Published in Towards Data Science

Eligijus Bujokas Follow

Mar 14, 2020 · 15 min read · ✨ · Listen

Save

Text Classification Using Word Embeddings and Deep Learning in Python — Classifying Tweets from Twitter

Text classification code along with an in-depth explanation about what is happening under the hood using Python and Tensorflow

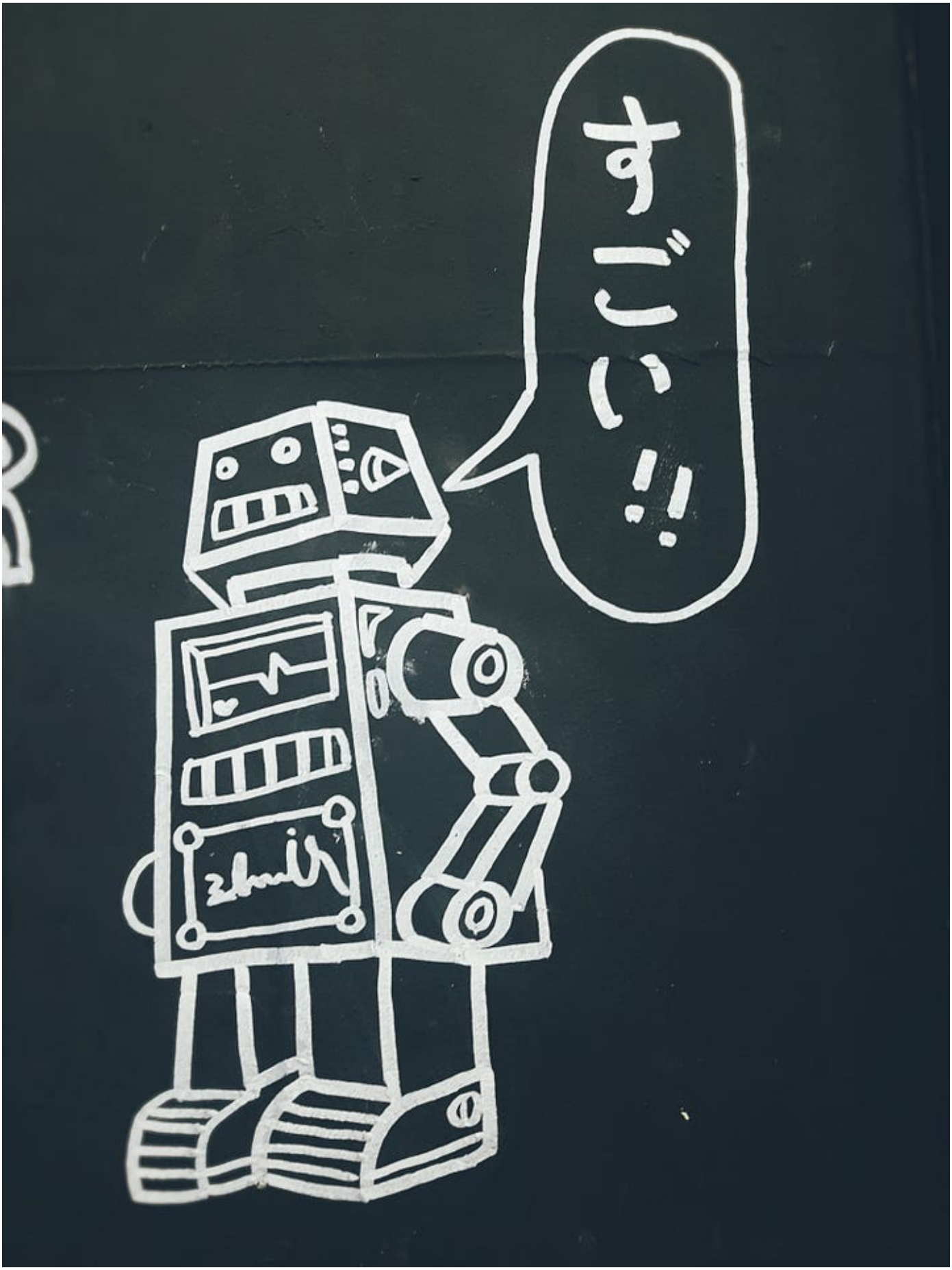


Photo by 수안 최 on Unsplash

The purpose of this article is to help a reader understand how to leverage word embeddings and deep learning when creating a text classifier.

Additionally, the often overlooked parts of text modelling like what are word embeddings, what is the Embedding layer or the input for the deep learning model will be covered here.

