Sentiment Analysis

Leila Khalili

September 2020

1 Baseline 1: gensim's word2vec

The code for this baseline is named `b_word2vec.py`.

1.1 Tokenization

: The training data is tokenized using the Punkt sentence tokenizer of Natural Language Toolkit[1]. In this method stop words are note removed. Moreover, no cleaning has been applied to the data.

```
tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
```

The input for the next step should be a numpy.ndarray. So using functions `review_sentences` and `review_wordlist` in the `util_func.py`, we can have tokenized sentences in an nd.arrary.

1.2 word embedding

1.2.1 without pre-trained model

For word embedding I used word2vec in gensim library [2]. I trained the gensim model by the training data provided for the assignment. The model is saved so future runs it can just easily be loaded. In the parameters of the function `Word2Vec`, `sentences` is the output of tokenization part. This model vectorize the words into a vector of size 300.

```
from gensim.models import word2vec
print("Training model")
num_features = 300
model_name = "300features_40minwords_10context"
model = word2vec.Word2Vec(sentences,\
                          min_count=5,
                         window=5,
                         size=num_features,
                         sample=6e-4,
                         alpha=0.03,
                         workers=2)

model.init_sims(replace=True)
# Saving the model
model.save(model_name)
```

In the next step each sentence is represented by a single vector which is the average of vectors for each word in the sentence.

In the last step the sentences in the test data is tokenized. Then the trained model from the last section is used for word embedding. Then by averaging the vector representation of the sentences is calculated.

1.2.2 with "Google News word2vec" pre-trained model

In this case I downloaded the "Google News word2vec" pre-train model [3] and loaded it as the model in my code.

```
from gensim.models import KeyedVectors
model = KeyedVectors.load_word2vec_format('./input/GoogleNews-vectors-negative300.bin', binary=True)
```

1.3 Classification

For classification, I used " random forest" classifier. The `result` is the predicted labels of the test dataset.

```
# Fitting a random forest classifier to the training data
forest = RandomForestClassifier(n_estimators = 100)
forest = forest.fit(trainDataVecs, train["label"])
result = forest.predict(testDataVecs)
```

The accuracy, F1 score and confusion matrix for **without pre-trained data** are calculated

```
[[  0  41  41]
 [  1 186 116]
 [  0 124 174]]
The accuracy of the model in the tested data is:  0.527086383601757
The f1 score of the model in the tested data is:  0.493733306030092
Program Executed in 2.3151502039982006
```

The labled test data are saved in the file named `testing_output_word2vec.csv`.

The accuracy, F1 score and confusion matrix for **with "Google News word2vec" pre-trained model** are calculated

```
[[  0  40  42]
 [  0 230  73]
 [  1 114 183]]
The accuracy of the model in the tested data is:  0.6046852122986823
The f1 score of the model in the tested data is:  0.5649811069837029
Program Executed in 877.049520508037
```

The labled test data are saved in the file named `testing_output_google.csv`.

2 Baseline 2: TF-IDF

2.1 Tokenization

The input data for TF-IDF is the strings. I just used the biult in tokenizer of this function.It worth mentioning that in this method stop words will be removed from the tokens.

2.2 word embedding

I used the function `tfidf` to vectorize the sentences.

```
def tfidf(data):
tfidf_vectorizer = TfidfVectorizer()
vec = tfidf_vectorizer.fit_transform(data)
return vec, tfidf_vectorizer
```

Then I vectorized the train and test data set

```
X_train_tfidf, tfidf_vectorizer = uf.tfidf(train['sentence'])
trainDataVecs = X_train_tfidf.toarray()
X_test_tfidf = tfidf_vectorizer.transform(test['sentence'])
testDataVecs = X_test_tfidf.toarray()
```

2.3 Classification

I used random forest to classify the data as mentioned in section 1.3

The accuracy, F1 score and confusion matrix are calculated

```
[[  0  36  46]
 [  0 189 114]
 [  0 107 191]]
The accuracy of the model in the tested data is:  0.5563689604685212
The f1 score of the model in the tested data is:  0.5208940865327627
Program Executed in 5.290632124058902
```

The labled test data are saved in the file named `testing_output_tf_idf.csv`.

model	Pre-train	accuracy	F1	time
word2vec	F	52.71	49.37	2.32
TF-IDF	F	55.64	52.09	5.29
word2vec	T	60.47	56.5	877.05
USE	T	65.15	61.4	41.48

Table 1: A summary of efficiency of the methods used in this assignment.

3 proposed model

Among all the methods I used, Google's Universal Sentence encoder was the most efficient one both in accuracy and operation time.

For this method I first downloaded a pretrained universal sentence encoder

```
#download the model to local so it can be used again and again
!mkdir ./input/module/module_useT
# Download the module, and uncompress it to the destination folder.
!curl -L "https://tfhub.dev/google/universal-sentence-encoder-large/3?tf-hub-format=compressed"
| tar -zxvC ./input/module/module_useT
```

Then the module is loaded to the disk and used to embed the sentences using the following code

```
def embed_useT(module):
    with tf.Graph().as_default():
        sentences = tf.placeholder(tf.string)
        embed = hub.Module(module)
        embeddings = embed(sentences)
        session = tf.train.MonitoredSession()
    return lambda x: session.run(embeddings, {sentences: x})
x_training = np.reshape(train.sentence.values,(len(train.sentence.values)))
x_test = np.reshape(test.sentence.values,(len(test.sentence.values)))
embed_fn = uf.embed_useT('./input/module/module_useT')
x_training = embed_fn(x_training)
x_test = embed_fn(x_test)
```

Finally simillar to 1.3 a random forest model is applied to classify the data. The output file for the proposed method is saved as `testing_output_proposed.csv`. The method with a higher accuracy is the USE.

```
[[  2  41  39]
 [  1 243  59]
 [  2  96 200]]
The accuracy of the model in the tested data is:  0.6515373352855052
The f1 score of the model in the tested data is:  0.6140187410758821
Program Executed in 41.481160297989845
```

For the proposed model I tried different text preprocessing techniques, like removing stop-words, punctuation, lemmatization and stemming. However, they didn't improve the accuracy of the model. It is mentioned in the website of USE that "The module performs best effort text input preprocessing, therefore it is not required to preprocess the data before applying the module" [4].

4 Discussion

Among all the methods I tried there were two which gave better answers which are compared in the following table 1.

The pre-trained word2vec model increases the accuracy considerably and the reason is that our training data is relatively small. However, the computation time is very high. The proposed model which is "universal sentence encoding" is giving the best accuracy and time. As it mentioned in the assignment instruction averaging the vectorized words is not a good representation for sentence representation. So even though the google's word2vec model is trained on lots of data it is not doing a good job.

Comparing word2vec and tf-idf, I think for sentiment analysis word2vec is a better choice. tf-idf is better in the applications that want to reflect how important a word is in a document. However, word2vec gets deeper in the document and helps to derive relations between simillar words.

References

[1] MS Windows NT kernel description. https://www.nltk.org/$_m$*odules/nltk/tokenize/punkt.html.Accessed* : 2010−09−30.

[2] MS Windows NT kernel description. https://radimrehurek.com/gensim/models/word2vec.html. Accessed: 2010-09-30.

[3] MS Windows NT kernel description. https://github.com/mmihaltz/word2vec-GoogleNews-vectors. Accessed: 2010-09-30.

[4] MS Windows NT kernel description. https://tfhub.dev/google/universal-sentence-encoder/1. Accessed: 2010-09-30.