Convolutional Neural Networks for Text Classification

1. Introduction

In this laboratory we are going to explore the problem of text classification, from a deep learning perspective. This problem has many **applications** such as:

- Sentiment analysis for reviews, comments, posts and other on-line utterings.
- Finding toxic comments in social media, insincere questions on question answering platforms such as Quora or fake reviews on websites.
- Determine if a text advert will be clicked or not.

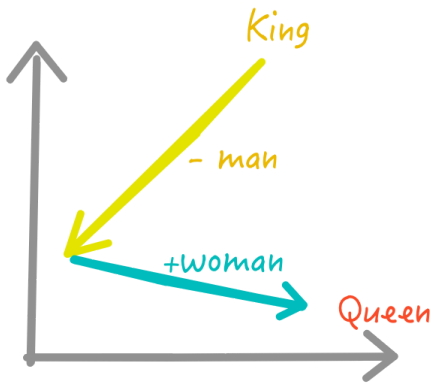
Unlike what happens with images, the pre-processing in text data is critical. Next we are going to present a few preprocessing techniques for text.

2. Data Processing

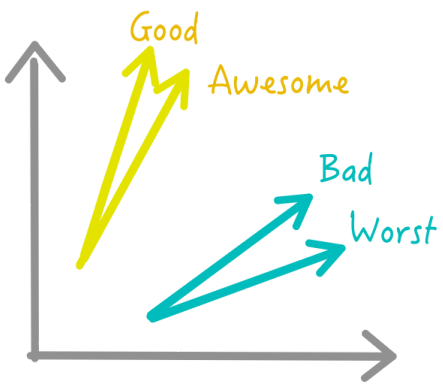
2.1. **Word2vec embeddings**

A way of representing the vocabulary is necessary. Using **one-hot encoding** might come to mind here. The problem with this is that it cannot accurately express the similarity between different words, such as the cosine similarity:

$$\frac{\boldsymbol{x}^\top \boldsymbol{y}}{|\boldsymbol{x}||\boldsymbol{y}|} \in [-1, 1].$$



a) Learns Analogy



b) Similar Words have same angles

Thus, in general we are using **word2vec** - a fixed length vector representation of words capturing the similarity and analogy relationships between different words. See in the above image an example of how this works.

In general, **pre-trained word vectors** are used, after being trained on large corpora of text such as Wikipedia, Twitter, etc. From these the most commonly used are **Glove** and **FastText** with 300 dimensional word vectors.

The word2vec embeddings used here are going to be GLoVE - vectors pre-trained on the Wikipedia corpus. The data structure holding this is a dictionary in which the key is the word and the value is the word vector (an np.array of length 300). This dictionary has around 1 billion entries.

Note that with GLoVE the authors have not converted the words to lower-case. This is why some variations of (lower/upper-case letters in some words) might not be captured (e.g. there might be the word *Good* but not *good*).

2.2. Text preprocessing methods for deep learning

2.2.1. Clean special characters and remove punctuation - the preprocessing has to match the preprocessing done before training the word embedding used. An example of how this can be done in Python can be found below:

```
def clean_text(x):
    pattern = r'[^a-zA-z0-9\s]'
    text = re.sub(pattern, '', x)
    return x
```

2.2.2. Remove the numbers as most embeddings have been preprocessed like this. E.g.:

```
def clean_numbers(x):
    if bool(re.search(r'\d', x)):
        x = re.sub('[0-9]{5,}', '#####', x)
        x = re.sub('[0-9]{4}', '####', x)
        x = re.sub('[0-9]{3}', '###', x)
        x = re.sub('[0-9]{2}', '##', x)
    return x
```

2.2.3. Correct misspells - helps with getting better embedding coverage as the misspelled data is not present in the word2vec.

2.2.4. Expand contractions - the words written with an apostrophe.

2.3. Text preprocessing methods used in conventional Machine Learning

Besides what we have enumerated above, with traditional Machine Learning techniques, we might find ourselves in the situation of doing also the following pre-processings:

2.3.1. Stemming - convert words into their base forms. E.g. *organize, organizing, organizes*.

In order to do this we use heuristic rules (e.g. remove the *s* at the end of any word: *cats -> cat*).