Finally, a showcase of all the concepts will be put to practice on a data set published from Twitter about whether a tweet is about a natural disaster or not.

The main technologies used in this article are **Python** and **Keras API.**

A fully functioning text classification pipeline with a dataset[1] from Twitter can be found here: https://github.com/Eligijus112/twitter-genuine-tweets.

Word embeddings file which is used in this article can be found here: https://nlp.stanford.edu/projects/glove/.

The pipeline for creating a deep learning model using labelled texts is as follows:

- **Split the data into text (X) and labels (Y)**
- **Preprocess X**
- **Create a word embedding matrix from X**
- **Create a tensor input from X**
- **Train a deep learning model using the tensor inputs and labels (Y)**
- **Make predictions on new data**

In this article, I will go through each of these steps. The first part of the article will work with a small example data set to cover all the concepts. The second part of the article will implement all the concepts into a real-life example regarding whether a tweet is about a natural disaster or not.

The main building blocks of a deep learning model that uses text to make predictions are word embeddings.

From wiki: **Word embedding** is the collective name for a set of language modelling and feature learning techniques in natural language processing (NLP) where **words or phrases from the vocabulary are mapped to vectors of real numbers.** For example,

“dad” = [0.1548, 0.4848, 1.864]

“mom” = [0.8785, 0.8974, 2.794]

In short, **word embeddings are numerical vectors representing strings.**

In practice, the word representations are either **100, 200 or 300-dimensional vectors and they are trained on very large texts.**

One very important feature of word embeddings is that similar words in a semantic sense have a smaller distance (either Euclidean, cosine or other) between them than words that have no semantic relationship. For example, words like “mom” and “dad” should be closer mathematically than the words “mom” and “ketchup” or “dad” and “butter”.

The second important feature of word embeddings is that, when creating input matrices for models, no matter how many unique words we have in the text corpus, we will have the same number of columns in the input matrices. This is a huge win when compared to the one-hot encoding technique where the number of columns is usually equal to the number of unique words in a document. This number can be hundreds of thousands or even millions. Dealing with very wide input matrices is computationally very demanding.

For example,

Imagine a sentence: **Clark likes to walk in the park.**

There are 7 unique words here. Using one hot encoded vector, we would represent each word by:

```
Clark = [1, 0, 0, 0, 0, 0, 0]
likes = [0, 1, 0, 0, 0, 0, 0]
to = [0, 0, 1, 0, 0, 0, 0]
walk = [0, 0, 0, 1, 0, 0, 0]
in = [0, 0, 0, 0, 1, 0, 0]
the = [0, 0, 0, 0, 0, 1, 0]
park = [0, 0, 0, 0, 0, 0, 1]
```

Whereas if using 2-dimensional word embeddings we would deal with vectors of:

```
Clark = [0.13, 0.61]
likes = [0.23, 0.66]
to = [0.55, 0.11]
walk = [0.03, 0.01]
in = [0.15, 0.69]
the = [0.99, 0.00]
park = [0.98, 0.12]
```

Now imagine having **n** sentences. The vectors in the one-hot encoded case would grow exponentially while the embedding representation vectors of words would stay the same in size. This is why when working with a lot of texts, word embeddings are used to represent words, sentences or the whole document.

Word embeddings are created using a neural network with one input layer, one hidden layer, and one output layer.

For more about creating word embeddings visit the article:

Creating Word Embeddings: Coding the Word2Vec Algorithm in Python using Deep Learning

When I was writing another article that showcased how to use word embeddings in a text classification objective I…

medium.com

For the computer to determine which text is ‘good’ and which is ‘bad’, we need to label it. There could be any number of classes and the classes themselves could mean a very wide variety of things. Let us construct some text:

```
d = [
  ('This article is awesome', 1),
  ('There are just too much words here', 0),
  ('The math is actually wrong here', 0),
  ('I really enjoy learning new stuff', 1),
  ('I am kinda lazy so I just skim these texts', 0),
  ('Who cares about AI?', 0),
  ('I will surely be a better person after reading this!', 1),
  ('The author is pretty cute :)', 1)
]
```

We have 8 tuples where the first coordinate is the text and the second coordinate is the label. Label 0 means a negative sentiment and label 1 means a positive sentiment. To build a functioning model, we would need a lot of more data (in my practice, a thousand or more labelled data points would start giving good results if there are only two classes and the classes are balanced).

