index

June 23, 2022

1 Classification With Word Embeddings - Codealong

1.1 Introduction

In this lesson, you'll use everything you've learned in this section to perform text classification using word embeddings!

1.2 Objectives

You will be able to:

- Use pretrained word embeddings from popular pretrained models such as GloVe
- Incorporate embedding layers into neural networks using Keras

1.3 Getting Started

Load the data, and all the libraries and functions.

[1]:
```
import pandas as pd

import numpy as np
np.random.seed(0)

from nltk import word_tokenize
from gensim.models import word2vec
```

Now, load the dataset. You'll be working with the same dataset you worked with in the previous lab for this section, which you'll find inside `News_Category_Dataset_v2.zip`. ***Go into the repo and unzip this file before continuing.***

Once you've unzipped this dataset, go ahead and use `pandas` to read the data stored in `'News_Category_Dataset_v2.json'` in the cell below.

NOTE: When using the `pd.read_json()` function, be sure to include the `lines=True` parameter, or else it will crash!

[2]:
```
df = pd.read_json('News_Category_Dataset_v2.json', lines=True)
df = df.sample(frac=0.2)
print(len(df))
df.head()
```

```
40171
```

[2]:
```
                                                authors        category  \
23341                                                           POLITICS
100639                                 JamesMichael Nichols    QUEER VOICES
184179              Party Earth, Contributor\nContributor           TRAVEL
136649  Shelly Ulaj, Contributor\nFounder and CEO of W…         DIVORCE
196185                                     Ellie Krupnick  STYLE & BEAUTY

             date                                                headline  \
23341  2017-06-21  Jared Kushner Arrives In Israel For Whirlwind …
100639 2015-01-23  'The Best Thing Is To See How Much Love Can Do…
184179 2012-07-25  Berlin's Nightlife: 48 Hours You Might Not Rem…
136649 2013-12-13              Finding Strength to Stand on Your Own
196185 2012-03-18  Alexander Wang Lawsuit Will Move To Federal Co…

                                                      link  \
23341   https://www.huffingtonpost.com/entry/jared-kus…
100639  https://www.huffingtonpost.com/entry/stacy-hol…
184179  https://www.huffingtonpost.com/entry/berlins-n…
136649  https://www.huffingtonpost.com/entry/finding-s…
196185  https://www.huffingtonpost.com/entry/alexander…

                                       short_description
23341   It remains unclear what approach the White Hou…
100639
184179  If you think spending time boozing and schmooz…
136649  I was so used to being taken care of by family…
196185  Representatives of Alexander Wang's brand cont…
```

Now, let's transform the dataset, as you did in the previous lab.

In the cell below:

- Store the column that will be the target, `'category'`, in the variable `target`
- Combine the `'headline'` and `'short_description'` columns and store the result in a column called `'combined_text'`. When concatenating these two columns, make sure they are separated by a space character (`' '`)
- Use the `'combined_text'` column's `.map()` method to use the `word_tokenize` function on every piece of text

- Store the `.values` attribute from the newly tokenized `'combined_text'` column in the variable `data`

[3]:
```
target = df['category']
df['combined_text'] = df['headline'] + ' ' + df['short_description']
data = df['combined_text'].map(word_tokenize).values
```

1.4 Loading A Pretrained GloVe Model

For this lab, you'll be loading the pretrained weights from ***GloVe*** (short for *Global Vectors for Word Representation*) from the Stanford NLP Group. These are commonly accepted as some of the best pre-trained word vectors available, and they're open source, so you can get them for free! Even the smallest file is still over 800 MB, so you'll you need to download this file manually.

Note that there are several different sizes of pretrained word vectors available for download from the page linked above – for the purposes of this lesson, you'll only need to use the smallest one, which still contains pretrained word vectors for over 6 billion words and phrases! To download this file, follow the link above and select the file called `glove.6b.zip`. For simplicity's sake, you can also start the download by clicking this link. You'll be using the GloVe file containing 50-dimensional word vectors for 6 billion words. Once you've downloaded the file, unzip it, and move the file `glove.6B.50d.txt` into the same directory as this Jupyter notebook.

