol
in what was dubbed as "The Greatest Game Ever Played." I'm no fan of golf, and these
scrappy underdog sports flicks are a dime a dozen (most recently done to grand effect
with "Miracle" and "Cinderella Man"), but some how this film was enthralling all the
same.<br /><br />The film starts with some creative opening credits (imagine a
Disneyfied version of the animated opening credits of HBO's "Carnivale" and "Rome"),
but lumbers along slowly for...
```

Code Listing 19: An excerpt of movie review from the IMDb dataset with positive sentiment. We note the presence of typos and of the HTML markups < br/ >.

In Listing 20 we show an example of movie review with negative sentiment, instead. No HTML markup or special character are identified.

```
Once again Mr. Costner has dragged out a movie for far longer than necessary. Aside from
the terrific sea rescue sequences, of which there are very few I just did not care
about any of the characters. Most of us have ghosts in the closet, and Costner's
character are realized early on, and then forgotten until much later, by which time
I did not care. The character we should really care about is a very cocky,
overconfident Ashton Kutcher. The problem is he comes off as kid who thinks he's better
than anyone else around him and shows no signs of a cluttered closet. His only obstacle
appears to be winning over Costner. Finally when we are well past the half way point of
this stinker, Costner tells us all about Kutcher's ghosts. We are told why Kutcher is
driven to be the best with no prior inkling or foreshadowing. No magic here, it was all
I could do to keep from turning it off an hour in.
```

Code Listing 20: A full movie review with negative sentiment from the IMDb dataset.

The dataset contains 418 duplicated records; we remove them from the analysis that follows. We are left with 49’582 reviews, with sentiment distribution shown in Table 1.

Sentiment	Number of reviews (%)
`sentiment==0`	24’698 (49.81)
`sentiment==1`	24’884 (50.19)

Table 1: Number of movie reviews per sentiment value, after removal of duplicates.

Due to the presence of HTML markups[58] and other special characters in text (e.g. smileys ‘:-)’), we apply the preprocessing function `preprocessor()` from [46], Chapter 8, which is shown in Listing 21.

```
# introducing preprocessor()
def preprocessor(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('', text)
    text = (re.sub('', ' ', text.lower()) +
            ' '.join(emoticons).replace('-', ''))
    return text
```

[58]http://www.simplehtmlguide.com/examplesheet.php

```
# applying preprocessor() to the dataframe df of movie reviews
df['review'] = df['review'].apply(preprocessor)
```

Code Listing 21: Function to preprocess movie reviews from [46], Chapter 8.

The preprocessing function removes all HTML markups, it lowercases text, it substitutes non-alphanumeric characters (including the underscore) with whitespaces, and removes the 'nose' from emoticons. The reader can replicate the text preprocessing pipeline so far by running the Jupyter notebook `NLP_IMDb_Case_Study_ML.ipynb`.

6.3.1 Basic linguistic analysis of movie reviews

We perform some basic linguistic analysis of movie reviews before moving to machine learning routines. We start by computing the statistics of word counts per each review; to do so, we apply a whitespace tokenizer after stripping leading and trailing whitespaces. The shortest movie reviews in the provided dataset are shown in Listing 22 (one review per line); all reviews therein, except the third one, are characterized by a negative sentiment.
On average, a movie review contains 235 words; the shortest reviews are 6 words long, while the longest review comprises 2'498 words. This review contains the detailed description of a professional wrestling series of matches involving WWE[59] athletes, and it shows a positive sentiment.

```
read the book forget the movie
primary plot primary direction poor interpretation
brilliant and moving performances by tom courtenay and peter finch
i hope this group of film makers never re unites
more suspenseful more subtle much much more disturbing
what a script what a story what a mess
this movie is terrible but it has some good effects
```

Code Listing 22: Shortest movie reviews (after preprocessing); they all show `sentiment==0`, except the third one.

In Figure 7 we collect the distributions of 1) word counts, 2) average word length, and 3) stopword counts, for both negative and positive sentiment reviews. The mean of word counts for negative sentiment reviews is[60] 234 (168), while for positive sentiment reviews is 237 (180). The longest negative review is 1'550 words long. We show an excerpt of it in Listing 23. On the other hand, the average word length for both negative and positive sentiment reviews is equal to 4 (0). The maximum value of average word lengths is equal to 11 characters; this value corresponds to the single (positive) review shown in Listing 24.

