CSC791 NLP P1: Sentimental Analysis

Shreyas Chikkballapur Muralidhara - schikkb

Report Requirement:

1. Report and compare the performance of the **baseline** with **TWO** word embedding techniques, as well as the performance of the **proposed solution**.
2. In your report, you need to include screenshots of the metrics for performance at least with the accuracy and F1 score. Pick and choose other metrics as you see fit.
3. Include code snippets and explain what method is on each requirement above: 1) code of method to tokenize, 2) code of method to calculate vectors, 3) code of generating the model you applied to classify training and testing dataset.

Solution:

For the sentimental analysis, I have used the most popular and preferred 2 word embedding models as **baseline**:

Baseline 1 – **Word2Vec** word embedding model

Baseline 2 **– TF-IDF (Term Frequency-Inverse Document Frequency)** word embedding model

Proposed Solution Model: **Doc2Vec** sentence embedding model

PERFORMANCE and METRICS -

Base Line 1 - Word2Vec

Word2Vec is the most popular and first neural net word embedding model, that is preferred for word embedding. There are 3 different types of Word2Vec Parameter learning model – **One-word** context, **Multi-word** context and **Skip-gram** model. I have implemented the Skip-gram model, that predicts the context of words having one target word on the input. As suggested in problem description, I have implemented Word2Vec using the **Gensim library** in python3.

The baseline model was trained using corpus containing 1660 statements with their corresponding sentiment labels. The dataset was split into (80%) training -1328 statements and (20%) validation – 332 statements. But since the class labels were not balanced as there were: Class 1 – 736, Class 2 – 661 and Class 3 – 263, Word2vec model was trained lot on Class 1 & 2, and less on Class 0. Training the model on **Decision Tree Classifier** with default hyperparameters, where model couldn't recognize the class 0 labels in the validation dataset. Hence the overall accuracy turned out be lower even though class 1 and class 2 had better Validation accuracy.

Word2Vec Validation Metrics – Accuracy, F1 score and Classification report:

```
Baseline Model 1 - Word2Vec validation metrics:
Accuracy - 0.4759
f1 score - 0.4323
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        53
           1       0.48      0.67      0.56       147
           2       0.48      0.45      0.46       132

    accuracy                           0.48       332
   macro avg       0.32      0.37      0.34       332
weighted avg       0.40      0.48      0.43       332
```

The test dataset consisted of 683 statements, where the class labels were Imbalanced Class 1 – 303, Class 2 – 298 and class 0 – 82. The Word2Vec model doesn't consider the link of words in sentence, rather it just averages the word vectors to generate the sentence vector. So, it doesn't capture the sequential and complex structure of sentence. Hence the **overall accuracy** of the base line model is quite low – **44.36%**, while the **F1 score** – **27.27%.** Below are the Word2Vec Test metrics:

Word2Vec Test Metrics – Accuracy, F1 score and Classification report:

```
Baseline Model 1 - Word2Vec Test metrics:
Accuracy - 0.4436
f1 score - 0.2727
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        82
           1       0.44      1.00      0.61       303
           2       0.00      0.00      0.00       298

    accuracy                           0.44       683
   macro avg       0.15      0.33      0.20       683
weighted avg       0.20      0.44      0.27       683
```

Base Line 2 - Term Frequency – Inverse Document Frequency (TF-IDF)

TF-IDF uses term weighting by exploration of useful statistical measure. The first part **term frequency** means the number of times a word occurs in the document by total number of words in the document. While second part **Inverse document frequency** means logarithmic value of inverse number of documents, in which the term we're interested occurs. This was implemented using the **Count Vectorizer** and **Tfidf Transformer** from the **sklearn.feature_extraction.text** module.

The term frequency was implemented using the *Count Vectorizer matrix* which generated the *frequency matrix for words in the corpus*. The *tfidf transformer replaced the counts for each cell by its tfidf score* for this term. The dataset was split into (80%) training -1328 statements and (20%) validation – 332 statements. But since the class labels were not balanced as there were: Class 1 – 736, Class 2 – 661 and Class 3 – 263, Word2vec model was trained lot on Class 1 & 2, and less on Class 0. The model was trained using **Naïve Bayes Classifier** with default hyperparameters, where model couldn't recognize the class 0 labels in the validation dataset. Hence the overall accuracy turned out be lower even though class 1 and class 2 had better Validation accuracy. **Validation accuracy** for Tfidf – **51.51%** while the **F1 Score** – **47.07%**.

