Sentiment Analysis / Text Classification Using RNN(Bi-LSTM)(Recurrent Neural Network)

Saad Arshad Follow
Sep 21, 2019 · 5 min read

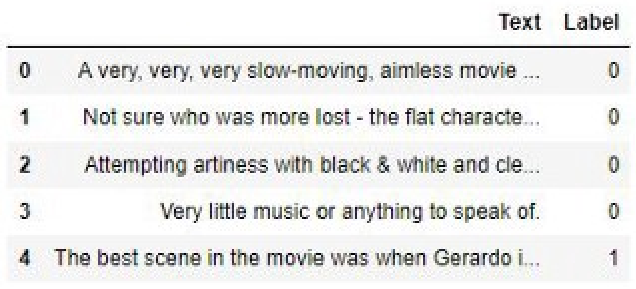
There are lots of applications of text classification. For example, hate speech detection, intent classification, and organizing news articles. The focus of this article is Sentiment Analysis which is a text classification problem. We will be classifying the IMDB comments into two classes i.e. positive and negative.

We use Python and Jupyter Notebook to develop our system, the libraries we will use include Keras, Gensim, Numpy, Pandas, Regex(re) and NLTK. We will also use Google News Word2Vec Model. The complete code and data can be downloaded from here.

Data Exploration

First, we have a look at our data. As the data file is a tab-separated file(tsv), we will read it by using pandas and pass arguments to tell the function that the delimiter is tab and there is no header in our data file. Then we set the header of our data frame.

```
import pandas as pd
data = pd.read_csv('imdb_labelled.tsv',
                   header = None,
                   delimiter='\t')

data.columns = ['Text', 'Label']
df.head()
```



	Text	Label
0	A very, very, very slow-moving, aimless movie ...	0
1	Not sure who was more lost - the flat characte...	0
2	Attempting artiness with black & white and cle...	0
3	Very little music or anything to speak of.	0
4	The best scene in the movie was when Gerardo i...	1

Then we check the shape of data

```
data.shape
```

Now we see the class distribution. We have 386 positive and 362 negative examples.

```
data.Label.value_counts()
```

. . .

Data Cleaning

The first step in data cleaning is to remove punctuation marks. We simply do it by using Regex. After removing the punctuation marks the data is saved in the same data frame.

```
import re

def remove_punct(text):
    text_nopunct = ''
    text_nopunct = re.sub('['+string.punctuation+']', '', text)
    return text_nopunct

data['Text_Clean'] = data['Text'].apply(lambda x: remove_punct(x))
```

In the next step, we tokenize the comments by using NLTK's word_tokenize. If we pass a string 'Tokenizing is easy' to word_tokenize. The output is ['Tokenizing', 'is', 'easy']

```
from nltk import word_tokenize
tokens = [word_tokenize(sen) for sen in data.Text_Clean]
```

Then we lower case the data.

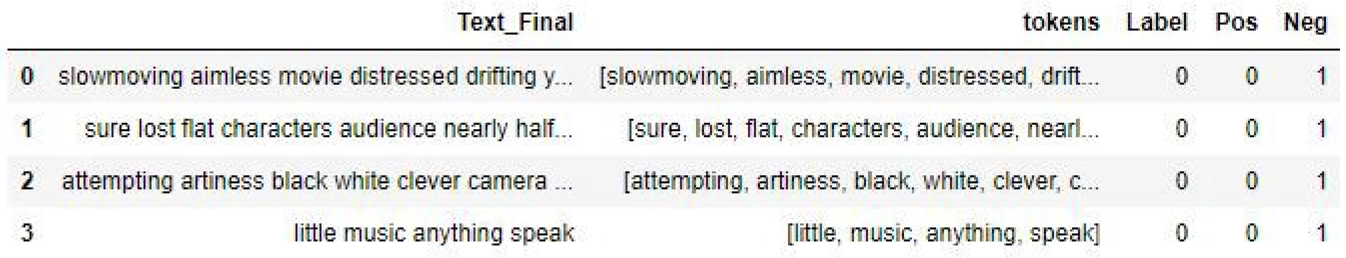
```
def lower_token(tokens):
    return [w.lower() for w in tokens]

lower_tokens = [lower_token(token) for token in tokens]
```

After lower casing the data, stop words are removed from data using NLTK's stopwords.

```
from nltk.corpus import stopwords

stoplist = stopwords.words('english')

def removeStopWords(tokens):
    return [word for word in tokens if word not in stoplist]

filtered_words = [removeStopWords(sen) for sen in lower_tokens]

data['Text_Final'] = [' '.join(sen) for sen in filtered_words]
data['tokens'] = filtered_words
```