Note that when stemming we might get some nonsense words like *poni* from *ponies*. But it will still work in conventional methods as we are not focusing on the meaning of the words, but rather count them. For deep learning this is one of the main reasons this is not going to work.

```
from nltk.stem import  SnowballStemmer
from nltk.tokenize.toktok import ToktokTokenizer
def stem_text(text):
    tokenizer = ToktokTokenizer()
    stemmer = SnowballStemmer('english')
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    tokens = [stemmer.stem(token) for token in tokens]
    return ' '.join(tokens)
```

2.3.2. Lemmatization - similar to stemming, but the ending is removed only if the base form is present in a dictionary.

```
from nltk.stem import WordNetLemmatizer
from nltk.tokenize.toktok import ToktokTokenizer
def lemma_text(text):
    tokenizer = ToktokTokenizer()
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    tokens = [wordnet_lemmatizer.lemmatize(token) for token in tokens]
    return ' '.join(tokens)
```

3. Representation

3.1. Sequence Creation for Deep Learning

In deep learning applied on text, we do not need to hand-engineer the features. The deep learning algorithm used takes as input a **sequence of text**, does some magic and then learns the structure of that text like a human does.

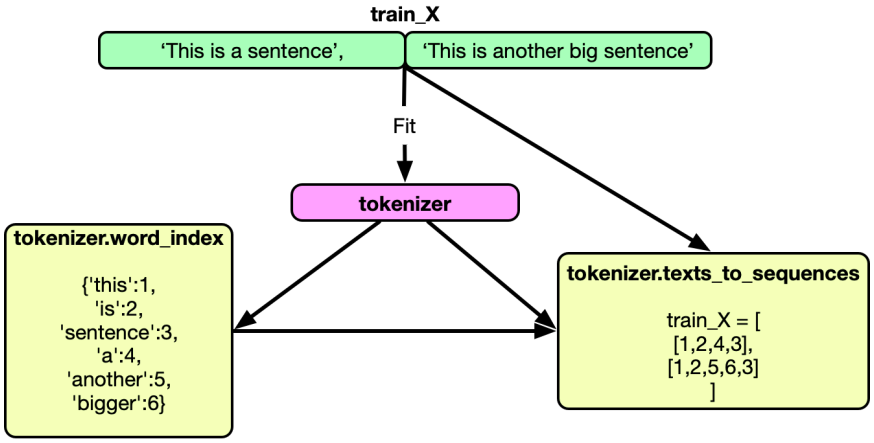
However we should strongly consider the inability of machines to understand characters. They expect their data in a numerical form. Thus we need to represent our data as a series of numbers. For this we can use the **keras tokenizer**. The following two steps might be necessary in different problems:

I. **Tokenizer** - utility function used to split the sequence into words. Parameters:
- num_words - keep a pre-specified number of words in text.
- filter - specify the non-wanted tokens.
- lower - convert the text into lower-case.

The tokenizer once fitted to the data keeps an index of words which can be accessed by tokenizer.word_index. The words in the indexed dictionary are ranked in order of frequencies.

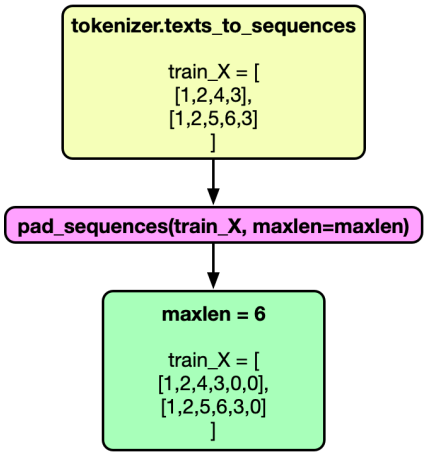
```
Tokenizer(num_words=None, filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n',
lower=True, split=' ', char_level=False, oov_token=None, document_count=0, **kwargs)
```

Below we have an example of what happens with a given sentence after tokenization:



II. **Pad Sequence.** The model (e.g. in CNNs) expects the same number of words/tokens from each training example. E.g.:

```
train_X = pad_sequences(train_X, maxlen=maxlen)
test_X = pad_sequences(test_X, maxlen=maxlen)
```