TF-IDF Validation Metrics – Accuracy, F1 score and Classification report:

```
Baseline Model 2 - TF-IDF validation metrics:
Accuracy - 0.5151
f1 score - 0.4707
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        53
           1       0.54      0.62      0.58       147
           2       0.49      0.61      0.54       132

    accuracy                           0.52       332
   macro avg       0.34      0.41      0.37       332
weighted avg       0.43      0.52      0.47       332
```

The test dataset consisted of 683 statements, where the class labels were Imbalanced Class 1 – 303, Class 2 – 298 and class 0 – 82. The TF-IDF model doesn't consider the link of words in sentence, rather based on the frequency of word occurrence in the corpus generates the count that is later normalized to term frequency.

Even though it doesn't capture the sequential and complex structure of sentence, it doesn't average out the word vectors like the word2vec model, providing better TF normalized scores for each cell in the matrix. *Hence the **overall accuracy** of the base line model-2 is higher than the Word2Vec model with value– **59.44%**, while the **F1 score** – **55.62%**.* Below are the TF-IDF Test metrics:

TF-IDF Test Metrics – Accuracy, F1 score and Classification report:

```
Baseline Model 2 - TF-IDF Test metrics:
Accuracy - 0.5944
f1 score - 0.5562
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        82
           1       0.59      0.71      0.64       303
           2       0.60      0.64      0.62       298

    accuracy                           0.59       683
   macro avg       0.40      0.45      0.42       683
weighted avg       0.52      0.59      0.56       683
```

Proposed Solution – Doc2Vec with WordNet Lemmatizer and Porter Stemmer:

Doc2Vec which is inspired from word2Vec, is one of the most popular models for sentence vectorization. Earlier model TF-IDF uses BOW (bag of words) which doesn't capture the semantics and syntactic order of the words in the text. On the other hand, unsupervised algorithm Doc2vec learns the fixed length feature vectors for documents/paragraphs. Instead of generating word vectors as earlier and averaging the word vectors, it generates the sentence vectors from the tagged document. This was implemented using **Doc2vec, TaggedDocument** from the **genism.models.doc2vec** module.

As a part of pre-process, I have implemented 3 steps – **Removing punctuations and special characters** and replacing it by blanks using **re** module. **Stemming the sentences**/document vectors using **Porter Stemmer** from **nltk.stem** module. Stemmer replaces the word in the corpus by its root word by *Suffix stripping*. Lemmatize the sentences/ document vectors using **Word Net Lemmatizer** from **nltk.stem.** Word Net Lemmatizer uses POS (parts-of-Speech) parameter to identify the context in which it needs to lemmatize. It reduces the inflated words and hence ensures that root words will just belong to the language.

For the pre-processed set of sentences/ documents, we generate the tagged documents with the length as tag and words of the sentence. Develop the doc2vec model to consider each word of the corpus with min count = 1, and generate Vocabulary with the tagged document of training set. Later obtain the document vectors for the train, test and validation tagged documents. The dataset was split into (80%) training -1328 statements and (20%) validation – 332 statements. But since the class labels were not balanced as there were: Class 1 – 736, Class 2 – 661 and Class 3 – 263, Doc2vec

model was trained lot on Class 1 & 2, and less on Class 0. The model was trained using **Support Vector Classifier** with default hyperparameters, where model couldn't recognize the class 0 labels in the validation dataset. Hence the overall accuracy turned out be lower even though class 1 and class 2 had better Validation accuracy. **Validation accuracy** for Doc2Vec – **42.17%** while the **F1 Score** – **37.89%**.

Doc2Vec Validation Metrics – Accuracy, F1 score and Classification report:

```
Baseline Model 1 - Word2Vec validation metrics:
Accuracy - 0.4217
f1 score - 0.3789
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        53
           1       0.45      0.63      0.52       147
           2       0.38      0.36      0.37       132

    accuracy                           0.42       332
   macro avg       0.28      0.33      0.30       332
weighted avg       0.35      0.42      0.38       332
```

The test dataset consisted of 683 statements, where the class labels were Imbalanced Class 1 – 303, Class 2 – 298 and class 0 – 82. Tagged documents generated with pre-process reduces the Vocab size for the model, by mapping suffix stripping and mapping corpus to root words. Hence the document vectors generated capture sequential and complex structure of the sentence, which was not the case with Word2Vec and TF-IDF model. T*he **overall accuracy** of the Doc2Vec model is higher than the Word2Vec model with value– **59.44%**, while the **F1 score** – **55.62%**.* Below are the Doc2Vec Test metrics:

Doc2Vec Test Metrics – Accuracy, F1 score and Classification report:

```
Proposed Solution 2 - Doc2Vec Test metrics:
Accuracy - 0.5944
f1 score - 0.5562
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        82
           1       0.59      0.71      0.64       303
           2       0.60      0.64      0.62       298

    accuracy                           0.59       683
   macro avg       0.40      0.45      0.42       683
weighted avg       0.52      0.59      0.56       683
```

Performance of the proposed model is definitely better than the predecessor Word2Vec, but it was the same as TF-IDF base line model. I believe that the Proposed model will yield better results by tuning the Doc2Vec hyperparameters vector size, min count and epochs for training.