As our problem is a binary classification. We need to pass our model a two-dimensional output vector. For that, we add two one hot encoded columns to our data frame.

```
pos = []
neg = []
for l in data.Label:
    if l == 0:
        pos.append(0)
        neg.append(1)
    elif l == 1:
        pos.append(1)
        neg.append(0)

data['Pos']= pos
data['Neg']= neg

data = data[['Text_Final', 'tokens', 'Label', 'Pos', 'Neg']]
data.head()
```



	Text_Final	tokens	Label	Pos	Neg
0	slowmoving aimless movie distressed drifting y...	[slowmoving, aimless, movie, distressed, drift...	0	0	1
1	sure lost flat characters audience nearly half...	[sure, lost, flat, characters, audience, nearl...	0	0	1
2	attempting artiness black white clever camera ...	[attempting, artiness, black, white, clever, c...	0	0	1
3	little music anything speak	[little, music, anything, speak]	0	0	1

. . .

Splitting Data into Test and Train

Now we split our data set into train and test. We will use 90 % data for training and 10 % for testing. We use random state so every time we get the same training and testing data.

```
data_train, data_test = train_test_split(data,
                                         test_size=0.10,
                                         random_state=42)
```

Then we build training vocabulary and get maximum training sentence length and total number of words training data.

```
all_training_words = [word for tokens in data_train["tokens"] for
word in tokens]
training_sentence_lengths = [len(tokens) for tokens in
data_train["tokens"]]
TRAINING_VOCAB = sorted(list(set(all_training_words)))
print("%s words total, with a vocabulary size of %s" %
(len(all_training_words), len(TRAINING_VOCAB)))
print("Max sentence length is %s" % max(training_sentence_lengths))
```

Then we build testing vocabulary and get maximum testing sentence length and total number of words in testing data.

```
all_test_words = [word for tokens in data_test[“tokens”] for word in
tokens]
test_sentence_lengths = [len(tokens) for tokens in
data_test[“tokens”]]
TEST_VOCAB = sorted(list(set(all_test_words)))
print(“%s words total, with a vocabulary size of %s” %
(len(all_test_words), len(TEST_VOCAB)))
print(“Max sentence length is %s” % max(test_sentence_lengths))
```

. . .

Loading Google News Word2Vec model

Now we will load the Google News Word2Vec model. This step may take some time. You can use any other pre-trained word embeddings or train your own word embeddings if you have sufficient amount of data.

```
word2vec_path = 'GoogleNews-vectors-negative300.bin.gz'
word2vec = models.KeyedVectors.load_word2vec_format(word2vec_path,
binary=True)
```

. . .

Tokenize and Pad sequences

Each word is assigned an integer and that integer is placed in a list. As all the training sentences must have same input shape we pad the sentences.

For example if we have a sentence “How text to sequence and padding works”. Each word is assigned a number. We suppose how = 1, text = 2, to = 3, sequence =4, and = 5, padding = 6, works = 7. After texts_to_sequences is called our sentence will look like [1, 2, 3, 4, 5, 6, 7]. Now we suppose our MAX_SEQUENCE_LENGTH = 10. After padding our sentence will look like [0, 0, 0, 1, 2, 3, 4, 5, 6, 7]

We do same for testing data also. For complete code visit.

```
tokenizer = Tokenizer(num_words=len(TRAINING_VOCAB), lower=True,
char_level=False)
tokenizer.fit_on_texts(data_train[“Text_Final”].tolist())
training_sequences =
tokenizer.texts_to_sequences(data_train[“Text_Final”].tolist())

train_word_index = tokenizer.word_index
print(‘Found %s unique tokens.’ % len(train_word_index))

train_cnn_data = pad_sequences(training_sequences,
                               maxlen=MAX_SEQUENCE_LENGTH)
```

Now we will get embeddings from Google News Word2Vec model and save them corresponding to the sequence number we assigned to each word. If we could not get embeddings we save a random vector for that word.

```
train_embedding_weights = np.zeros((len(train_word_index)+1,
 EMBEDDING_DIM))for word,index in train_word_index.items():
 train_embedding_weights[index,:] = word2vec[word] if word in
word2vec else
np.random.rand(EMBEDDING_DIM)print(train_embedding_weights.shape)
```

. . .

