Sentiment Analysis

Georges Sakr

11/19/2019

Outline

- Word Embedding
- Sentiment Analysis

Word Embedding

- Since most of the statistical algorithms, e.g machine learning and deep learning techniques, work with numeric data, therefore we have to convert text into numbers.
- Several approaches exist in this regard. However, the most famous ones are:
  - Bag of Words
  - TF-IDF
  - word2vec.

Several libraries exist, such as Scikit-Learn and NLTK, which can implement these techniques in one line of code

Bag of Words

- Suppose we have a corpus with three sentences:

```
"I like to play football"
"Did you go outside to play tennis"
"John and I play tennis"
```

- Now if we have to perform text classification, or any other task, on the above data using statistical techniques, we can not do so since statistical techniques work only with numbers. Therefore we need to convert these sentences into numbers.

Tokenization

The first step in this regard is to convert the sentences in our corpus into tokens or individual words. Look at the table below:

Sentence 1	Sentence 2	Sentence 3
I	did	jhon
like	you	and
to	go	I
play	outside	play
football	to	tennis
	play	
	tennis	

Dictionary of word frequency

The next step is to create a dictionary that contains all the words in our corpus as keys and the frequency of the occurrence of the words as values. In other words, we need to create a histogram of the words in our corpus. Look at the following table:

Table

Word	Frequency
I	2
like	1
to	2
play	3
football	1
did	1
you	1
go	1
outside	1
tennis	2
Jhon	1
and	1

Sorting the Table

- In our corpus, we only had three sentences, therefore it is easy for us to create a dictionary that contains all the words.
- In the real world, there will be millions of words in the dictionary. Some of the words will have a very small frequency.
- The words with very small and very high frequency are not very useful, hence such words are removed.
- One way to remove them is to sort the word frequency dictionary in the decreasing order of the frequency and then filter the words.

Table

Word	Frequency
play	3
I	2
to	2
tennis	2
football	1
like	1
did	1
you	1
go	1
outside	1
Jhon	1
and	1

Creating the bag of words

- To create the bag of words model, we need to create a matrix where the columns correspond to the most frequent words in our dictionary where rows correspond to the document or sentences.
- Suppose we filter the 8 most occurring words from our dictionary. Then the document frequency matrix will look like this:

	Play	Tennis	To	I	Football	Did	You	go
Sentence 1	1	0	1	1	1	0	0	0
Sentence 2	1	1	1	0	0	1	1	1
Sentence 3	1	1	0	1	0	0	0	0

How did we fill the table

- It is important to understand how the above matrix is created.
- In the above matrix, the first row corresponds to the first sentence. In the first column, the word "play" occurs once, therefore we added 1 in the first column.
- The word in the second column is "Tennis", it doesn't occur in the first sentence, therefore we added a 0 in the second column for sentence 1.
- Final matrix corresponds to the bag of words model.
- The first row shows the numeric representation of Sentence 1. This numeric representation can now be used as input to the statistical models