```
...not to send it into hyperdrive.<br /><br />** END SPOILERS **<br /><br />Perhaps, the
Lost World plot and the turn-of-the-century setting should give me a hint that
this is more an homage to pulps. The failures I find with the film agree with this idea.
But I am at a loss why I should pay to see thin characters and plot holes simply
because many dime novels had them as well. And pulp stories is part of
the "crap they can't sell adults anymore", anyway. We have become a bit more
sophisticated and our pulp needs to grow up as well. Raiders of the Lost Ark lost
```

[59]World Wrestling Entertainment `https://www.wwe.com/`

[60]We report the standard deviation in brackets, rounded to the nearest integer.

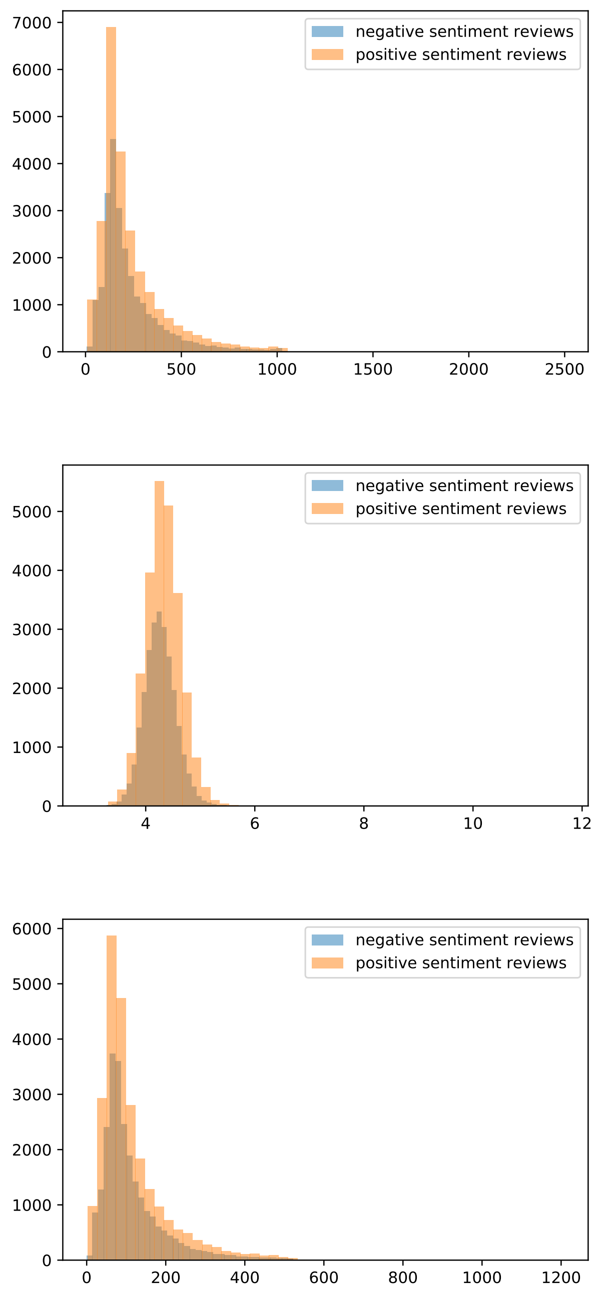


Figure 7: Distribution of word counts (top panel), average word lengths (middle panel) and stopword counts with `nltk` (bottom panel) for negative and positive movie reviews.

```
none of its pulp feel and avoided so much badness.<br /><br />4 out of 10--the movie
is enjoyable but as I think about the plot, it seeps ever lower.
```

Code Listing 23: An excerpt from the longest movie review with negative sentiment (we show the original entry, not the preprocessed one, for sake of readability).

