Text Classification: The First Step Toward NLP Mastery

Build a Strong Baseline by Following Simple Steps

Hicham EL BOUKKOURI Follow
Jun 18, 2018 · 9 min read

Natural Language Processing (**NLP**) is a wide area of research where the worlds of artificial intelligence, computer science, and linguistics collide. It includes a bevy of interesting topics with cool real-world applications, like **named entity recognition**, **machine translation** or **machine question answering**. Each of these topics has its own way of dealing with textual data.

But before diving into the deep end and looking at these more complex applications, we need to wade in the shallow end and understand how simpler tasks such as **text classification** are performed.

Text classification offers a good framework for getting familiar with textual data processing without lacking interest, either. In fact, there are many interesting applications for text classification such as **spam detection** and **sentiment analysis**. In this post, we will tackle the latter and show in detail how to build a strong baseline for sentiment analysis classification. This will allow us to get our hands dirty and learn about basic feature extraction methods which are yet very efficient in practice.

So let's begin with a simple question: what is sentiment analysis ?

Word cloud of the sentiment analysis article on Wikipedia

Sentiment analysis aims to estimate the **sentiment polarity** of a body of text based solely on its content. The sentiment polarity of text can be defined as a value that says whether the expressed opinion is **positive** ($polarity=1$), **negative** ($polarity=0$), or neutral. In this tutorial, we will assume that texts are either positive or negative, but that they can't be neutral. Under this assumption, Sentiment Analysis can be expressed as the following classification problem:

```
• Feature: the string representing the input text
• Target: the text's polarity (0 or 1)
```

numerical features.
Unfortunately, we can't even use one-hot encoding as we would do on a categorical feature (such as a `color` feature with values `red`, `green`, `blue`, etc.) because the texts aren't categories, and there is probably no text that is exactly the same as another. Using one-hot encoding in this case would simply result in learning "by heart" the sentiment polarity of each text in the training dataset. So how can we proceed?
At first glance, solving this problem may seem difficult — but actually, very simple methods can go a long way.
We need to transform the main feature — i.e., a succession of words, spaces, punctuation and sometimes other things like emojis — into some numerical features that can be used in a learning algorithm. To achieve this, we will follow two basic steps:

- A **pre-processing** step to make the texts cleaner and easier to process
- And a **vectorization** step to transform these texts into numerical vectors.

Let's dive in !

Pre-Processing

A simple approach is to assume that the smallest unit of information in a text is the word (as opposed to the character). Therefore, we will be representing our texts as **word sequences**. For instance:

```
Text: This is a cat.  -->  Word Sequence: [this, is, a, cat]
```

In this example, we removed the punctuation and made each word lowercase because we assume that punctuation and letter case don't influence the meaning of words. In fact, we want to avoid making distinctions between similar words such as `This` and `this` or `cat.` and `cat`.
Moreover, real life text is often "dirty." Because this text is usually automatically scraped from the web, some HTML code can get mixed up with the actual text. So we also need to tidy up these texts a little bit to avoid having HTML code words in our word sequences. For example :

```
<div>This is not a sentence.<\div> --> [this, is, not, a, sentence]
```

Making these changes to our text before turning them into word sequences is called **pre-processing**. Despite being very simple, the pre-processing techniques we have seen so far work very well in practice. Depending on the kind of texts you may encounter, it may be relevant to include more complex pre-processing steps. But keep in mind that the more steps you add, the longer the pre-processing will take.