scikit-learn

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(allsentences)
print(X.toarray())
```

Frequency

- max_df : float in range [0.0, 1.0] or int, default=1.0 When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words). If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.
- min_df : float in range [0.0, 1.0] or int, default=1 When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.
- max_features : int or None, default=None If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

TF-IDF

Before we actually see the TF-IDF model, let us first discuss a few problems associated with the bag of words model.

"I like to play football" "Did you go outside to play tennis" "John and I play tennis"

The resulting bag of words model looked like this:

	Play	Tennis	To	I	Football	Did	You	go
Sentence 1	1	0	1	1	1	0	0	0
Sentence 2	1	1	1	0	0	1	1	1
Sentence 3	1	1	0	1	0	0	0	0

Problems with BOW

- One of the main problems associated with the bag of words model is that it assigns equal value to the words, irrespective of their importance. For instance, the word "play" appears in all the three sentences, therefore this word is very common, on the other hand, the word "football" only appears in one sentence. The words that are rare have more classifying power compared to the words that are common.
- The idea behind the TF-IDF approach is that the words that are more common in one sentence and less common in other sentences should be given high weights

Tokenization

The first step in this regard is to convert the sentences in our corpus into tokens or individual words. Look at the table below:

Sentence 1	Sentence 2	Sentence 3
I	did	jhon
like	you	and
to	go	I
play	outside	play
football	to	tennis
	play	
	tennis	

Find TF-IDF Values

- TF value refers to term frequency and can be calculated as follows

$$TF = \frac{Frequency\ of\ the\ word\ in\ the\ sentence}{Total\ number\ of\ words\ in\ the\ sentence} \quad (1)$$

- For instance, look at the word "play" in the first sentence. Its term frequency will be 0.20 since the word "play" occurs only once in the sentence and the total number of words in the sentence are 5, hence, $1/5 = 0.20$.

Find TF-IDF Values

- IDF refers to inverse document frequency and can be calculated as follows:

$$IDF = \frac{Total\ number\ of\ sentences}{Number\ of\ sentences\ containing\ the\ word} \tag{2}$$

- It is important to mention that the IDF value for a word remains the same throughout all the documents as it depends upon the total number of documents. On the other hand, TF values of a word differ from document to document. Let's find the IDF frequency of the word "play". Since we have three documents and the word "play" occurs in all three of them, therefore the IDF value of the word "play" is $3/3 = 1$.
- Finally

$$TFIDF = TF \times IDF \tag{3}$$

Example

Word	Frequency	Word	Sorted Frequency	IDF
I	2	play	3	$3/3 = 1$
like	1	tennis	2	$3/2 = 1.5$
to	2	to	2	$3/2 = 1.5$
play	3	I	2	$3/2 = 1.5$
football	1	football	1	$3/1 = 3$
Did	1	Did	1	$3/1 = 3$
you	1	you	1	$3/1 = 3$
go	1	go	1	$3/1 = 3$
outside	1	outside	1	
tennis	2	like	1	
John	1	John	1	
and	1	and	1	

TF-IDF Values

Word	Sentence 1	Sentence 2	Sentence 3
play	0.20 x 1 = 0.20	0.14 x 1 = 0.14	0.20 x 1 = 0.20
tennis	0 x 1.5 = 0	0.14 x 1.5 = 0.21	0.20 x 1.5 = 0.30
to	0.20 x 1.5 = 0.30	0.14 x 1.5 = 0.21	0 x 1.5 = 0
I	0.20 x 1.5 = 0.30	0 x 1.5 = 0	0.20 x 1.5 = 0.30
football	0.20 x 3 = 0.6	0 x 3 = 0	0 x 3 = 0
did	0 x 3 = 0	0.14 x 3 = 0.42	0 x 3 = 0
you	0 x3 = 0	0.14 x 3 = 0.42	0 x 3 = 0
go	0x 3 = 0	0.14 x 3 = 0.42	0 x 3 = 0

Note

- The values in the columns for sentence 1, 2, and 3 are corresponding TF-IDF vectors for each word in the respective sentences.
- The use of the log function with TF-IDF.
- It is important to mention that to mitigate the effect of very rare and very common words on the corpus, the log of the IDF value can be calculated before multiplying it with the TF-IDF value. In such case the formula of IDF becomes:

$$IDF = \log\left(\frac{Total\ number\ of\ sentences}{Number\ of\ sentences\ containing\ the\ word}\right) \tag{4}$$

Python

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=8)
allsentences=["I like to play football",
"Did you go outside to play tennis",
"John and I play tennis"]
X = vectorizer.fit_transform(allsentences)
print(X.toarray())
print(vectorizer.get_feature_names())
```

Problems with TF-IDF and Bag of Words Approach

- In the bag of words and TF-IDF approach, words are treated individually and every single word is converted into its numeric counterpart.
- The context information of the word is not retained.
- Consider two sentences "big red machine and carpet" and "big red carpet and machine". If you use a bag of words approach, you will get the same vectors for these two sentences. However, we can clearly see that in the first sentence we are talking about a "big red machine", while the second sentence contains information about the "big red carpet". Hence, context information is very important. The N-Grams model basically helps us capture the context information.