```
whoops looks like it s gonna cost you a whopping 198 00 to buy a copy either dvd or
video format from itv direct ouch sorry about this but imdb won t let me
submit this comment unless it has at least 10 lines so
blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah
blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah
blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah
blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah
blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah
blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah blahblah
```

Code Listing 24: Movie review (after preprocessing) with the longest average word length.

The distribution of stopword counts is obtained using the stopword list in the `NLTK` library. The average number of stopwords in negative reviews is 116 (83), while in positive reviews it is 115 (87). A single review does contain no stopword: it is the second review shown in Listing 22. Considering all negative reviews, the maximum number of stopwords (i.e. 726) appears in the longest review (whose excerpt is provided in Listing 23). However, the maximum number of stopwords for positive reviews (i.e. 1'208) is not reached in the longest review, but in a review with 2'306 words.

6.4 Machine learning

We now compute and discuss the results of all approaches aimed at the classification of text documents we have presented in this tutorial.

6.4.1 Training and test datasets

To train machine learning classifiers for both the **Classical Approach** and **Modern Approach** using the pipelines from Section 2.1 and 3.2, we use a 80-20 random split of the 49'582 available data points. To do so, we apply random shuffling with seed `np.random.seed(0)`; then, we retrieve the training dataset by considering the first 39'666 records of the shuffled dataset; the test dataset consists of the remaining 9'916 records.

6.4.2 Classical Approach: machine learning pipeline

As discussed in Section 4, for the **Classical Approach** we use the random forests, adaptive and gradient boosting algorithms, to train the machine learning models in the pipeline from Section 2.1. At the core of the pipeline runs a 5-fold cross-validation hyperparameter tuning procedure; we select the combination of text document NLP preprocessing and machine learning classifier with the best Area-Under-Curve (AUC) average performance on the five validation sets.

'Bag-of-' models: text preprocessing. For bag-of-words models (both 1-gram and 2-gram) and all machine learning algorithms, in the cross-validation routine the NLP preprocessing with

TfidfVectorizer() considers 1) stopword removal with stop_words:[stopwords,None], 2) token removal with max_df:[1.0,0.1,0.3,0.5] and max_features:[None,1000].
The default cases stop_words:None, max_df:1.0 and max_features:None are considered to remove no stopword or token based on document frequency or maximum feature frequency. We implement token filtering using max_df and max_features as the number of tokens retrieved by TfidfVectorizer() from the 39'666 training data is equal to 94'216 for 1-gram models, and to 2'115'592 for 2-gram ones[61]. In case of bag-of-POS models, no stopword removal is needed. There are only 33 POS-tags: neither document frequency nor maximum feature POS-tag selection is performed.

'Bag-of-' models: classifiers. For all machine learning classifiers, in the cross-validation routine we consider the hyperparameter grids shown in Table 2. In the case of adaptive boosting, at each iteration a decision tree is grown using the function DecisionTreeClassifier(max_depth=5); in other words, we consider decision trees with maximum depth equal to 5. After few preliminary runs with the smaller grid n_estimators:[100,200,300,400], we decided to increase the number of estimators for extreme gradient boosting to the values shown in Table 2, due to good scaling of run time.

Classifier	Hyperparameter grid
Random Forests (RF)	n_estimators:[100,200,300,400], max_depth:[1,5,10].
Adaptive boosting (ADA)	n_estimators:[100,200,300,400], learning_rate:[0.001,0.01,0.1,1.0].
Gradient Boosting (XGB)	n_estimators:[100,300,500,1000], learning_rate:[0.001,0.01,0.1,1.0], max_depth:[1,10,20].

Table 2: **Classical Approach**: summary of hyperparameter grids considered in the cross-validation routines, for 'bag-of-' models and all machine learning classifiers.

6.4.3 Classical Approach: machine learning results

In Tables 3 and 4 we collect the results from the **Classical Approach**, using all machine learning classifiers, on both bag-of-words and bag-of-POS models.
Bag-of-words models. Table 3 shows that 2-gram models outperform 1-gram models, for all three classes of machine learning algorithms. The extreme gradient boosting algorithm consistently outperforms both random forests and adaptive boosting; the 2-gram bag-of-words extreme gradient boosting model reaches an out-of-sample AUC=0.955 with moderate learning rate and 500 estimators (or boosting rounds). The tree depth of the model is higher than the XGBoostClassifier() default, i.e. max_depth==3. Text is preprocessed by a 30% maximum document frequency threshold, with no stopword or word by maximum feature removal. The extreme gradient boosting algorithm scales well: the performance improvement of the 2-gram model compared to a 1-gram model is small but the increase in run time is equal to 52 minutes,

[61]We implement 2-gram models specifying ngram_range:(1,2) in TfidfVectorizer(). In other words, we retrieve single tokens *and* their 2-grams.

Model	Best NLP pipeline and classifier	AUC	Accuracy	Run time (hh:mm:ss)
1-gram, RF	`max_depth:10`, `n_estimators:400`, `max_df:0.5`, `max_features:None`, `stop_words:stopwords`.	0.926	0.844	00:17:31
1-gram, ADA	`learning_rate:0.1`, `n_estimators:300`, `max_df:0.5`, `max_features:None`, `stop_words:None`.	0.901	0.830	04:53:24
1-gram, XGB	`max_depth:10`, `learning_rate:0.1`, `n_estimators:1000`, `max_df:1.0`, `max_features:None`, `stop_words:None`.	0.950	0.876	16:54:58
2-gram, RF	`max_depth:10`, `n_estimators:400`, `max_df:0.1`, `max_features:None`, `stop_words:None`.	0.941	0.849	01:06:04
2-gram, ADA	`learning_rate:0.1`, `n_estimators:400`, `max_df:1.0`, `max_features:None`, `stop_words:None`.	0.902	0.829	31:58:55
2-gram, XGB	`max_depth:10`, `learning_rate:0.1`, `n_estimators:500`, `max_df:0.3`, `max_features:None`, `stop_words:None`.	**0.955**	**0.886**	17:47:17