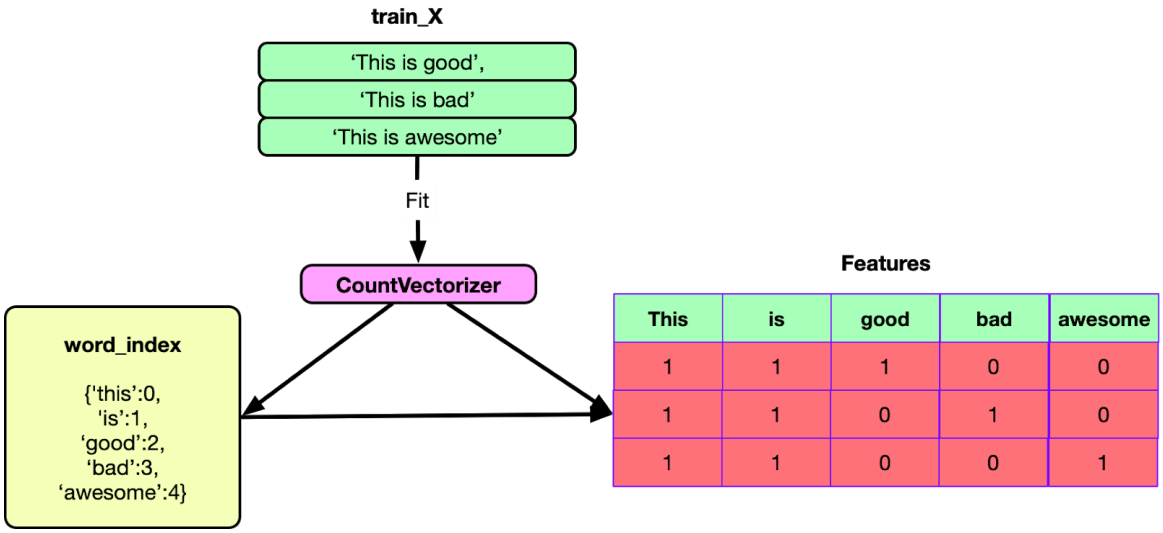


Now the train data is a list of lists of numbers (all of them having the same length).

3.2. Text representation in conventional Machine Learning techniques

In conventional ML methods used for text classification, we need to create features for text. The representations that can achieve this are presented below.

I. **Bag of Words** - creates a dictionary of the most common words in all the sentences.



In order to implement this, we can use the **CountVectorizer** class in Python. The most significant parameters for this class are:

- ngram_range - (1, 3) => 1-grams to 3-grams are going to be considered.
- min_df - min no of times a feature should appear in a corpus to be used.

These features can then be used with any ML classification model (e.g. Logistic Regression, Naive Bayes, SVM, etc). An example of implementation is here.

II. **TFIDF Features** - consider the features only for the significant words. Given a document in a corpus, two metrics are of interest here about the words in that document:

- **Term Frequency** - how important is the word in the document?

$$TF(word\ in\ a\ document) = \frac{No\ of\ occurances\ of\ that\ word\ in\ document}{No\ of\ words\ in\ document}$$

- **Inverse Document Frequency** - how important is the term in the corpus?

$$IDF(word\ in\ a\ corpus) = -log(ratio\ of\ documents\ that\ include\ the\ word)$$

TFIDF is the multiplication of these two scores.