Theory of N-Grams Model

Wikipedia defines an N-Gram as "A contiguous sequence of N items from a given sample of text or speech". Here an item can be a character, a word or a sentence and N can be any integer. When N is 2, we call the sequence a bigram. Similarly, a sequence of 3 items is called a trigram, and so on.

Example

- Let's see a simple example of character bigrams where each character is a state.
- Football is a very famous game
- The character bigrams for the above sentence will be: fo, oo, ot, tb, ba, al, ll, l, i, is and so on. You can see that bigrams are basically a sequence of two consecutively occurring characters.
- Similarly, the trigrams are a sequence of three contiguous characters, as shown below:
- foo, oot, otb, tba and so on.

N-Grams of words

- In the previous two examples, we saw character bigrams and trigrams. We can also have bigrams and trigrams of words.
- “big red machine and carpet”. The bigram of this sentence will be “big red”, “red machine”, “machine and”, “and carpet”. The bigrams for the sentence “big red carpet and machine” will be “big red”, “red carpet”, “carpet and”, “and machine”.
- Here in this case with bigrams, we get a different vector representation for both of the sentences.

Python

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(ngram_range=(2, 2))
allsentences=["I like to play football",
"Did you go outside to play tennis",
"John and I play tennis"]
X = vectorizer.fit_transform(allsentences)
print(X.toarray())
print(vectorizer.get_feature_names())
```

Word Embedding

- A potential drawback with one-hot encoded feature vector approaches such as N-Grams, bag of words and TF-IDF approach is that the feature vector for each document can be huge.
- If you have a half million unique words in your corpus and you want to represent a sentence that contains 10 words, your feature vector will be a half million dimensional one-hot encoded vector where only 10 indexes will have 1. This is a wastage of space and increases algorithm complexity exponentially resulting in the curse of dimensionality.

Word Embedding - 2

- In word embeddings, every word is represented as an n-dimensional dense vector.
- The words that are similar will have similar vector.
- Word embeddings techniques such as GloVe and Word2Vec have proven to be extremely efficient for converting words into corresponding dense vectors. The vector size is small and none of the indexes in the vector is actually empty.

Implementing Word Embedding with Keras Sequential Models

- To implement word embeddings, the Keras library contains a layer called Embedding(). The embedding layer is normally used as a first layer in the sequential model for NLP tasks.
- The embedding layer can be used to peform three tasks in Keras:
  - learn word embeddings and save the resulting model
  - learn the word embeddings in addition to performing the NLP tasks such as text classification, sentiment analysis, etc.
  - load pretrained word embeddings and use them in a new model

Embedding layer

```
embedding_layer = Embedding(200, 32, input_length=50)
```

- The first parameter in the embedding layer is the size of the vocabulary or the total number of unique words in a corpus.
- The second parameter is the number of the dimensions for each word vector. For instance, if you want each word vector to have 32 dimensions, you will specify 32 as the second parameter. And finally, the third parameter is the length of the input sentence.
- The output of the word embedding is a 2D vector where words are represented in rows, whereas their corresponding dimensions are presented in columns. Finally, if you wish to directly connect your word embedding layer with a densely connected layer, you first have to flatten your 2D word embeddings into 1D.

Custom Word Embedding

We will perform simple text classification tasks that will use word embedding following these steps:

- Add the corpus and label
- Find the total number of words in our corpus
- *one_hot* all sentences
- Make equal length by adding 0 to the empty index
- Create the Model

GloVe

- In the previous section we trained custom word embeddings. However, we can also use pretrained word embeddings.
- Several types of pretrained word embeddings exist, however we will be using the GloVe word embeddings from Stanford NLP since it is the most famous one and commonly used. The word embeddings can be downloaded from this link
- The smallest file is named "Glove.6B.zip". The size of the file is 822 MB. The file contains 50, 100, 200, and 300 dimensional word vectors for 400k words. We will be using the 100 dimensional vector.

Code

- In the last section, we used one_hot function to convert text to vectors. Another approach is to use Tokenizer function from keras.preprocessing.text library.
- You simply have to pass your corpus to the Tokenizer's fit_on_text method.
- We will create a dictionary that will contain words as keys and the corresponding 100 dimensional vectors as values, in the form of an array. Execute the following script:

Code-2