Pessimistic depiction of the pre-processing step

Using `Python 3`, we can write a pre-processing function that takes a block of text and then outputs the cleaned version of that text. But before we do that, let's quickly talk about a very handy thing called **regular expressions**.
A regular expression (or **regex**) is a sequence of characters that represent a search pattern. Each character has a meaning; for example, `.` means `any character that isn't the newline character: '\n'`. These characters are often combined with quantifiers, such as `*`, which means `zero or more`. Combining these two characters, we can make the regex that looks for an expression in the form

```
Input string: <a>bcd>
Difference between greedy and non-greedy search :
greedy: <.*>    -->    <a>bcd>
non-greedy: <.*?>    -->    <a>
```

Regular expressions are very useful for processing strings. For example, the `<.*?>` regex we introduced before can be used to detect and remove HTML tags. But we will also be using other regex such as `\'` to remove the character `'` so that words like `that's` become `thats` instead of two separate words `that` and `s`.
Using `re`, the `Python` library for regular expressions, we write our pre-processing function:

```
In [1]: import re


        def clean_text(text):
            """
            Applies some pre-processing on the given text.

            Steps :
            - Removing HTML tags
            - Removing punctuation
            - Lowering text
            """

            # remove HTML tags
            text = re.sub(r'<.*?>', '', text)

            # remove the characters [\], ['] and ["]
            text = re.sub(r"\\", "", text)
            text = re.sub(r"\'", "", text)
            text = re.sub(r"\"", "", text)

            # convert text to lowercase
            text = text.strip().lower()

            # replace punctuation characters with spaces
            filters='!"\'#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n'
            translate_dict = dict((c, " ") for c in filters)
```

preprocessing.ipynb hosted with ❤ by GitHub view raw

Pre-processing function with an example of a clean word sequence

Vectorization

Now that we have a way to extract information from text in the form of word sequences, we need a way to transform these word sequences into numerical features: this is **vectorization**.
The simplest text vectorization technique is Bag Of Words (BOW). It starts with a list of words called the vocabulary (this is often all the words that occur in the training data). Then, given an input text, it outputs a numerical vector which is simply the vector of word counts for each word of the vocabulary. For example :

```
Training texts: ["This is a good cat", "This is a bad day"]
=> vocabulary: [this, cat, day, is, good, a, bad]
New text: "This day is a good day"    -->    [1, 0, 2, 1, 1, 1, 0]
```

As we can see, the values for “cat” and “bad” are **0** because these words don’t appear in the original text.
Using BOW is making the assumption that the more a word appears in a text, the more it is representative of its meaning. Therefore, we assume that given a set of positive and negative text, a good classifier will be able to detect patterns in word distributions and learn to predict the sentiment of a text based on which words occur and how many times they do.
To use BOW vectorization in `Python`, we can rely on `CountVectorizer` from the `scikit-learn` library. In addition to performing vectorization, it will also allow us to remove stop words (i.e., very

Moreover, we can pass our custom pre-processing function fromearlier to automatically clean the text before it's vectorized.

```
In [1]: from sklearn.feature_extraction.text import CountVectorizer


        training_texts = [
            "This is a good cat",
            "This is a bad day"
        ]

        test_texts = [
            "This day is a good day"
        ]

        # this vectorizer will skip stop words
        vectorizer = CountVectorizer(
            stop_words="english",
            preprocessor=clean_text
        )

        # fit the vectorizer on the training text
        vectorizer.fit(training_texts)

        # get the vectorizer's vocabulary
        inv_vocab = {v: k for k, v in vectorizer.vocabulary_.items()}
        vocabulary = [inv_vocab[i] for i in range(len(inv_vocab))]

        # vectorization example
        pd.DataFrame(
```

vectorization.ipynb hosted with ❤ by GitHub view raw

Example of text cleaning + vectorization

Use Case : IMDB Movie Reviews

Let's practice! The IMDB movie reviews dataset is a set of 50,000 reviews, half of which are positive and the other half negative. This dataset is widely used in sentiment analysis benchmarks, which makes it a convenient way to evaluate our own performance against existing models.