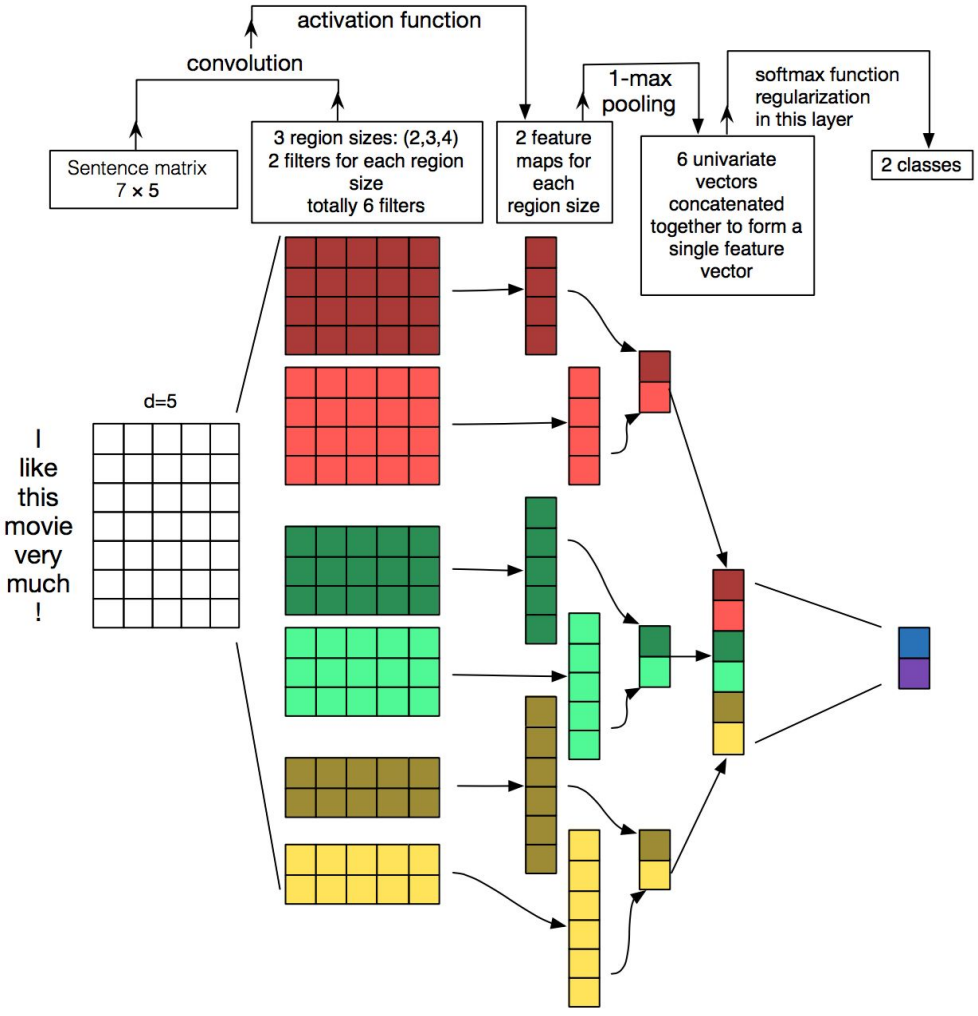
We want to find important words in the document which are not very common (e.g. *a, the, then,* etc). This is why TF is high if the word is very common and IDF is high if the word is rare.

For this types of features, we can use *TFIDFVectorizer* class from Python. Its most significant parameters are the same as the ones for the CountVectorizer.

III. **Hashing Features** - the number of ngrams in a document corpus can get to be pretty big. With this method we can map any n-gram to a number range (e.g. between 1 and 1024). In order to avoid collisions (two different n-grams mapping to the same number, use a range as large as possible).

IV. **Word2Vec features** - see above.

4. CNN for Text Classification



Despite their common applications in Visual tasks, CNNs have been recently applied to various NLP tasks and the results were promising. In order to understand why the CNN approach is also suitable for text classification, we need to see our documents as images.

E.g. for a sentence with maxlen=70 and embedding_size=300, we can create a matrix of numbers 70x300 to represent this sentence. Similar to what happens with images (matrices of pixels).

In this case, each row of the matrix corresponds to one-word vector.
Convolution in text classification: fix the kernel size to **filter_size x embed_size** aka (3, 300) in this case and move it only vertically (we need to consider the whole embedding as opposed to what happens with image pixels).

What needs to be mentioned here is the fact that there are two major approaches when implementing CNNs for text classifications: one working with **words as features** and one working **at character level**. For the scope of this lab we are going to implement and compare the both of them. However, I might alternate the two different types of features in the continuation of this theoretical introduction.

Elements in a CNN for text classification

- **1D convolution filters** - used as n-gram detectors, each of them being specialized in a closely-related family of n-grams. Filters are not homogeneous - a single filter can and often does detect multiple distinctly different families of n-grams.