1.4.1 Getting the Total Vocabulary

Although the pretrained GloVe data contains vectors for 6 billion words and phrases, you don't need all of them. Instead, you only need the vectors for the words that appear in the dataset. If a word or phrase doesn't appear in the dataset, then there's no reason to waste memory storing the vector for that word or phrase.

This means that you need to start by computing the total vocabulary of the dataset. You can do this by adding every word in the dataset into a Python `set` object. This is easy, since you've already tokenized each comment stored within `data`.

In the cell below, add every token from every comment in data into a set, and store the set in the variable `total_vocabulary`.

HINT: Even though this takes a loop within a loop, you can still do this with a one-line list comprehension!

[4]:
```
total_vocabulary = set(word for headline in data for word in headline)
```

[5]:
```
len(total_vocabulary)
print('There are {} unique tokens in the dataset.'.
  ↪format(len(total_vocabulary)))
```

```
There are 71173 unique tokens in the dataset.
```

Now that you have gotten the total vocabulary, you can get the appropriate vectors out of the GloVe file.

[6]:
```
glove = {}
with open('glove.6B.50d.txt', 'rb') as f:
    for line in f:
        parts = line.split()
        word = parts[0].decode('utf-8')
        if word in total_vocabulary:
            vector = np.array(parts[1:], dtype=np.float32)
            glove[word] = vector
```

After running the cell above, you now have all of the words and their corresponding vocabulary stored within the dictionary, `glove`, as key/value pairs.

Double-check that everything worked by getting the vector for a word from the `glove` dictionary. It's probably safe to assume that the word `'school'` will be mentioned in at least one news headline, so let's get the vector for it.

Get the vector for the word `'school'` from `glove` in the cell below.

[7]:
```
glove['school']
```

[7]:
```
array([-0.90629  ,  1.2485   , -0.79692  , -1.4027   , -0.038458 ,
       -0.25177  , -1.2838   , -0.58413  , -0.11179  , -0.56908  ,
       -0.34842  , -0.39626  , -0.0090178, -1.0691   , -0.35368  ,
       -0.052826 , -0.37056  ,  1.0931   , -0.19205  ,  0.44648  ,
        0.45169  ,  0.72104  , -0.61103  ,  0.6315   , -0.49044  ,
       -1.7517   ,  0.055979 , -0.52281  , -1.0248   , -0.89142  ,
        3.0695   ,  0.14483  , -0.13938  , -1.3907   ,  1.2123   ,
        0.40173  ,  0.4171   ,  0.27364  ,  0.98673  ,  0.027599 ,
       -0.8724   , -0.51648  , -0.30662  ,  0.37784  ,  0.016734 ,
        0.23813  ,  0.49411  , -0.56643  , -0.18744  ,  0.62809  ],
      dtype=float32)
```

Great—it worked! Now that you've gotten the word vectors for every word in the dataset, the next step is to combine all the vectors for a given headline into a ***Mean Embedding*** by finding the average of all the vectors in that headline.

1.5 Creating Mean Word Embeddings

For this step, it's worth the extra effort to write your own mean embedding vectorizer class, so that you can make use of pipelines from scikit-learn. Using pipelines will save us time and make the code a bit cleaner.

The code for a mean embedding vectorizer class is included below, with comments explaining what each step is doing. Take a minute to examine it and try to understand what the code is doing.

[8]:
```
class W2vVectorizer(object):

    def __init__(self, w2v):
        # Takes in a dictionary of words and vectors as input
        self.w2v = w2v
        if len(w2v) == 0:
            self.dimensions = 0
        else:
            self.dimensions = len(w2v[next(iter(glove))])

    # Note: Even though it doesn't do anything, it's required that this object␣
↪implement a fit method or else
    # it can't be used in a scikit-learn pipeline
    def fit(self, X, y):
```

```
        return self

    def transform(self, X):
        return np.array([
            np.mean([self.w2v[w] for w in words if w in self.w2v]
                    or [np.zeros(self.dimensions)], axis=0) for words in X])
```

1.6 Using Pipelines

Since you've created a mean vectorizer class, you can pass this in as the first step in the pipeline, and then follow it up with the model you'll feed the data into for classification.