Getting the Dataset

The dataset is available online and can be either directly downloaded from Stanford's website or obtained by running in a terminal (Linux):

```
wget http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
```

Then, we need to extract the dowloaded files. You can once again either do it manually or by running:

```
tar -zxvf aclImdb_v1.tar.gz
```

We now have a data folder called `aclImdb`. From there, we can use the following function to load the training/test datasets from IMDB:

```
In [1]: import os
        import numpy as np
        import pandas as pd


        def load_train_test_imdb_data(data_dir):
            """Loads the IMDB train/test datasets from a folder path.
            Input:
            data_dir: path to the "aclImdb" folder.

            Returns:
            train/test datasets as pandas dataframes.
            """
        for sentiment in ["neg", "pos"]:
            score = 1 if sentiment == "pos" else 0

            path = os.path.join(data_dir, split, sentiment)
            file_names = os.listdir(path)
            for f_name in file_names:
                with open(os.path.join(path, f_name), "r") as
f:
                    review = f.read()
                    data[split].append([review, score])
```

load_data.ipynb hosted with ❤ by GitHub view raw

Let’s train a sentiment analysis classifier. One thing to keep in mind is that the feature vectors that result from BOW are usually very large (80,000-dimensional vectors in this case). So we need to use simple algorithms that are efficient on a large number of features (e.g., Naive Bayes, linear SVM, or logistic regression). Let’s train a linear SVM classifier for example.

Because the IMDB dataset is balanced, we can evaluate our model using the accuracy score (i.e., the proportion of samples that were correctly classified).

```
In [1]: from sklearn.metrics import accuracy_score
        from sklearn.svm import LinearSVC


        # Transform each text into a vector of word counts
        vectorizer = CountVectorizer(stop_words="english",
                                     preprocessor=clean_text)

        training_features = vectorizer.fit_transform(train_data["text"
        ])
        test_features = vectorizer.transform(test_data["text"])

        # Training
        model = LinearSVC()
        model.fit(training_features, train_data["sentiment"])
        y_pred = model.predict(test_features)

        # Evaluation
        acc = accuracy_score(test_data["sentiment"], y_pred)

        print("Accuracy on the IMDB dataset: {:.2f}".format(acc*100))

        Accuracy on the IMDB dataset: 83.68
```

application.ipynb hosted with ❤ by GitHub view raw

Application to the IMDB Movie Reviews dataset

As you can see, following some very basic steps and using a simple linear model, we were able to reach as high as an 83.68% accuracy on the IMDB dataset. To realize how good this is, a recent state-of-the-art model can get around 95% accuracy. So this isn’t bad at all, but there is still some room for improvement.

. . .

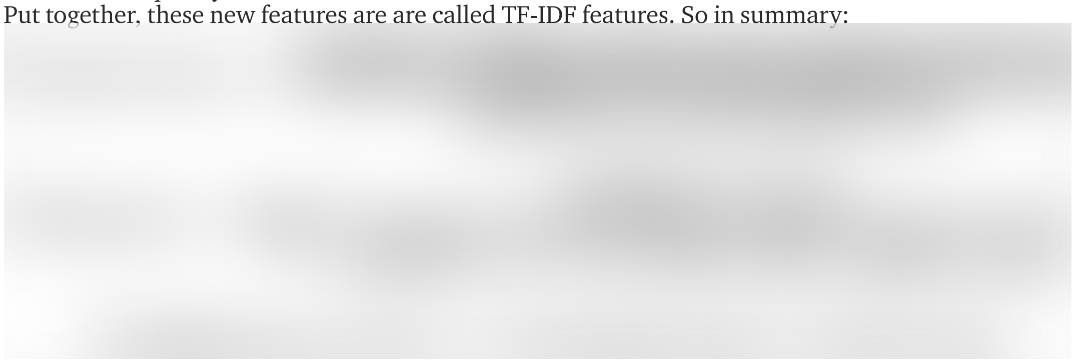
Improving the Current Model

Putting aside anything fine-tuning related, there are some changes we can make to immediately improve the current model.

higher its features (word counts) will be.