Let us do some classical text preprocessing:

```
import re

def clean_text(
    string: str,
    punctuations=r'''!()-[]{};:'"\,<>./?@#$%^&*_~''',
    stop_words=['the', 'a', 'and', 'is', 'be', 'will']) -> str:
    """
    A method to clean text
    """
    # Cleaning the urls
    string = re.sub(r'https?://\S+|www\.\S+', '', string)

    # Cleaning the html elements
    string = re.sub(r'<.*?>', '', string)

```

```
    # Removing the punctuations
    for x in string.lower():
        if x in punctuations:
            string = string.replace(x, "")

    # Converting the text to lower
    string = string.lower()

    # Removing stop words
    string = ' '.join([word for word in string.split() if word not in stop_words])

    # Cleaning the whitespaces
    string = re.sub(r'\s+', ' ', string).strip()

    return string
```

text_preprocesing_embed hosted with ❤ by GitHub view raw

```
'who cares about ai'
'i will surely be a better person after reading this'
'the author is pretty cute'
```

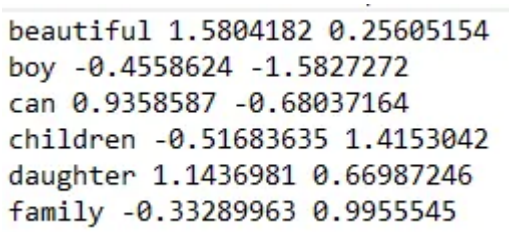
The labels (**Y_train**):

```
[1, 0, 0, 1, 0, 0, 1, 1]
```

Now that we have a preprocessed text in the **X_train** matrix and the class matrix **Y_train, we need to construct the input for the neural network.**

The input of a deep learning model with the Embedding layer uses an **embedding matrix.** The embedding matrix is a matrix of row size equal to the number of unique words in the document and has a column size of the embedding vector dimension. Thus, to construct an embedding matrix, one needs to either create the word embedding vectors or use pre-trained word embeddings. In this example, we will read a fictional word embedding file and construct the matrix.

The usual format in which word embeddings are stored is in a **text document.**



Small embedding example

Let us call the above embedding file **mini_embedding.txt**. For a quick copy-paste use:

```
beautiful 1.5804182 0.25605154
boy -0.4558624 -1.5827272
can 0.9358587 -0.68037164
children -0.51683635 1.4153042
daughter 1.1436981 0.66987246
family -0.33289963 0.9955545
```

In this example, the embedding dimension is equal to 2 but in the word embeddings from the link https://nlp.stanford.edu/projects/glove/, the dimension is 300. In either case, t**he structure is that the word is the first element, followed by the coefficients separated by white spaces. The coordinates end when there is a new line separator at the end of the line.**

To read such text documents let us create a class:

```
import numpy as np


class Embeddings():
    """
    A class to read the word embedding file and to create the word embedding matrix
    """

    def __init__(self, path, vector_dimension):
        self.path = path
        self.vector_dimension = vector_dimension

    @staticmethod
```

```
    def get_coefs(word, *arr):
        return word, np.asarray(arr, dtype='float32')

    def get_embedding_index(self):
        embeddings_index = dict(self.get_coefs(*o.split(" ")) for o in open(self.path, errors='ign
        return embeddings_index

    def create_embedding_matrix(self, tokenizer, max_features):
        """
        A method to create the embedding matrix
        """
        model_embed = self.get_embedding_index()

        embedding_matrix = np.zeros((max_features + 1, self.vector_dimension))
        for word, index in tokenizer.word_index.items():
            if index > max_features:
                break
            else:
                try:
                    embedding_matrix[index] = model_embed[word]
                except:
                    continue
        return embedding_matrix
```

read_embeddings hosted with ❤ by GitHub view raw

```
  vector_dimension=2
)
embedding_matrix = embedding.create_embedding_matrix()
```

We have not scanned any documents yet thus the embedding matrix will return all the words that are in the **mini_embeddings.txt** file:

```
array([[ 1.58041823,  0.25605154],
       [-0.4558624 , -1.58272719],
       [ 0.93585873, -0.68037164],
       [-0.51683635,  1.41530418],
       [ 1.1436981 ,  0.66987246],
       [-0.33289963,  0.99555451]])
```

The embedding matrix will always have the number of columns equal to the number of the embedding dimension and the row count will be equal to the number of unique words in the document or a user-defined number of rows.

Unless you have a vast amount of RAM in your machine, it is generally advised to create the embedding matrix using at the maximum all the unique words of the training document with which you are building the embedding matrix. In the GloVe embedding file, there are millions of words, most of them not even appearing once on most text documents. Thus creating the embedding matrix with all the unique words from the large embeddings file is not advised.