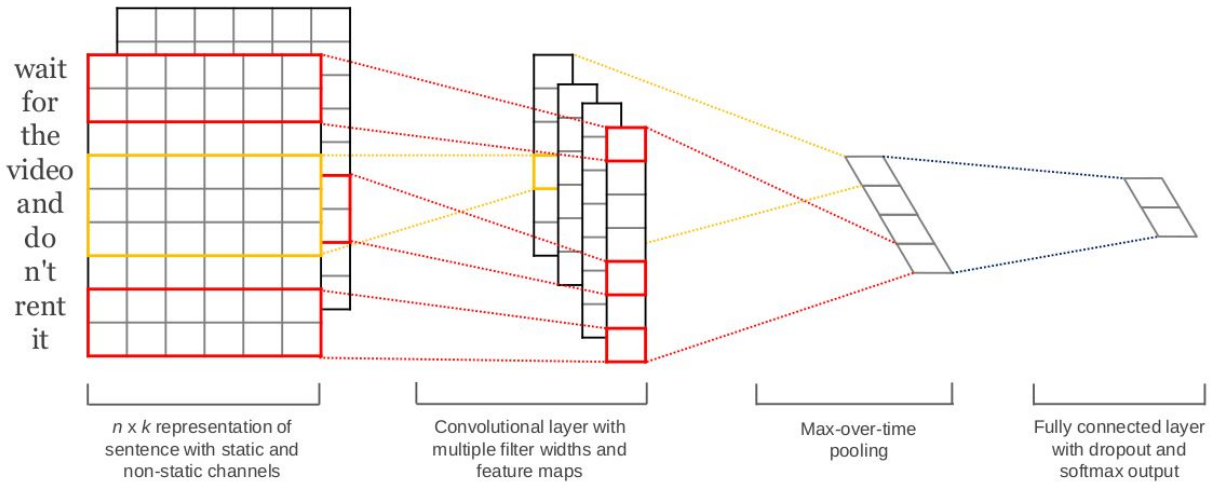
  The result of each convolution will fire when a special pattern is detected. Depending on the kernel size, the patterns detected are going to consist of **2, 3 or 5 adjacent words** (e.g. *"I hate", "very good"*).

- **Max-pooling** - extract the relevant ngrams for making a decision. A **thresholding behavior** is induced here. The values below a given threshold are ignored when they are irrelevant to making a prediction.

Behavior explained

I. Words are represented as **embedding vectors.**
II. Apply ***one convolutional layer*** with ***m filters***. The result is an m-dimensional vector for each document n-gram.
III. Combine the vectors using ***max pooling*** followed by a ***ReLU activation***.
IV. The result is then passed to a ***linear layer*** for the final classification.

Architecture



Given an ***n-word input text*** $w_1, w_2, \ldots, w_n$ we embed each symbol as **d dimensional vector** resulting in the word vectors $\mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_n}$ from R^d. The resulting dxn matrix is then fed into a **convolutional layer** where we pass a sliding window over the text.

For each l-words n-gram:

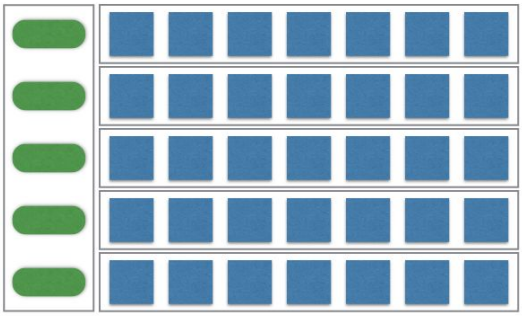
$$\mathbf{u_i} = [\mathbf{w}_i, \ldots, \mathbf{w}_{i+\ell-1}] \in R^{d \times \ell} \; ; \; 0 \leq i \leq n - \ell$$

And for each filter f_j from R^{dxl} we compute $< \mathbf{u_i}, \mathbf{f_j} >$. This convolution gives matrix **F from $R^{m \times n}$.** Apply max-pooling across the ngram dimensions => $p \in R^m$ **.** Feed p into the **ReLU non-linearity**. Finally there is a linear fully connected layer $W \in R^{c \times m}$ - which produces a distribution over the classification classes from which the strongest class is outputted.

$$\mathbf{u}_i = [\mathbf{w}_i; \ldots; \mathbf{w}_{i+\ell-1}]$$
$$F_{ij} = \langle \mathbf{u}_i, \mathbf{f}_j \rangle$$
$$p_j = \mathrm{ReLU}(\max_i F_{ij})$$
$$\mathbf{o} = \mathrm{softmax}(\mathbf{W}\mathbf{p})$$

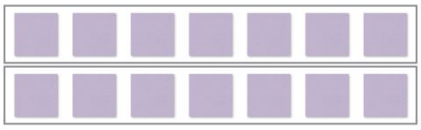
Note that in this example, we only have one convolutional layer. In practice, we use multiple window sizes $l \in L, L \subset N$ by using multiple conv layers in parallel and concatenating the resulting p^l vectors.