Run the cell below to create pipeline objects that make use of the mean embedding vectorizer that you built above.

[9]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

rf =  Pipeline([('Word2Vec Vectorizer', W2vVectorizer(glove)),
                ('Random Forest', RandomForestClassifier(n_estimators=100,␣
 ↪verbose=True))])
svc = Pipeline([('Word2Vec Vectorizer', W2vVectorizer(glove)),
                  ('Support Vector Machine', SVC())])
lr = Pipeline([('Word2Vec Vectorizer', W2vVectorizer(glove)),
                ('Logistic Regression', LogisticRegression())])
```

Now, you'll create a list that contains a tuple for each pipeline, where the first item in the tuple is a name, and the second item in the list is the actual pipeline object.

[10]:
```
models = [('Random Forest', rf),
          ('Support Vector Machine', svc),
          ('Logistic Regression', lr)]
```

You can then use the list you've created above, as well as the `cross_val_score()` function from scikit-learn to train all the models, and store their cross validation scores in an array.

NOTE: Running the cell below may take several minutes!

[11]:
```
#  This cell may take several minutes to run
scores = [(name, cross_val_score(model, data, target, cv=2).mean()) for name,␣
 ↪model, in models]
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:   15.6s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.6s finished
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:   15.8s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.6s finished
//anaconda3/envs/learn-env/lib/python3.6/site-packages/sklearn/svm/base.py:193:
FutureWarning: The default value of gamma will change from 'auto' to 'scale' in
version 0.22 to account better for unscaled features. Set gamma explicitly to
'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)
//anaconda3/envs/learn-env/lib/python3.6/site-packages/sklearn/svm/base.py:193:
FutureWarning: The default value of gamma will change from 'auto' to 'scale' in
version 0.22 to account better for unscaled features. Set gamma explicitly to
'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)
//anaconda3/envs/learn-env/lib/python3.6/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
//anaconda3/envs/learn-env/lib/python3.6/site-
packages/sklearn/linear_model/logistic.py:469: FutureWarning: Default
multi_class will be changed to 'auto' in 0.22. Specify the multi_class option to
silence this warning.
  "this warning.", FutureWarning)
//anaconda3/envs/learn-env/lib/python3.6/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
//anaconda3/envs/learn-env/lib/python3.6/site-
packages/sklearn/linear_model/logistic.py:469: FutureWarning: Default
multi_class will be changed to 'auto' in 0.22. Specify the multi_class option to
silence this warning.
  "this warning.", FutureWarning)
```

[12]:
```
scores
```

[12]:
```
[('Random Forest', 0.31960910174587964),
 ('Support Vector Machine', 0.3036012008096788),
 ('Logistic Regression', 0.3255087322734529)]
```

These scores may seem pretty low, but remember that there are 41 possible categories that headlines could be classified into. This means the naive accuracy rate (random guessing) would achieve an accuracy of just over 0.02! Our models have plenty of room for improvement, but they do work!

1.7 Deep Learning With Word Embeddings

To end, you'll see an example of how you can use an ***Embedding Layer*** inside of a deep neural network to compute your own word embedding vectors on the fly, right inside the model!

Don't worry if you don't understand the code below just yet – you'll be learning all about ***Sequence***

Models like the one below in the next section!

Run the cells below.

[13]:
```
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Input, Dense, LSTM, Embedding
from keras.layers import Dropout, Activation, Bidirectional, GlobalMaxPool1D
from keras.models import Sequential
from keras import initializers, regularizers, constraints, optimizers, layers
from keras.preprocessing import text, sequence
```