Pretrained word embeddings in a deep learning model are put in a matrix and used in the input layer **as weights**. From the Keras API documentation https://keras.io/layers/embeddings/:

```
keras.layers.Embedding(input_dim, output_dim,...)

Turns positive integers (indexes) into dense vectors of fixed size.
eg. [[4], [20]] -> [[0.25, 0.1], [0.6, -0.2]]

This layer can only be used as the first layer in a deep learning
model.
```

The main two input arguments are the **input_dim** and the **output_dim**.

The **input_dim** is equal to the total number of unique words in our text (or a certain number of unique words which a user defines).

The **output_dim** is equal to the embedding vector dimensions.

To construct the unique word dictionary we will use the **Tokenizer()** method from the Keras library.

```
from keras.preprocessing.text import Tokenizer
```

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)
```

As a reminder, our preprocessed **X_train** is:

```
'this article is awesome'
'there are just too much words here'
'the math is actually wrong here'
'i really enjoy learning new stuff'
'i am kinda lazy so i just skim these texts'
'who cares about ai'
'i will surely be a better person after reading this'
'the author is pretty cute'
```

The Tokenizer() method creates an internal dictionary of unique words and assigns an integer to every word. The output of **tokenizer.word_index**:

```
{'i': 1,
 'is': 2,
 'this': 3,
 'just': 4,
 'here': 5,
 'the': 6,
 'article': 7,
 'awesome': 8,
 'there': 9,
 'are': 10,
 'too': 11,
 'much': 12,
 'words': 13,
 'math': 14,
 'actually': 15,
 'wrong': 16,
 'really': 17,
 'enjoy': 18,
 'learning': 19,
 'new': 20,
 'stuff': 21,
 'am': 22,
 'kinda': 23,
 'lazy': 24,
 'so': 25,
 'skim': 26,
 'these': 27,
```

```
 'texts': 28,
 'who': 29,
 'cares': 30,
 'about': 31,
 'ai': 32,
 'will': 33,
 'surely': 34,
 'be': 35,
 'a': 36,
 'better': 37,
 'person': 38,
 'after': 39,
 'reading': 40,
 'author': 41,
 'pretty': 42,
 'cute': 43}
```

There are 43 unique words in our X_train texts. Lets us convert the texts into indexed lists:

```
tokenizer.texts_to_sequences(X_train)
```

```
[[3, 7, 2, 8],
 [9, 10, 4, 11, 12, 13, 5],
 [6, 14, 2, 15, 16, 5],
 [1, 17, 18, 19, 20, 21],
 [1, 22, 23, 24, 25, 1, 4, 26, 27, 28],
 [29, 30, 31, 32],
 [1, 33, 34, 35, 36, 37, 38, 39, 40, 3],
 [6, 41, 2, 42, 43]]
```

The first sentence in our X_train matrix '**this article is awesome**' is converted into a list of [3, 7, 2, 8]. These indexes represent the key values in the tokenizer created dictionary:

```
{...
 'is': 2,
 'this': 3,
 ...
 'article': 7,
 'awesome': 8,
 ...}
```

The **text_to_sequence()** method gives us a list of lists where each item has different dimensions and is not structured. Any machine learning model needs to know the number of feature dimensions and that number must be the same both for training and predictions on new observations. To convert the sequences into a well-structured matrix for deep learning training we will use the **pad_sequances()** method from Keras:

```
import numpy as np
from keras.preprocessing.sequence import pad_sequences

# Getting the biggest sentence
max_len = np.max([len(text.split()) for text in X_train])

# Creating the padded matrices
X_train_NN = tokenizer.texts_to_sequences(X_train)
X_train_NN = pad_sequences(string_list, maxlen=max_len)
```

The X_train_NN object looks like this:

```
array([[ 0,  0,  0,  0,  0,  0,  3,  7,  2,  8],
       [ 0,  0,  0,  9, 10,  4, 11, 12, 13,  5],
       [ 0,  0,  0,  0,  6, 14,  2, 15, 16,  5],
       [ 0,  0,  0,  0,  1, 17, 18, 19, 20, 21],
       [ 1, 22, 23, 24, 25,  1,  4, 26, 27, 28],
       [ 0,  0,  0,  0,  0,  0, 29, 30, 31, 32],
       [ 1, 33, 34, 35, 36, 37, 38, 39, 40,  3],
       [ 0,  0,  0,  0,  0,  6, 41,  2, 42, 43]])
```

The number of rows is equal to the number of X_train elements and the number of columns is equal to the longest sentence (which is equal to **10** words). The number of columns is usually defined by the user before even reading the document. This is because when working with real-life labelled texts the longest texts can be very long (thousands of words) and this would lead to issues with computer memory when training the neural network.

To create a tidy input for the neural network using preprocessed text I use my defined class **TextToTensor:**