```
for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions = asarray(records[1:], dtype='float32')
    embeddings_dictionary [word] = vector_dimensions
glove_file.close()
```

Summary

To use text data as input to the deep learning model, we need to convert text to numbers. However unlike machine learning models, passing sparse vector of huge sizes can greately affect deep learning models. Therefore, we need to convert our text to small dense vectors. Word embeddings help us convert text to dense vectors.

Deep Learing on Real World Data

- Classification in more detail using a real-world dataset.
- We will use three different types of deep neural networks:
  - Densely connected neural network (Basic Neural Network)
  - Convolutional Neural Network (CNN)
  - Long Short Term Memory Network (LSTM), which is a variant of Recurrent Neural Networks.

Dataset

Download the dataset from here and extract the compressed file, you will see a CSV file.

- The file contains 50,000 records and two columns: review and sentiment.
- The review column contains text for the review and the sentiment column contains sentiment for the review.
- The sentiment column can have two values i.e. “positive” and “negative” which makes our problem a binary classification problem.

Sample Review

```
"Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his pa
rents are fighting all the time.<br /><br />This movie is slower than a soap opera... and suddenly,
Jake decides to become Rambo and kill the zombie.<br /><br />OK, first of all when you're going to m
ake a film you must Decide if its a thriller or a drama! As a drama the movie is watchable. Parents
are divorcing & arguing like in real life. And then we have Jake with his closet which totally ruins
all the film! I expected to see a BOOGEYMAN similar movie, and instead i watched a drama with some m