```
Using TensorFlow backend.
```

Next, you'll convert the labels to a one-hot encoded format.

[14]:
```
y = pd.get_dummies(target).values
```

Now, you'll preprocess the text data. To do this, you start from the step where you combined the headlines and short description. You'll then use Keras' preprocessing tools to tokenize each example, convert them to sequences, and then pad the sequences so they're all the same length.

Note how during the tokenization step, you set a parameter to tell the tokenizer to limit the overall vocabulary size to the 20000 most important words.

[15]:
```
tokenizer = text.Tokenizer(num_words=20000)
tokenizer.fit_on_texts(list(df['combined_text']))
list_tokenized_headlines = tokenizer.texts_to_sequences(df['combined_text'])
X_t = sequence.pad_sequences(list_tokenized_headlines, maxlen=100)
```

Now, construct the neural network. In the embedding layer, you specify the size you want the word vectors to be, as well as the size of the embedding space itself. The embedding size you specified is 128, and the size of the embedding space is best as the size of the total vocabulary that we're using. Since you limited the vocabulary size to 20000, that's the size you choose for the embedding layer.

Once the data has passed through an embedding layer, you feed this data into an LSTM layer, followed by a Dense layer, followed by output layer. You also add some Dropout layers after each of these layers, to help fight overfitting.

Our output layer is a Dense layer with 41 neurons, which corresponds to the 41 possible classes in the labels. You set the activation function for this output layer to `'softmax'`, so that the network will output a vector of predictions, where each element's value corresponds to the percentage chance that the example is the class that corresponds to that element, and where the sum of all elements in the output vector is 1.

[16]:
```
model = Sequential()
```

[17]:
```
embedding_size = 128
model.add(Embedding(20000, embedding_size))
model.add(LSTM(25, return_sequences=True))
model.add(GlobalMaxPool1D())
```

```
model.add(Dropout(0.5))
model.add(Dense(50, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(41, activation='softmax'))
```

Once you have designed the model, you still have to compile it, and provide important parameters such as the loss function to use (`'categorical_crossentropy'`, since this is a multiclass classification problem), and the optimizer to use.

[18]:
```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

After compiling the model, you quickly check the summary of the model to see what the model looks like, and make sure the output shapes line up with what you expect.

[19]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, None, 128)         2560000
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 25)          15400
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 25)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 25)                0
_________________________________________________________________
dense_1 (Dense)              (None, 50)                1300
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_2 (Dense)              (None, 41)                2091
=================================================================
Total params: 2,578,791
Trainable params: 2,578,791
Non-trainable params: 0
_________________________________________________________________
```

Finally, you can fit the model by passing in the data, the labels, and setting some other hyperparameters such as the batch size, the number of epochs to train for, and what percentage of the training data to use for validation data.

If trained for three epochs, you'll find the model achieves a validation accuracy around 40%.

Run the cell below for three epochs. Note that this is a large network, so the training will take some time!

[20]:
```
#  This cell may take several minutes to run
model.fit(X_t, y, epochs=3, batch_size=32, validation_split=0.1)
```

```
Train on 36153 samples, validate on 4018 samples
Epoch 1/3
36153/36153 [==============================] - 97s 3ms/step - loss: 3.1289 -
acc: 0.2131 - val_loss: 2.6295 - val_acc: 0.3342
Epoch 2/3
36153/36153 [==============================] - 94s 3ms/step - loss: 2.5276 -
acc: 0.3542 - val_loss: 2.3489 - val_acc: 0.4052
Epoch 3/3
36153/36153 [==============================] - 94s 3ms/step - loss: 2.2191 -
acc: 0.4165 - val_loss: 2.2006 - val_acc: 0.4368
```

[20]:
```
<keras.callbacks.History at 0x1a47a28908>
```

After two epochs, the model performs as well as the shallow algorithms you tried above. However, the LSTM Network was able to achieve a validation accuracy around 40% after only three epochs of training. It's likely that if you trained for more epochs or added in the rest of the data, the performance would improve even further (but the run time would get much, much longer).

It's common to add embedding layers in LSTM networks, because both are special tools most commonly used for text data. The embedding layer creates it's own vectors based on the language in the text data it trains on, and then passes that information on to the LSTM network one word at a time. You'll learn more about LSTMs and other kinds of ***Recurrent Neural Networks*** in the next section!

1.8 Summary

In this codealong, you used everything you know about word embeddings to perform text classification, and then you built a multi-layer perceptron model that incorporated a word embedding layer in it's own architecture!