```
1   mport numpy as np
2   from keras.preprocessing.sequence import pad_sequences
```

```
from keras.preprocessing.sequence import pad_sequences


class TextToTensor():

    def __init__(self, tokenizer, max_len):
        self.tokenizer = tokenizer
        self.max_len = max_len

    def string_to_tensor(self, string_list: list) -> list:
        """
        A method to convert a string list to a tensor for a deep learning model
        """
        string_list = self.tokenizer.texts_to_sequences(string_list)
        string_list = pad_sequences(string_list, maxlen=self.max_len)

        return string_list
```

text_to_tensor hosted with ❤ by GitHub view raw

```
# Converting to tensor
TextToTensor_instance = TextToTensor(
tokenizer=tokenizer,
max_len=max_len
)

X_train_NN = TextToTensor_instance.string_to_tensor(X_train)
```

Now that we can create a tensor from the texts we can start using the **Embedding** layer from the Keras API.

```
from keras.models import Sequential
from keras.layers import Embedding

model = Sequential()
model.add(Embedding(
  input_dim=44,
  output_dim=3,
  input_length=max_len))

model.compile('rmsprop', 'mse')
output_array = model.predict(X_train_NN)[0]
```

Notice that in the Embedding layer, input_dim is equal to 44, but our texts have only 43 unique words. This is because of the Embedding definition in Keras API:

input_dim: int > 0. Size of the vocabulary, i.e. **maximum integer index + 1.**

The output_array looks like this:

```
array([[-0.03353775,  0.01123261,  0.03025569],
       [-0.03353775,  0.01123261,  0.03025569],
       [-0.03353775,  0.01123261,  0.03025569],
       [-0.03353775,  0.01123261,  0.03025569],
       [-0.03353775,  0.01123261,  0.03025569],
       [-0.03353775,  0.01123261,  0.03025569],
       [ 0.04183744, -0.00413301,  0.04792741],
       [-0.00870543, -0.00829206,  0.02079277],
       [ 0.02819189, -0.04957005,  0.03384084],
       [ 0.0394035 , -0.02159669,  0.01720046]], dtype=float32)
```

The input sequence is (the first element of X_train_NN):

```
array([0, 0, 0, 0, 0, 0, 3, 7, 2, 8])
```

The Embedding layer automatically assigns an integer a vector of size output_dim, which in our case is equal to 3. We do not have control of that inner computation and the vectors that are assigned to each of the integer indices do not have the feature that closely related words in a semantic sense have a smaller distance between them than those who have a different semantic sense.

To tackle this issue we will use the pre-trained word embeddings from the Stanford NLP department (https://nlp.stanford.edu/projects/glove/). To create the embedding matrix we will use the previously defined method.

Let us assume that **X_train** is once again the list of preprocessed text.

```
embed_path = 'embeddings\\glove.840B.300d.txt'
embed_dim = 300

# Tokenizing the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)

# Creating the embedding matrix
embedding = Embeddings(embed_path, embed_dim)
embedding_matrix = embedding.create_embedding_matrix(tokenizer,
len(tokenizer.word_counts))
```

While the document **glove.840B.300d.txt** has hundreds of thousands of unique words, the final shape of the embedding matrix is **(44, 300)**. This is because we want to save as much memory as possible and the number of unique words in the whole of our document is equal to **44**. Saving the coordinates of all other words from the text document would be a waste because we would not use them anywhere.

To use the embedding matrix in deep learning models we need to pass that matrix as the **weights** parameter in the Embedding layer.

```
from keras.models import Sequential
from keras.layers import Embedding

# Converting to tensor
TextToTensor_instance = TextToTensor(
tokenizer=tokenizer,
max_len=max_len
)
X_train_NN = TextToTensor_instance.string_to_tensor(X_train)

model = Sequential()
model.add(Embedding(
  input_dim=44,
  output_dim=300,
  input_length=max_len,
  weights=[embedding_matrix]))

model.compile('rmsprop', 'mse')
output_array = model.predict(X_train_NN)[0]
```

The output_array's shape is now **(10, 300)** and the output looks like this:

```
array([[ 0.18733 ,  0.40595 , -0.51174 , ...,  0.16495 ,  0.18757 ,
         0.53874 ],
       [ 0.18733 ,  0.40595 , -0.51174 , ...,  0.16495 ,  0.18757 ,
         0.53874 ],
       [ 0.18733 ,  0.40595 , -0.51174 , ...,  0.16495 ,  0.18757 ,
         0.53874 ],
       ...,
       [-0.34338 ,  0.1677  , -0.1448  , ...,  0.095014, -0.073342,
         0.47798 ],
       [-0.087595,  0.35502 ,  0.063868, ...,  0.03446 , -0.15027 ,
         0.40673 ],
       [ 0.16718 ,  0.30593 , -0.13682 , ..., -0.035268,  0.1281  ,
         0.023683]], dtype=float32)
```

Photo by Sheri Hooley on Unsplash

Up to this point we have covered:

- **What are word embeddings**
- **Creating tensors from text**
- **Creating a word embedding matrix**
- **What is a Keras Embedded layer**
- **How to leverage the Embedding matrix**

Now let us put everything together and deal with a real-life problem determining whether a tweet from Twitter is about a natural disaster or not.

```
# Importing generic python packages
import pandas as pd

# Reading the data
train = pd.read_csv('data/train.csv')[['text', 'target']]
test = pd.read_csv('data/test.csv')

# Creating the input for the pipeline
X_train = train['text'].tolist()
Y_train = train['target'].tolist()

X_test = test['text'].tolist()
```

The shape of train data is **(7613, 2),** meaning, there **are** 7613 tweets to work with. Let us check the distribution of the tweets:

```
train.groupby(['target'], as_index=False).count()
```

	target	text
0	0	4342
1	1	3271

Distribution of tweets

As we can see, the classes, at least for the real-world data case, are balanced.

A sample of “good” tweets:

```
[
'Our Deeds are the Reason of this #earthquake May ALLAH Forgive us
all',
'Forest fire near La Ronge Sask. Canada',
"All residents asked to 'shelter in place' are being notified by
officers. No other evacuation or shelter in place orders are
expected",
'13,000 people receive #wildfires evacuation orders in California ',
'Just got sent this photo from Ruby #Alaska as smoke from #wildfires
pours into a school'
]
```

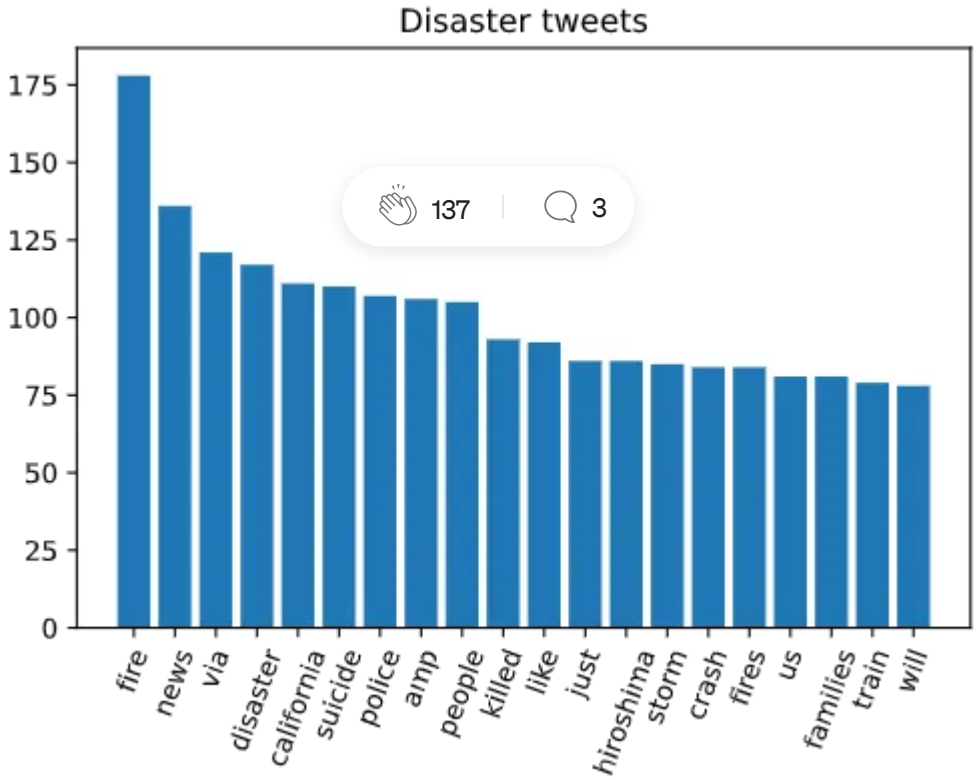
A sample of the bad tweets:

```
[
"What's up man?",
'I love fruits',
'Summer is lovely',
'My car is so fast',
'What a goooooooaaaaaal!!!!!!'
]
```

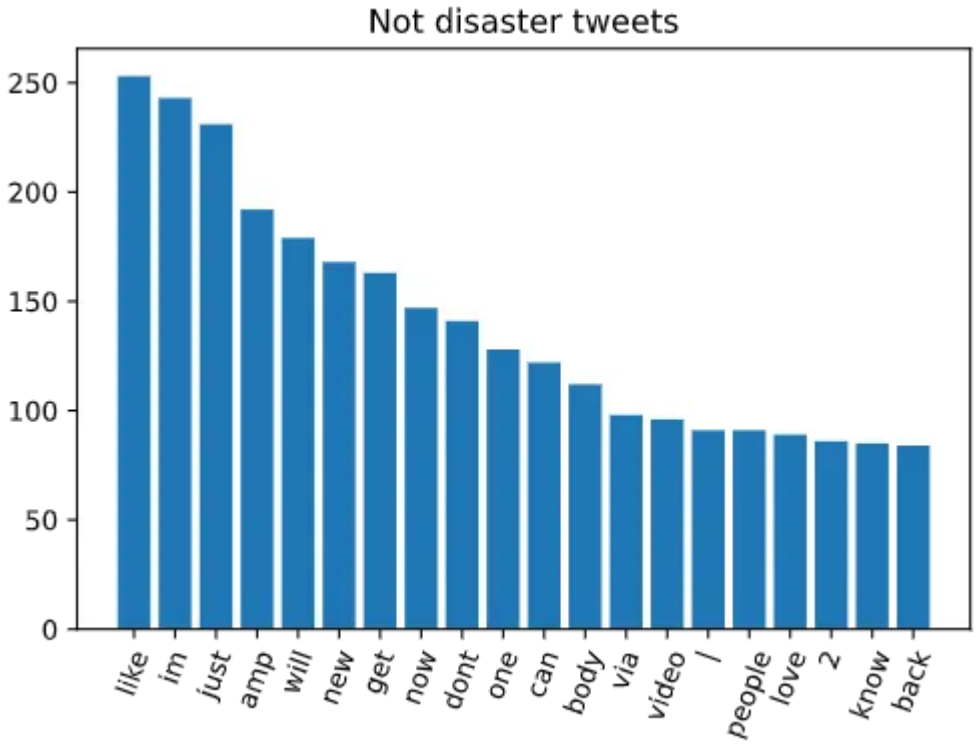


```
# Counting the number of words
from collections import Counter

# Plotting functions
import matplotlib.pyplot as plt

X_train = [clean_text(text) for text in X_train]
Y_train = np.asarray(Y_train)

# Tokenizing the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)

# Getting the most frequent words
```

```
d1 = train.loc[train['target']==1, 'text'].tolist()
d0 = train.loc[train['target']==0, 'text'].tolist()

d1 = [clean_text(x, stop_words=stop_words) for x in d1]
d0 = [clean_text(x, stop_words=stop_words) for x in d0]

d1_text = ' '.join(d1).split()
d0_text = ' '.join(d0).split()

topd1 = Counter(d1_text)
topd0 = Counter(d0_text)

topd1 = topd1.most_common(20)
topd0 = topd0.most_common(20)

plt.bar(range(len(topd1)), [val[1] for val in topd1], align='center')
plt.xticks(range(len(topd1)), [val[0] for val in topd1])
plt.xticks(rotation=70)
plt.title('Disaster tweets')
plt.show()