eaningless thriller spots.<br /><br />3 out of 10 just for the well playing parents & descent dialog
s. As for the shots with Jake: just ignore them."
```

You can see that our text contains punctuations, brackets, and a few HTML tags as well. We will preprocess this text and finally, let's see the distribution of positive and negative sentiments in our dataset.

Count

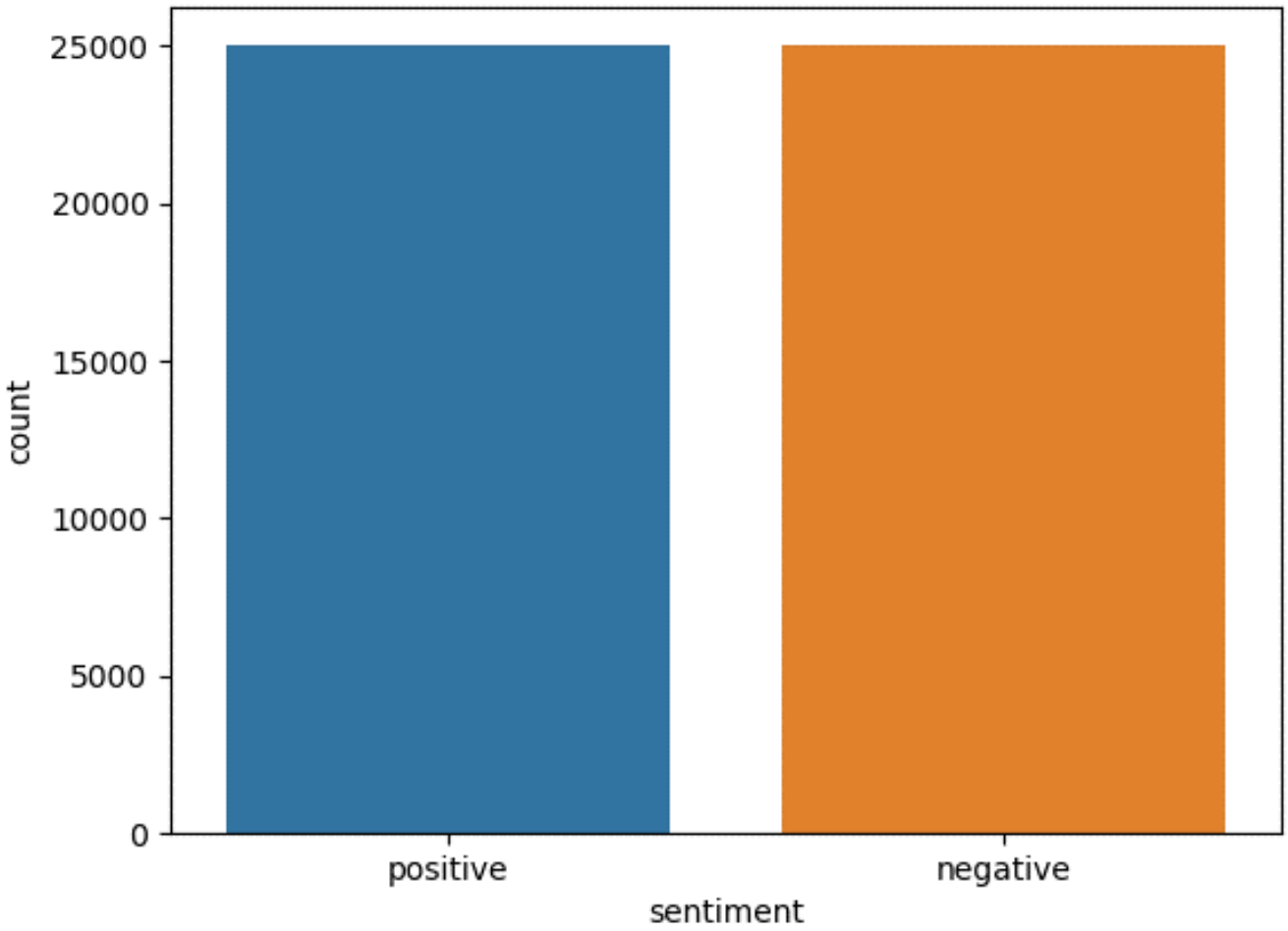

Figure 1: Sample Count

Data Preprocessing

- Our dataset contained punctuations and HTML tags.
- We will define a function that takes a text string as a parameter and then performs preprocessing on the string to remove special characters and HTML tags from the string. Finally, the string is returned to the calling function. Look at the following script:

Preprocessing Script

```
def remove_tags(text):
    return TAG_RE.sub('', text)
def preprocess_text(sen):
    # Removing html tags
    sentence = remove_tags(sen)

    # Remove punctuations and numbers
    sentence = re.sub('[^a-zA-Z]', ' ', sentence)

    # Single character removal
    sentence = re.sub(r"\s+[a-zA-Z]\s+", ' ', sentence)

    # Removing multiple spaces
    sentence = re.sub(r'\s+', ' ', sentence)

    return sentence
```

Store reviews in list

```
X = []
sentences = list(movie_reviews['review'])
for sen in sentences:
    X.append(preprocess_text(sen))

print(X[3])
```

Labels to Digits

Since we only have two labels in the output i.e. “positive” and “negative”. We can simply convert them into integers by replacing “positive” with digit 1 and negative with digit 0.

```
y=movie_reviews['sentiment']
y=np.array(list(map(lambda x: 1 if x=="positive" else 0, y)))
```

Split the dataset

Divide our dataset into train and test sets. The train set will be used to train our deep learning models while the test set will be used to evaluate how well our model performs.