To fix this issue, we can use Term Frequency (TF) instead of word counts and divide the number of occurrences by the sequence length. We can also downscale these frequencies so that words that occur all the time (e.g., topic-related or stop words) have lower values. This downscaling factor is called Inverse Document Frequency (IDF) and is equal to the logarithm of the inverse word document frequency.

Put together, these new features are are called TF-IDF features. So in summary:



Formulas for computing TF-IDF features

In practice, we can train a new Linear SVM on TF-IDF features simply by replacing the `CountVectorizer` with a `TfIdfVectorizer`. This results in an accuracy of 86.64%, which is a 2% improvement over using BOW features.

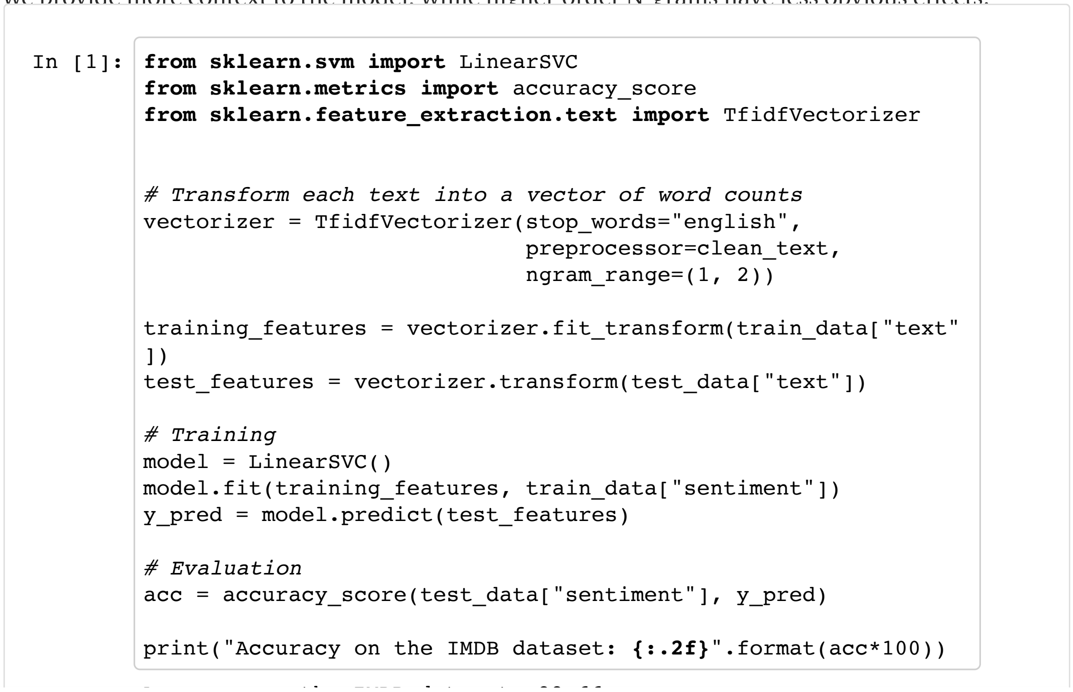
The second thing we can do to further improve our model is to provide it with more context. In fact, considering every word independently can lead to some errors. For instance, if the word `good` occurs in a text, we will naturally tend to say that this text is positive, even if the actual expression that occurs is actually `not good`. These mistakes can be easily avoided with the introduction of N-grams.

An N-gram is a set of N successive words (e.g., `very good` [2-gram] and `not good at all` [4-gram]). Using N-grams, we produce richer word sequences.