Table 3: **Classical Approach**: summary of NLP bag-of-words and machine learning classifiers. All performance measures are evaluated on out-of-sample data.

only. The best model with the random forests algorithm shows deep trees, i.e. `max_depth:10`, and the maximum number of estimators `n_estimators==400`; the performance increases considerably using 2-grams (from an out-of-sample AUC=0.926 to AUC=0.941); however, the maximum document frequency of words to be used in the bag-of-word model is reduced considerably, from 50% to 10%. Lastly, adaptive boosting algorithms show the weakest performance among all machine learning models. Moreover, adaptive boosting algorithms do not scale: the performance improvement achieved by the 2-gram model is negligible with respect to the one shown by the 1-gram variant, but the increase in run time is equal to 27:05:31 hours.

Bag-of-part-of-speech models. Table 4 shows that POS models deliver much lower performance than bag-of-words ones. For all families of classifiers, the best model reaches an out-of-sample AUC lower than 70%, while the out-of-sample accuracy is slightly above 60%. Also in the case of machine learning models using POS tags as variables, extreme gradient boosting is the algorithm with the highest performance, followed by random forests and adaptive boosting. The use of few POS tags per movie review in a 'bag-of' model is not sufficient to train reliable classifiers; we also note that we implemented only 1-gram models.

6.4.4 Modern approach: machine learning pipeline

For the **Modern Approach**, no NLP preprocessing of movie review data is performed, as shown in the machine learning pipeline from Section 3.2. The hyperparameter grids for all machine

Model	Best classifier (on 1-gram)	AUC	Accuracy	Run time (hh:mm:ss)
RF	`max_depth:10`, `n_estimators:400`.	0.665	0.618	00:04:12
ADA	`learning_rate:0.01`, `n_estimators:400`.	0.662	0.617	00:11:02
XGB	`max_depth:1`, `learning_rate:0.1`, `n_estimators:1000`,	**0.667**	**0.623**	00:19:16

Table 4: **Classical Approach**: summary of NLP bag-of-POS and machine learning classifiers. All performance measures are evaluated on out-of-sample data.

learning algorithms are those shown in Table 2; all machine learning classifiers take as input the 300 dimensions of the text document embedding computed by the `SpaCy` pre-trained model `en_core_web_md`.

6.4.5 Modern approach: machine learning results

In Table 5 we collect results from the **Modern Approach** machine learning runs.

Model	Best classifier	AUC	Accuracy	Run time (hh:mm:ss)
RF	`max_depth:10`, `n_estimators:400`.	0.888	0.808	00:16:30
ADA	`learning_rate:0.1`, `n_estimators:400`.	0.916	0.843	01:51:36
XGB	`max_depth:10`, `learning_rate:0.1`, `n_estimators:1000`.	**0.932**	**0.857**	07:36:25

Table 5: **Modern Approach**: summary of results of machine learning on movie review embeddings. All performance measures are evaluated on out-of-sample data.

Also in the case of word embeddings, extreme gradient boosting delivers best performance, followed by adaptive boosting and, finally, random forests. However, the extreme gradient boosting algorithm with bag-of-words models outperforms the same algorithm run on of word embeddings. As the run time for training machine learning models with embeddings is much shorter than for bag-of-words models (e.g. the longest run time is reached in the case of extreme gradient boosting, and it is about 7.5 hours long), we perform a second round of cross-validation with extended grids, which we show in Table 6. Results are collected in Table 7; both extreme gradient and adaptive boosting algorithms show a moderate performance improvement with respect to the original runs (see Table 5). In the case of random forest classifiers the performance increase is more consistent. Once again, extreme gradient boosting outperforms all classifiers. The performance of extreme gradient boosting algorithms with word embeddings is worse than the one shown in the case of bag-of-words models, for both 1-gram and 2-gram variants. However, run times improve substantially; in fact, the full 5-fold cross-validation pipeline for a 2-gram bag-of-words model and extreme gradient boosting algorithms took 17:47:17 hours to complete; only 07:02:56 hours were needed in the case of word embeddings and the extended grid as in

Table 6.

Classifier	Hyperparameter grid
Random Forests (RF)	`n_estimators:[100,200,300,400,500,600,800,1000]`, `max_depth:[1,5,10,20]`.
Adaptive boosting (ADA)	`n_estimators:[100,200,300,400,500,700,900,1000]`, `learning_rate:[0.001,0.01,0.1,1.0]`.
Gradient Boosting (XGB)	`n_estimators:[100,300,500,1000,1500,2000]`, `learning_rate:[0.001,0.01,0.1,1.0]`, `max_depth:[1,10,20,30]`.

Table 6: **Modern Approach**: extended hyperparameter grids for all machine learning classifiers.

Model	Best classifier	AUC	Accuracy	Run time (hh:mm:ss)
RF	`max_depth:20`, `n_estimators:1000`.	0.896	0.821	00:24:11
ADA	`learning_rate:0.1`, `n_estimators:700`.	0.918	0.846	06:58:55
XGB	`learning_rate:0.1`, `n_estimators:2000`, `max_depth:10`.	**0.934**	**0.860**	07:02:56

Table 7: **Modern Approach**: summary of results of machine learning on movie review embeddings, considered the extended grid shown in Table 6. All performance measures are evaluated on out-of-sample data.

6.4.6 Contemporary approach: RNN architectures

In the case of the **Contemporary Approach**, we want to highlight the advantage that RNNs show over the **Classical** and **Modern Approaches** with regards to the required preprocessing of input text data. Therefore, in the `NLP_IMDb_Case_Study_RNN.ipynb` notebook we start the training of different RNN architectures by considering the import of 'raw' (e.g. not preprocessed) movie reviews. After import, we remove duplicates as described in Section 6.3. We continue with minimal text preprocessing using `punctation` from the `string` module and the `Counter` class from `collections` to surround all punctuation signs in text with whitespaces.Next we compute counts of all unique words in the entire corpus, as shown in Listing 25, which is taken from [46]. We use the word counts and preprocessed data to select the vocabulary size as well as the sequence length T for the training of the RNNs. We see that, while there are 102'966 unique words in our minimally preprocessed corpus, only 15'282 of them appear more than 30 times. Hence, for the word-to-integer transformation we ignore everything but the 15'000 most common words which we bijectively map to the set of integers $\{1, \ldots, 15'000\}$.