Understanding convolutions in text - a more visual approach



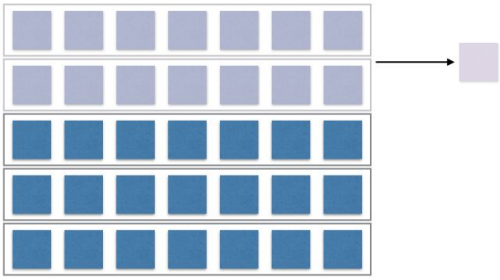
Input

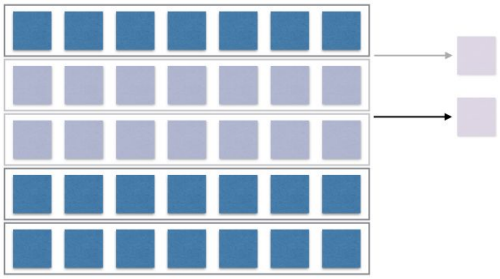
- Green boxes ⇔ words/characters depending on the approach taken.
- Blue boxes ⇔ the representation of the words/characters: around 70 symbols for **characters** (this is going to be a one-hot representation) and for **words**, we will work with dense vectors - pre-trained word embeddings.

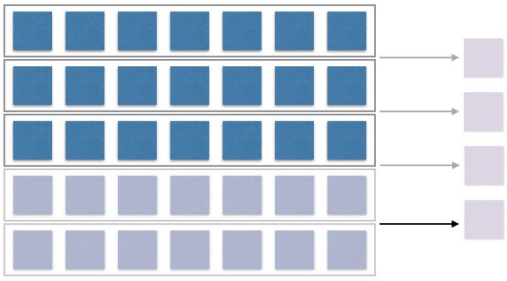


Filters ⇔ Kernels

- Can be of **any length** (the number of rows in the filter).
- **Restriction on the width** - the number of columns has to be the same as the representation of the words/characters.

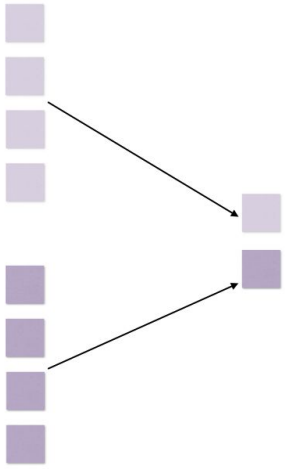






- As in CNNs for images, the filters convolve with the input in order to produce the output.
- Convolve ⇔ multiply with the corresponding cells and sum the results. Actually what is done here is the multiplication of the weights in the filters and the corresponding values in the representation of the words/characters.
- Do not forget about hyperparameters such as **stride** (you already know about this from the classification on images using CNNs). The terms involved in this multiplication depend on the stride.
- The length of the filter is another factor contributing to how the output of the convolution looks in the end.

Final Stage - Max Pooling + the concatenation of multiple feature maps.



Next we are going to use the **imdb dataset** for text classification. A good knowledge of Keras is going to be a plus as you are required to do the implementation using this framework.

As explained at the beginning of this section, you are required to implement both the approach that uses words as features and also the character based approach. In order to do so, follow the indications given in the Jupyter Notebook for this lab.

5. Conclusion

Initially designed to be useful in vision tasks, Convolutional Neural Networks have proved useful also in text classification. The architecture and implementation is similar to what we know from Vision. What really is different here is the interpretation of the operations, the representations and the pre-processings that need to be considered for the input data.

Bibliography

Images taken and explanations inspired from the following sources:

[1]https://mlwhiz.com/blog/2019/01/17/deeplearning_nlp_preprocess/

[2]https://towardsdatascience.com/nlp-learning-series-part-2-conventional-methods-for-text-classification-40f2839dd061

[3]https://towardsdatascience.com/nlp-learning-series-part-3-attention-cnn-and-what-not-for-text-classification-4313930ed566

[4]https://nlp.stanford.edu/projects/glove/

[5]https://debajyotidatta.github.io/nlp/deep/learning/word-embeddings/2016/11/27/Understanding-Convolutions-In-Text/

[6]https://www.kaggle.com/yekenot/2dcnn-textclassifier

[7]https://medium.com/jatana/report-on-text-classification-using-cnn-rnn-han-f0e887214d5f

[8]https://www.aclweb.org/anthology/W18-5408/

[9]https://debajyotidatta.github.io/nlp/deep/learning/word-embeddings/2016/11/27/Understanding-Convolutions-In-Text/

[10]https://richliao.github.io/supervised/classification/2016/11/26/textclassifier-convolutional/

[11]https://arxiv.org/abs/1509.01626