```
X_train, X_test, y_train, y_test =
  train_test_split(X, y, test_size=0.20, random_state=42)
```

Embedding Layer

The embedding layer converts our textual data into numeric data and is used as the first layer for the deep learning models in Keras.

As a first step, we will use the Tokenizer class from the keras.preprocessing.text module to create a word-to-index dictionary. In the word-to-index dictionary, each word in the corpus is used as a key, while a corresponding unique index is used as the value for the key. Execute the following script:

```
tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(X_train)

X_train = tokenizer.texts_to_sequences(X_train)
X_test = tokenizer.texts_to_sequences(X_test)
```

Padding

- If you view the X_train variable in variable explorer, you will see that it contains 40,000 lists where each list contains integers. Each list actually corresponds to each sentence in the training set. You will also notice that the size of each list is different. This is because sentences have different lengths.
- We set the maximum size of each list to 100. You can try a different size. The lists with size greater than 100 will be truncated to 100. For the lists that have length less than 100, we will add 0 at the end of the list until it reaches the max length. This process is called padding.
- The following script finds the vocabulary size and then perform padding on both train and test set.

Padding script

```
# Adding 1 because of reserved 0 index
vocab_size = len(tokenizer.word_index) + 1

maxlen = 100

X_train = pad_sequences(X_train, padding='post',
      maxlen=maxlen)
X_test = pad_sequences(X_test, padding='post', maxlen=maxlen)
```

Padding

- Now if you view the X_train or X_test, you will see that all the lists have same length i.e. 100. Also, the vocabulary_size variable now contains a value 92547 which means that our corpus has 92547 unique words.
- We will use GloVe embeddings to create our feature matrix. In the following script we load the GloVe word embeddings and create a dictionary that will contain words as keys and their corresponding embedding list as values.

GloVe

```
from numpy import array
from numpy import asarray
from numpy import zeros

embeddings_dictionary = dict()
glove_file = open('glove.6B.100d.txt', encoding="utf8")

for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions = asarray(records[1:], dtype='float32')
    embeddings_dictionary [word] = vector_dimensions
glove_file.close()
```

Embedding Matrix

Finally, we will create an embedding matrix where each row number will correspond to the index of the word in the corpus. The matrix will have 100 columns where each column will contain the GloVe word embeddings for the words in our corpus.

```
embedding_matrix = zeros((vocab_size, 100))
for word, index in tokenizer.word_index.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix[index] = embedding_vector
```

Once you execute the above script, you will see that embedding_matrix will contain 92547 rows (one for each word in the corpus). Now we are ready to create our deep learning models.

Text Classification with Simple Neural Network

The first model that we are going to develop is a simple neural network. Look at the following script:

```
model = Sequential()
embedding_layer = Embedding(vocab_size, 100,
  weights=[embedding_matrix], input_length=maxlen,
    trainable=False)
model.add(embedding_layer)

model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
```

Compile and Train

```
model.compile(optimizer='adam',
  loss='binary_crossentropy', metrics=['acc'])

print(model.summary())
history = model.fit(X_train, y_train,
  batch_size=128, epochs=6, verbose=1,
    validation_split=0.2)
score = model.evaluate(X_test, y_test, verbose=1)
print("Test Score:", score[0])
print("Test Accuracy:", score[1])
```

Text Classification with a Convolutional Neural Network

Let's create a simple convolutional neural network with 1 convolutional layer and 1 pooling layer. Remember, the code up to the creation of the embedding layer will remain same:

```
model = Sequential()
embedding_layer = Embedding(vocab_size, 100,
  weights=[embedding_matrix], input_length=maxlen,
    trainable=False)
model.add(embedding_layer)
model.add(Conv1D(128, 5, activation='relu'))
model.add(GlobalMaxPooling1D())
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
  loss='binary_crossentropy', metrics=['acc'])
```

Text Classification with Recurrent Neural Network (LSTM)

In this section, we will use an LSTM (Long Short Term Memory network) which is a variant of RNN, to solve sentiment classification problem.

```
model = Sequential()
embedding_layer = Embedding(vocab_size, 100,
  weights=[embedding_matrix], input_length=maxlen,
    trainable=False)
model.add(embedding_layer)
model.add(LSTM(128))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
  loss='binary_crossentropy', metrics=['acc'])
```