```
from string import punctuation
from collections import Counter

counts = Counter()
```

```
for i,review in enumerate(df['review']):
    text = ''.join([c if c not in punctuation else ' '+c+' '
                    for c in review]).lower()
    df.loc[i,'review'] = text
    counts.update(text.split())
```

Code Listing 25: Minimal preprocessing for the **Contemporary Approach**.

Furthermore, we note that the median number of words per (minimally) preprocessed review is 213; therefore, we generate integer sequences of length $T = 200$ by padding with 0's or slicing from the left the sequences originated from preprocessed reviews. The result is a `numpy` array of shape (49'582, 200). Training and test data sets are generated as in described in Section 6.4.1.

We fit and evaluate three different RNN architectures. All three networks include the same embedding layer (first layer), dropout layer (second last) and densely connected layer (last), aimed at processing the input, avoiding overfitting, and generating the output, respectively. However, the network architectures differ in the design of the recurrent layers. The dimension of the embedding layer is chosen as $\tau_0 = 256$ and incurs $(n + 1)\tau_0 =$ 3'840'256 parameters to train, where $n =$ 15'000 denotes the size of the vocabulary and the '+1' is due to the additional possible integer value 0 from padding.

Single layer LSTM. In the first example with a single LSTM layer as recurrent layer, the dimension of the hidden neurons $\boldsymbol{z}_t^{(1)} \in \mathbb{R}^{\tau_1}$ is set to $\tau_1 = 128$. The full architecture for the single layer LSTM is presented in Listing 26.

```
# creating a new model
model = tf.keras.Sequential()

# embedding layer with input_dim = vocabulary size and output_dim = tau_0
model.add(layers.Embedding(input_dim=vocab_size, output_dim=256))

# LSTM layer with tau_1 internal units
model.add(layers.LSTM(128))

# dropout layer to avoid overfitting
model.add(layers.Dropout(0.5))

# dense layer with 1 unit and sigmoid activation
model.add(layers.Dense(1, activation='sigmoid'))
```

Code Listing 26: Keras implementation of the single LSTM layer architecture.

We have $4((\tau_0 + 1)\tau_1 + \tau_1^2 =$ 197'120 parameters from the LSTM layer (see Section 5.2). The dropout layer does not require any parameters, but the output layer adds $(\tau_1 + 1) = 129$ parameters (the output is one dimensional). Hence, the single layer LSTM architecture comprises 4'037'505 trainable parameters, see Listing 27.

```
Model: "single_LSTM"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 256)         3840256
_________________________________________________________________
```

```
lstm (LSTM)                  (None, 128)               197120
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 129
=================================================================
Total params: 4,037,505
Trainable params: 4,037,505
Non-trainable params: 0
```

Code Listing 27: Keras output showing the architecture of the LSTM network defined in Listing 26.

Single layer GRU. For the single GRU layer architecture, we exchange the LSTM layer from Listing 26 with `layers.GRU(128)`, i.e. a GRU layer with $\tau_1 = 128$ neural activations. The number of trainable parameters from this layer is $3((\tau_0 + 1)\tau_1 + \tau_1^2 + \tau_1) =$ 148'224[62] which results in a total of 3'988'609 trainable parameters. The Keras output with the overview of the single layer GRU architecture is given in Listing 28.

```
Model: "single_GRU"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 256)         3840256
_________________________________________________________________
gru (GRU)                    (None, 128)               148224
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 129
=================================================================
Total params: 3,988,609
Trainable params: 3,988,609
Non-trainable params: 0
```

Code Listing 28: Keras output showing the architecture of the single GRU layer network.

Deep LSTM. Finally, we present an example of a deep recurrent architectures with two LSTM layers, see Listing 29 below. The deep LSTM RNN architecture comprises 4'169'089 trainable parameters.