plt.bar(range(len(topd0)), [val[1] for val in topd0], align='center')
plt.xticks(range(len(topd0)), [val[0] for val in topd0])
plt.xticks(rotation=70)
plt.title('Not disaster tweets')
plt.show()
```



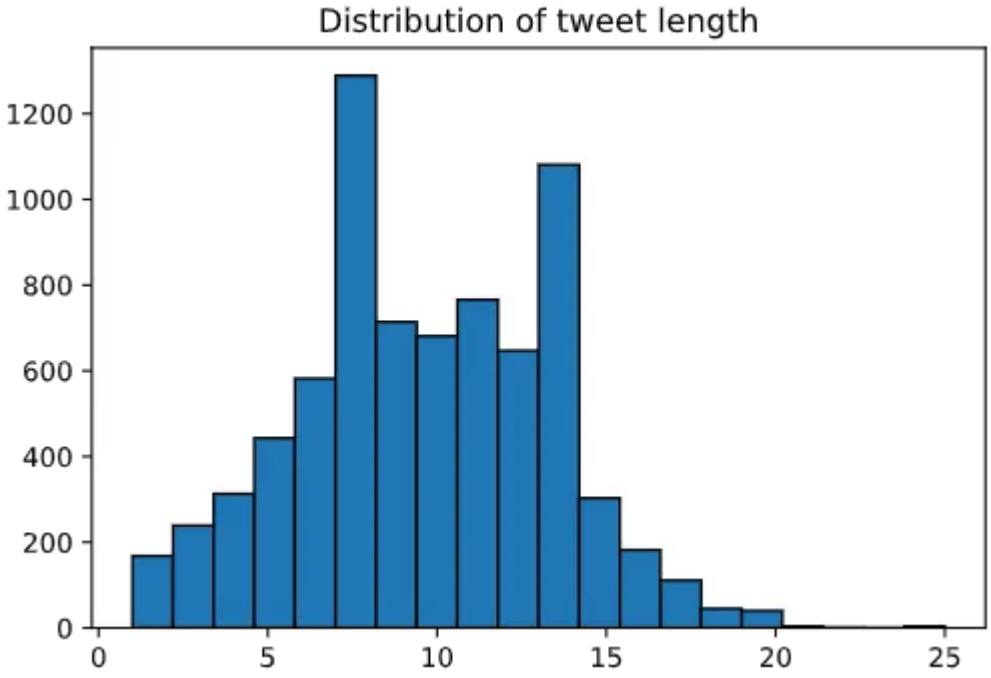
Top words in disaster tweets



Top words in not disaster tweets

The not disaster words are more generic than the disaster ones. One may expect that the GloVe embeddings and the deep learning model will be able to catch these differences.

The distribution of the number of words in each tweet:



Distribution of the number of words

We can say from the distribution above that creating the input tensors with a column size of 20 will only exclude a very little amount of words in tweets. On the pro side, we will win a lot of computational time.

The deep learning model architecture is the following:

The pipeline that wraps everything up which was mentioned in this article is also defined as a class in python:

The whole code and the whole working pipeline can be found here:

Eligijus112/twitter-genuine-tweets

This project can serve as a template to do any NLP task. The sentiment in this project is whether a string regards a...

github.com	

To train the model use the code:

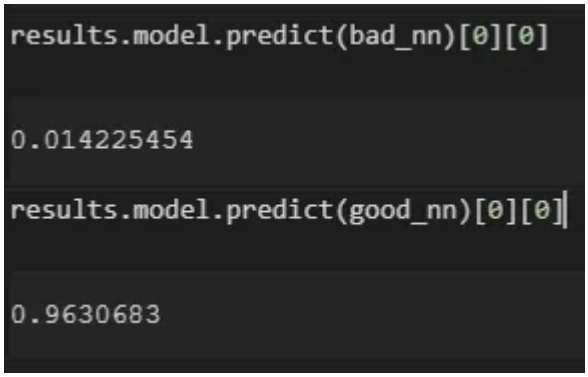
```
results = Pipeline(
X_train=X_train,
Y_train=Y_train,
embed_path='embeddings\\glove.840B.300d.txt',
embed_dim=300,
stop_words=stop_words,
X_test=X_test,
max_len=20,
epochs=10,
batch_size=256
)
```

Now let us create two texts:

good = [“Fire in Vilnius! Where is the fire brigade??? #emergency”]

bad = [“Sushi or pizza? Life is hard :((”]

```
TextToTensor_instance = TextToTensor(
tokenizer=results.tokenizer,
max_len=20
)

# Converting to tensors
good_nn = TextToTensor_instance.string_to_tensor(good)
bad_nn = TextToTensor_instance.string_to_tensor(bad)

# Forecasting
p_good = results.model.predict(good_nn)[0][0]
p_bad = results.model.predict(bad_nn)[0][0]
```



The **p_bad = 0.014** and **p_good = 0.963**. These probabilities are for the question of whether a tweet is about a disaster or not. So the tweet about sushi has a very low score and the tweet about the fire has a big score. This means that the logic that was presented in this article works at least on the made-up sentences.

In this article I have:

- **Presented the logic and overall workflow of using text data in a supervised problem.**
- **Shared a fully working code in Python that takes raw inputs, converts them to matrices and trains a deep-learning model using Tensorflow**
- **Interpreted the results.**

The code presented in this article can be applied to any text classification problem. Happy coding!

[1] Twitter disaster dataset

URL: https://www.kaggle.com/competitions/nlp-getting-started/data License: https://opensource.org/licenses/Apache-2.0

Classification Machine Learning Natural Language Process Python NLP