CSC791 NLP P1: Sentimental Analysis

Shreyas Chikkballapur Muralidhara - schikkb

IMPLEMENTATION STEPS:

Base line 1: Word2Vec

1. Tokenize sentences into word tokens using simple tokenize:

```
token_sentences = []
for sentence in input_df['sentence']:
    token_sentences.append(list(gensim.utils.simple_tokenize(sentence)))
input_df['word_tokens'] = token_sentences
```

2. Compute word vectors for training and validation sets, using word2Vec-skip gram, for each word or token average these word vectors to generate the vectors for sentence.

```
# Compute word vectors for the train dataset
w2vmodel = Word2Vec(X_train,size=100,window=5,min_count=4,sg=1)
print('Details of the model generated -',w2vmodel)

#Average the word vectors generated for sentence vector
X_train_vec = []
for sent_token in X_train:
    #count the words for which the vectors were generated
    ctr = 0
    word_token = np.empty((100),int)
    for word in sent_token:
      if word in w2vmodel.wv.vocab:
        word_token = word_token + np.array(w2vmodel[word])
        ctr = ctr + 1

    word_token = word_token/ctr
    X_train_vec.append(word_token)
```

3. Train and validate using decision tree classifier for each sentence in class 0, 1 & 2.

```
# Fit the model for the classifier
clf_word2vec = DecisionTreeClassifier().fit(X_train_vec, Y_train)

# Predicting the class labels for validation data
Y_val_pred = clf_word2vec.predict(X_val_vec)
```

```
 # Predicting the class labels for validation data
 Y_test_pred = clf_word2vec.predict(test_vec)
```

Base line 2: TF-IDF

1. Convert the collection of sentences to matrix of token counts using count vectorizer.

```
# Step 1 - Generate the token matrix
count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(X_train)
X_val_counts = count_vect.transform(X_val)
```

2. Transform the matrix into normalized TF or TF-IDF form.

```
# Step 2 - Transform the count matrix to TF-IDF
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
X_val_tfidf = tfidf_transformer.transform(X_val_counts)
```

3. Train and validate using the Naïve Bayes classifier.

```
# Step 3 - Train and validate the model using Naive Bayes Classifier
clf = MultinomialNB().fit(X_train_tfidf, Y_train)
Y_val_pred = clf.predict(X_val_tfidf)
```

```
X_test_counts = count_vect.transform(test_df['sentence'])
X_test_tfidf = tfidf_transformer.transform(X_test_counts)

Y_test_pred = clf.predict(X_test_tfidf)
```

Proposed Solution - Doc2Vec with pre-process Stemming and Lemmatization

1. Pre-processing the input data includes removing punctuations, perform stemming using Porter stemmer and Word Net Lemmatization.

```
# Perform text Preprocessing - Stemming using Porter Stemmer
preprocessed_sentence = []
for sentence in input_df['sentence']:
    lemma_sentence = []
    # Removing punctuations from the sentence
    sentence = re.sub(r'[^0-9A-Za-z]+', ' ', sentence)
    for word in word_tokenize(sentence):
        # Replace the word with stem word
        stem_word = PorterStemmer().stem(word)

        #Performing lemmmatization on stem words
        lemma_sentence.append(WordNetLemmatizer().lemmatize(stem_word))

    preprocessed_sentence.append(lemma_sentence)
```

2. Represent each sentence as tagged document containing list of words and associated tags.

```
# Step 1 - Generating tagged documents with list of words and their associated tags
X_train_tagged = [TaggedDocument(d, [i]) for i, d in enumerate(X_train)]
X_val_tagged = [TaggedDocument(d, [i]) for i, d in enumerate(X_val)]
```

3. Define the model and build the vocab using the training set.

```
# Step 2 - Define the model and build the vocab using training set
d2vmodel = Doc2Vec(min_count =1,vector_size=100, epochs=20)
d2vmodel.build_vocab(X_train_tagged)
```

4. Generate the document vectors for training, validation and testing dataset.

```
# Step 3 - Generate the document vectors for training and validation dataset
X_train_doc_vectors = []
for d in X_train_tagged:
    X_train_doc_vectors.append(d2vmodel.infer_vector(d.words))

X_train_doc_vectors = pd.DataFrame(X_train_doc_vectors)
```

5. Predicting the class labels using The Support Vector Classifier.

```
# Fit the model for the classifier
clf_word2vec = SVC().fit(X_train_doc_vectors, Y_train)

# Predicting the class labels for validation data
Y_val_pred = clf_word2vec.predict(X_val_doc_vectors)
```

```
# Predicting the class labels for validation data
Y_val_pred = clf_word2vec.predict(X_test_doc_vectors)
```