For example with N=2:

```
This is a cat. --> [this, is, a, cat, (this, is), (is, a), (a, cat)]
```

In practice, including N-grams in our TF-IDF vectorizer is as simple as providing an additional parameter `ngram_range=(1, N)`. Generally speaking, the use of bi-grams improves performance, as we provide more context to the model, while higher-order N-grams have less obvious effects.



```
In [1]: from sklearn.svm import LinearSVC
        from sklearn.metrics import accuracy_score
        from sklearn.feature_extraction.text import TfidfVectorizer


        # Transform each text into a vector of word counts
        vectorizer = TfidfVectorizer(stop_words="english",
                                     preprocessor=clean_text,
                                     ngram_range=(1, 2))

        training_features = vectorizer.fit_transform(train_data["text"
        ])
        test_features = vectorizer.transform(test_data["text"])

        # Training
        model = LinearSVC()
        model.fit(training_features, train_data["sentiment"])
        y_pred = model.predict(test_features)

        # Evaluation
        acc = accuracy_score(test_data["sentiment"], y_pred)

        print("Accuracy on the IMDB dataset: {:.2f}".format(acc*100))
```

Top highlight

application_TFIDF_bigrams.ipynb hosted with ❤ by GitHub view raw

Code for training a Linear SVM on TF-IDF features with 2-grams

Putting it all together, we achieve an even higher accuracy score of 88.66% which is another 2% improvement over the last version of the model.

. . .

In this post we have seen how to build a strong baseline for text classification following a few simple steps:

- First is the pre-processing step, which is crucial but doesn't need to be too complex. In fact, the only thing we need to do is to remove punctuation and convert everything to lowercase.
- Then comes the vectorization step, which produces numerical features for the classifier. For this we used TF-IDF, a simple vectorization technique that consists in computing word frequencies and downscaling them for words that are too common.
- Finally, for additional context, we provide the model with N-grams, i.e., N-tuples of successive words. Applying this method on the IMDB Movie Reviews dataset, we managed to train a sentiment analysis classifier that scores around 89% (which is only 6% away from the current state-of-the-art).

What Next ?

Features resulting from count-based vectorization methods like TF-IDF have some disadvantages. For instance:

- They don't account for word position and context (despite using N-grams, which is only a quick fix).
- TF-IDF word vectors are usually very high dimensional (>1M features if using bi-grams).
- They are not able to capture semantics.

For this reason, many applications today rely on word embeddings and neural networks, which together can achieve state-of-the-art results. This will be the topic of **the next post** in this series, so make sure not to miss it!

Thanks to Samuel O. Ronsin and Reda Affane.

Machine Learning Classification NLP Naturallanguageprocessing Data Science

986 claps

WRITTEN BY

Hicham EL BOUKKOURI Follow

I'm a PhD student working on #NLP and Word Embeddings at @CNRS (France). Before that, I studied #MachineLearning at @ENSAEParisTech and interned at @dataiku.

data from the trenches Follow

the nitty gritty of data science by the experts @ dataiku

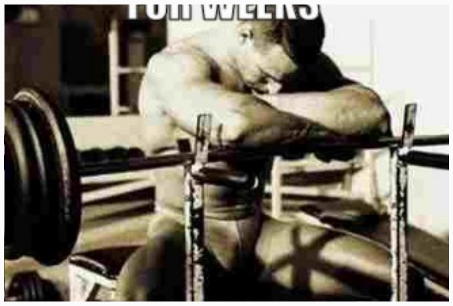
More From Medium

More from data from the trenches

Training Cutting-Edge Neural Networks with Tensor2Tensor and 10 lines of code

Alex Wolf in data...
Oct 17, 2018 · 9 m...
786

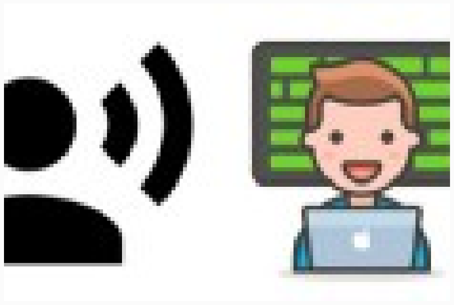
More from data from the trenches



Automatic Feature Engineering: An Event-Driven Approach

Du Phan in data...
Oct 8, 2018 · 7 mi...
456

Related reads



Introduction to Natural Language Processing for Text

Ventsislav...
Nov 16, 2018 · 16...
1.3K