Defining RNN

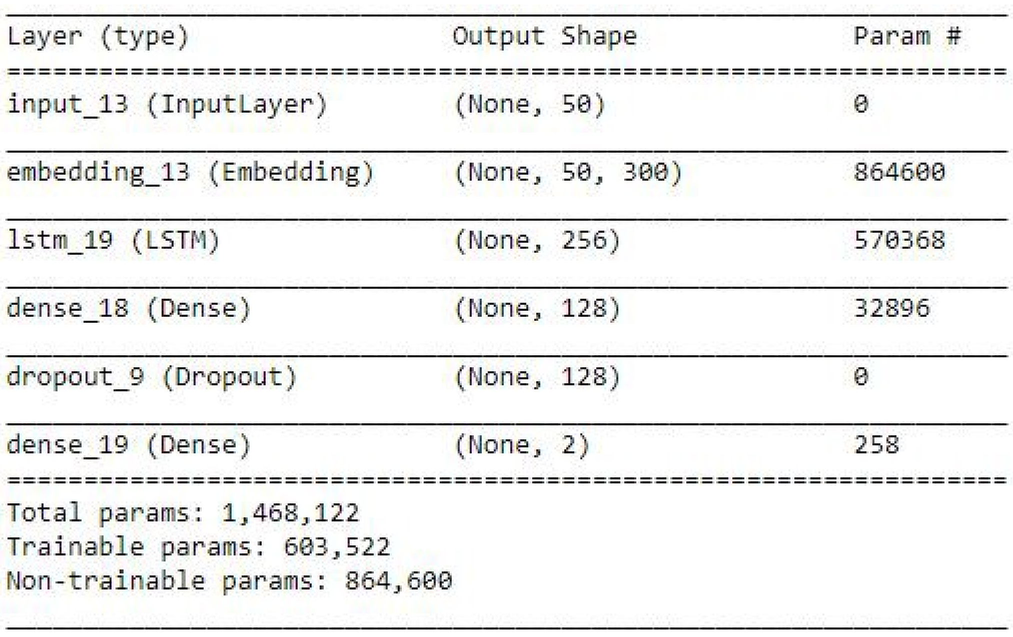
Text as a sequence is passed to a RNN. The embeddings matrix is passed to embedding_layer. The outpu of embedding layer is passed to LSTM layer. This model has 256 LSTM cells. Here we ignore the hidden states of all of the cells and only take the output from last LSTM cell. The output is passed to a Dense layer then Dropout and then Final Dense layer is applied.

model.summary() will print a brief summary of all the layers with there output shapes.

```
def rnn(embeddings,
        max_sequence_length,
        num_words,
        embedding_dim,
        labels_index):

    embedding_layer = Embedding(num_words,
                                embedding_dim,
                                weights=[embeddings],
                                input_length=max_sequence_length,
                                trainable=False)

    sequence_input = Input(shape=(max_sequence_length,),
                                  dtype='int32')
    embedded_sequences = embedding_layer(sequence_input)

    lstm = LSTM(256)(embedded_sequences)

    x = Dense(128, activation='relu')(lstm)
    x = Dropout(0.2)(x)
    preds = Dense(labels_index, activation='sigmoid')(x)

    model = Model(sequence_input, preds)
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
    model.summary()
    return model
```

Now we will execute the function.

```
model = rnn(train_embedding_weights,
            MAX_SEQUENCE_LENGTH,
            len(train_word_index)+1,
            EMBEDDING_DIM,
            len(list(label_names)))
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_13 (InputLayer)        (None, 50)                0
_________________________________________________________________
embedding_13 (Embedding)     (None, 50, 300)           864600
_________________________________________________________________
lstm_19 (LSTM)               (None, 256)               570368
_________________________________________________________________
dense_18 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_9 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_19 (Dense)             (None, 2)                 258
=================================================================
Total params: 1,468,122
Trainable params: 603,522
Non-trainable params: 864,600
_________________________________________________________________
```

. . .

Training RNN

The number of epochs is the amount to which your model will loop around and learn, and batch size is the amount of data which your model will see at a single time. As we are training on small data set in just a few epochs our model will over fit.

```
num_epochs = 5
batch_size = 32
hist = model.fit(x_train,
                 y_tr,
                 epochs=num_epochs,
                 validation_split=0.1,
                 shuffle=True,
                 batch_size=batch_size)
```

. . .

Testing out model

Wow! with just five iterations and a small data set we were able to get 80 % accuracy.

```
predictions = model.predict(test_cnn_data,
                            batch_size=1024,
                            verbose=1)
labels = [1, 0]
prediction_labels=[]
for p in predictions:
    prediction_labels.append(labels[np.argmax(p)])
sum(data_test.Label==prediction_labels)/len(prediction_labels)
```

Machine Learning Naturallanguageprocessing Sentiment Analysis Text Classification

Recurrent Neural Network