```
Model: "deep_LSTM"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 256)         3840256
_________________________________________________________________
lstm (LSTM)                  (None, None, 128)         197120
_________________________________________________________________
lstm_1 (LSTM)                (None, 128)               131584
_________________________________________________________________
```

[62]We note that the difference between the number of trainable parameters from the Keras implementation of GRUs in Tensorflow 2.0 and the formulae in Section 5.3 is equal to $3\tau_1 = 384$. In fact, by default, Keras adds biases to both the input matrices W_r, W_u, W and the neural activation matrices U_r, U_u, U. This explains the extra term $3\tau_1$. For more details, the interested reader can consult `https://stackoverflow.com/questions/57318930/calculating-the-number-of-parameters-of-a-gru-layer-keras`

```
dropout (Dropout)            (None, 128)              0
_________________________________________________________________
dense (Dense)                (None, 1)                129
=================================================================
Total params: 4,169,089
Trainable params: 4,169,089
Non-trainable params: 0
```

Code Listing 29: Keras output showing the architecture of the network with two LSTM layers.

6.4.7 Contemporary approach: results

In Table 8 we collect the results from the three RNN architectures introduced above. The two layer LSTM architecture shows best performance with an out-of-sample AUC=0.951, followed by the single layer LSTM network, which is only marginally better than the GRU architecture. Overall, the RNNs perform better than the best models from the **Modern Approach** and only slightly worse than the best model from the **Classical Approach**. However, those are the results of extended cross-validation on extensive hyperparameter grids; on the other hand, in the case of the RNN architecture we did not perform any extensive fine-tuning and we used input data with minimal preprocessing. We selected a learning rate equal to 0.001, a batch size of 256 inputs and an `EarlyStopping` callback based on the validation accuracy all architectures. This resulted in training for 7-9 epochs and run times under 30 minutes. Figure 8 shows the development of training and validation accuracy over epochs in the case of the single layer GRU architecture, which trained for 9 epochs.

RNN architecture	AUC	Accuracy	Run time (hh:mm:ss)
Single LSTM	0.947	0.873	00:20:33
Single GRU	0.946	0.874	00:21:18
Deep LSTM	**0.951**	**0.880**	00:28:11

Table 8: Summary of results of RNN classifiers. All performance measures are evaluated on out-of-sample data.

There are many ways in which we could try to further improve the performance of the RNNs we presented. For instance, we could use the weights of a pre-trained word embedding, either fixed or as trainable starting weights in the embedding layer, or we could perform more extensive preprocessing on the input data similar to the **Classical** and **Modern Approach**. Moreover, we could fine-tune the training parameters and save the model from the best training epoch using Keras' `callbacks` functionalities. Finally, we could tweak the architectures of the recurrent layers, as well as of the dropout layers. The interested reader can follow the examples in the community platform Kaggle[63] for further analysis.

7 Conclusions

In this tutorial we have provided the reader with three approaches to preprocess text data with NLP methodologies and subsequently perform text document classification using machine

[63] `https://www.kaggle.com/`

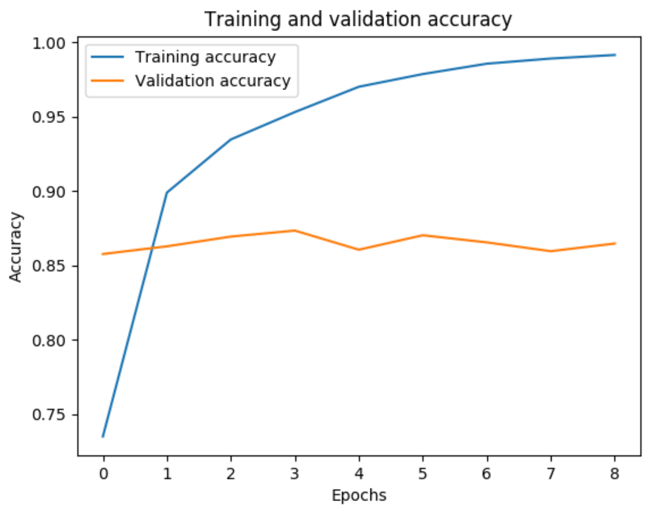


Figure 8: Training and validation accuracy of the single layer GRU architecture over 9 training epochs.

learning. To do so, we have introduced an NLP pipeline to preprocess text data, highlighting the differences which arise in the use of different Python resources. We have continued by describing the bag-of-words approach to the classification of text documents using NLP and machine learning classifiers like random forests, adaptive and extreme gradient boosting. We have introduced a second approach based on word embeddings by discussing prominent examples of embedding models and we have shown how to classify text documents using machine learning models fed on embeddings. Lastly, we have introduced recurrent neural network architectures (LSTM and GRU) and discussed how they can be used for text document classification.
We have applied all three approaches in a case study aimed at the classification of movie reviews from the IMDb. Empirical results show that extreme gradient boosting algorithms trained on bag-of-word models (1-gram and 2-gram variants) outperform all adaptive boosting and random forests on bag-of-words models and word embeddings, and perform marginally better than the one and two layer RNN architectures (2-gram variant). On the other hand, classifiers trained on POS-tags show poor performance. Lastly, we have focused our attention on LSTMs and GRUs architectures, showing that a deep layer LSTM network outperforms the single layer LSTM and GRU architectures. Although classifiers trained on bag-of-words models show better slightly performance, due to the use of extended hyperparameter grids resulting in very long training run times, we recommend to consider RNN architectures for further NLP and machine learning routines on IMDb movie review data, by considering different sets of hyperparameters (including the number of layers). The case study presented in this tutorial was based on textual data in English; NLP preprocessing of text in other languages, for example German, French and Italian, can be performed, for example, with the resources described in the Appendix. We argue that readers with a background in actuarial sciences can use the case study presented in this tutorial as a ‘sand-box’, before moving to case studies of relevance for the insurance industry, for either self-study or to design business relevant proof-of-concepts around NLP and machine learning.

7.1 Acknowledgment

We kindly thank Jürg Schelldorfer (Swiss Re) and Mario Wüthrich (ETH Zurich) for detailed comments that have helped us to improve the manuscript.

8 Appendix

In this Appendix, we collect some material on multilingual NLP resources, with a focus on the German, French and Italian languages.

8.1 Multilingual NLP: some resources

8.1.1 NLP analysis with CoreNLP

The Stanford CoreNLP toolbox[64] allows to perform NLP analysis in a multilingual context; it supports both the German and French languages, among others. With CoreNLP practitioners can perform POS-tagging, named entity recognition, parsing of text, sentiment analysis and information extraction. For Italian, one can access the java-based tool Tint[65], instead. However, we note that Tint uses CoreNLP, as well.

8.1.2 Multilingual pre-trained word embeddings

The Python library `SpaCy` offers pre-trained word embeddings for multiple languages, including German, French and Italian, among others. For example, the core model for "de_core_news_sm"[66] is a "German multi-task CNN trained on the TIGER and WikiNER corpus." The Tiger corpus[67] is curated by the Institute for Natural Language Processing of the University of Stuttgart; it comprises 900K tokens from sentences of German text, taken from the Frankfurter Rundschau[68] newspaper. WikiNER is a a corpus for multilingual named entity relationship from Wikipedia, which is originally discussed in [38]; the interested reader can access a mirror of the wp3 files of the project at `https://github.com/dice-group/FOX/tree/master/input/Wikiner`. Clearly, NLP in German must face linguistic specificities which do not apply to the English language; we refer to, in particular, the declension of articles, adjectival pronouns as well as nouns, the presence of compound words and the use of graphical signs like the 'umlaut' on vowels (ä, ë etc.). Similarly, the models "fr_core_news_sm" and "it_core_news_sm" for French and Italian are pre-trained on ad-hoc corpora, together with WikiNER. We refer to the `SpaCy` documentation for additional details.
Alternatively to `SpaCy`, the Python library `FastText`[69] contains pre-trained word embeddings for 157 languages, including German, French and Italian, trained on Common Crawl and Wikipedia. The word embeddings contain 300 dimensions, by default.

[64]`https://stanfordnlp.github.io/CoreNLP/index.html`
[65]`http://tint.fbk.eu/`
[66]`https://spacy.io/models/de`
[67]`https://www.ims.uni-stuttgart.de/en/research/resources/corpora/tiger/`
[68]`https://www.fr.de/`
[69]`https://fasttext.cc/`

References

[1] Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb), 1137–1155.

[2] Bengio, Y., Simard, P., Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 157-166.

[3] Berry, M.W., Kogan, J. (Eds.). (2010). *Text mining: applications and theory.* John Wiley & Sons, Ltd.

[4] Bird, S., Klein, E., Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit.* O'Reilly Media, Inc.

[5] Bojanowski, P., Grave, E., Joulin, A., Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5, 135-146.

[6] Brown, P.F., deSouza, P.V., Mercer, R.L., Della Pietra, V.J., Lai, J.C. (1992). Class-based n-gram models of natural language. *Computational Linguistics*, 18(4), 467–479.

[7] Chen, T., Guestrin, C. (2016). XGBoost: a scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*, San Francisco, California, USA, 785–794.

[8] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078.*

[9] Chung, J., Gulcehre, C., Cho, K., Bengio, Y. (2015). Gated feedback recurrent neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML '15)*, Lille, France, 2067-2075.

[10] Collobert, R., Weston, J. (2008). A unified architecture for natural language processing: deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*, Helsinki, Finland, 160–167.

[11] Duchi, J., Hazan, E., Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul), 2121-2159.

[12] Ferrario, A., Hämmerli, R. (2019). On boosting: theory and applications. *Available at SSRN:* `https://ssrn.com/abstract=3402687`

[13] Ferrario, A., Noll, A., Wuthrich, M.V. (2018). Insights from inside neural networks. *Available at SSRN:* `https://ssrn.com/abstract=3226852`

[14] Firth, J.R. (1957). A synopsis of linguistic theory 1930-1955. *Studies in Linguistic Analysis.*

[15] Gers, F.A., Schmidhuber, J., Cummins, F. (1999). Learning to forget: continual prediction with LSTM. In *Proceedings of the 9th International Conference on Artificial Neural Networks (ICANN '99)*, Edinburgh, UK, 850-855.

[16] Gers, F.A., Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN 2000), Neural Computing: New Challenges and Perspectives for the New Millennium*, Como, Italy, 189-194.

[17] Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep learning.* MIT Press.

[18] Greff, K., Srivastava, R.K., Koutnik, J., Steunebrink, B.R., Schmidhuber, J. (2016). LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10), 2222-2232.

[19] Harris, Z.S. (1954). Distributional structure. *Word*, 10(23), 146–162.

[20] Hastie, T., Tibshirani, R., Friedman, J. (2009). *The elements of statistical learning: data mining, inference, and prediction.* Springer.

[21] Hochreiter, S., Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735-1780.

[22] Houellebecq, M. (2005). *H. P. Lovecraft: against the world, against life.* Believer Magazine.

[23] Kingma, D.P., Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

[24] Kŭcera, H., Francis, W.N. (1967). *Computational analysis of present-day American English.* Dartmouth Publishing Group.

[25] Kuhn, M., Johnson, K. (2013). *Applied predictive modeling.* Springer.

[26] James, G., Witten, D., Hastie, T., Tibshirani, R. (2013). *An introduction to statistical learning.* Springer.

[27] Jurafsky, D., Martin, J.H. (2009). *Speech and language processing.* Pearson Prentice Hall.

[28] LeCun, Y., Bengio, Y., Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444.

[29] Lee, G.Y., Manski, S., Maiti, T. (2020). Actuarial applications of word embedding models. *ASTIN Bulletin*, 50(1), 1-24.

[30] Lovecraft, H.P. (1984). *The Dunwich horror and others.* Selected by A. Derleth, with texts edited by S.T. Joshi, and an introduction by R. Bloch. Arkham House Publishers, Inc.

[31] Maas, A.L., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y., Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (HLT '11)*, Portland, Oregon, USA, 142-150.

[32] Manning, C.D., Schütze, H. (1999). *Foundations of statistical natural language processing.* MIT Press.

[33] Manning, C.D., Raghavan, P., Schütze, H. (2008). *Introduction to information retrieval.* Cambridge University Press.

[34] Marcus, M., Santorini, B., Marcinkiewicz, M.A. (1993). Building a large annotated corpus of English: the Penn Treebank. *University of Pennsylvania Department of Computer and Information Science Technical Report*, No. MS-CIS-93-87.

[35] Mikolov, T., Chen, K., Corrado, G.S., Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781.*

[36] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems (NIPS '13)*, Lake Tahoe, NV, USA, 3111–3119.

[37] Mitchell, T.M. (1997). *Machine learning.* McGraw-Hill, Inc.

[38] Nothman, J., Ringland, N., Radford, W., Murphy, T., Curran, J.R. (2013). Learning multilingual named entity recognition from Wikipedia. Dataset. *Available at* `https://doi.org/10.6084/m9.figshare.5462500.v1`

[39] Nothman, J., Qin, H., Yurchak, R. (2018). Stop word lists in free open-source software packages. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, Melbourne, Australia, 7-12.

[40] Paice, C.D. (1990). Another stemmer. *ACM SIGIR Forum*, 24(3), 56-61.

[41] Pascanu, R., Mikolov, T., Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*, Atlanta, GA, USA, 1310-1318.

[42] Pennington, J., Socher, R., Manning, C.D. (2014). Glove: global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, 1532–1543.

[43] Pereira, F., Tishby, N., Lee, L. (1993). Distributional clustering of English words. In *Proceedings of the 31st Annual Meeting on Association for Computational Linguistics (ACL '93)*, Columbus, OH, USA, 183-190.

[44] Petrov, S., Das, D., McDonald, R. (2011). A universal part-of-speech tagset. *arXiv preprint arXiv:1104.2086.*

[45] Porter, M.F., (1980). An algorithm for suffix stripping. *Program*, 14(3), 130-137.

[46] Raschka, S., Mirjalili, V. (2017). *Python Machine Learning*. Packt Publishing, Ltd.

[47] Richman, R., Wuthrich, M.V. (2019). Lee and Carter go machine learning: recurrent neural networks. *Available at SSRN:* `https://ssrn.com/abstract=3441030`

[48] Rong, X. (2016). word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*

[49] Rumelhart, D.E., Hinton, G.E., Williams, R.J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536.

[50] Salle, A., Villavicencio, A. (2019). Why so down? The role of negative (and positive) pointwise mutual information in distributional semantics. *arXiv preprint arXiv:1908.06941*

[51] Shalev-Shwartz, S., Ben-David, S. (2014). *Understanding machine learning: from theory to algorithms*. Cambridge University Press.

[52] Williams, R.J., Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin, D.E. Rumelhart (Eds.), *Backpropagation: theory, architectures, and applications*, 433-486, Laurence Erlbaum Associates, Inc.