E/14/240
Pavithya M.B.D.

CO544 - Machine Learning and Data Mining
Assignment - Report

Document Representation

We were given a text file with data set for training purpose of the algorithm. First task was to represent the document in such a way that classifier understands data. This is also called preprocessing.

For that,

- First the given text file was converted to a csv format manually by replacing all the commas using spaces and then replacing tabs using commas.
- Then read the data/text from the given file.
- As the next step texts were tokenized and all unwanted characters such as punctuations, numerics, special characters etc were removed - cleaning.
- Then all the words were converted to lowercase.
- Stop words were removed and stemming was done at the same time.
- Then the preprocessed data was saved in separate text files for each text. Text files were placed in two folders according to their class values.
- For files where there were no text,
    - ➔ Instance was dropped when training(Handling missing values in training set)
    - ➔ Heading was assigned as the text when testing/predicting.(Handling missing values in test set)

Cleaning

Removing unwanted characters and changing in to one case. Below piece of code is used to achieve that.

```
def clean_str(istring):
    istring = re.sub(r"\n", "", istring)
    istring = re.sub(r"\r", "", istring)
    istring = re.sub(r"[0-9]", "digit", istring)
    istring = re.sub(r"\'", "", istring)
    istring = re.sub(r"\"", "", istring)
    istring = re.sub('['+string.punctuation+']', ' ', istring)
    return istring.strip().lower()
```

Stemming

Stemming is the process of reducing inflected (or sometimes derived) words to their word stem/ base. For example stemming algorithm should reduce the words fishing, fished, and fisher to the stem fish.
There were many stemmers available to achieve above task.
Such as,

- Snowball Stemmer
- Porter Stemmer

In my implementation I used ***porter stemmer*** by including below piece of code in the implementation.

```
from nltk.stem import PorterStemmer
ps = PorterStemmer()
w = ps.stem(w)
```

Stop words

If we want to do it manually we can import stop words from ntlk.corpus package. The words in text which were preprocessed/cleaned, were written to the file only if they are not stop words. (I have used CountVectorizer for bag of words representation it does stop word removing.)

```
from nltk.corpus import stopwords
if w not in stop_words:
        FILE.write("%s " %w)
```

Bag-of-Words representation

Text Analysis is a major application field for ML algorithms. However most algorithms cannot understand a sequence of symbols/text(raw data), as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length. In order to address this, scikit-learn provides utilities for the most common ways to extract numerical features from text content,

- **tokenizing** strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators.
- **counting** the occurrences of tokens in each document.

- **normalizing** and weighting with diminishing importance tokens that occur in the majority of samples / documents.

This can also be referred/called as **bag-of-words** representation, which is one of the feature extraction model for text that is used to achieve above requirement. The bag of words representation ignores grammar and order of words and create a vector of numerical values out of the given text by considering word frequencies. In my implementation I have initialized a CountVectorizer to use NLTK's tokenizer instead of its default one (which ignores punctuation and stopwords), then initialized a data_vector object, and turned text data into a vector.

```
from sklearn.feature_extraction.text import CountVectorizer
import nltk
data_vec = CountVectorizer(min_df=2, tokenizer=nltk.word_tokenize)
data_counts = data_vec.fit_transform(text_data.data)
```

In the obtained vector each **individual token occurrence frequency** (normalized or not) is treated as a **feature**.

Feature selection

Word counts are a good starting point, but are very basic. One issue with simple counts is that some words like "*the*" will appear many times and their large counts will not be very meaningful in the encoded vectors. An alternative is to calculate word frequencies, and by far the most popular method is called **TF-IDF**. This is an acronym than stands for "*Term Frequency – Inverse Document*" Frequency which are the components of the resulting scores assigned to each word.

- **Term Frequency**: This summarizes how often a given word appears within a document.
- **Inverse Document Frequency**: This downscales words that appear a lot across documents.

TF-IDF are word frequency scores that try to highlight words that are more interesting, for example words that are frequent in a document but not across all the documents.

The **TfidfVectorizer** will tokenize documents, learn the vocabulary and inverse document frequency weightings, and allow you to encode new documents. Alternately, if you already have a learned CountVectorizer, you can use it with a **TfidfTransformer** to just calculate the inverse document frequencies and start encoding documents. The same create, fit, and transform process is used as with the CountVectorizer.In my implementation I have created bag of words representation using CountVectorizer and then obtained the most important features by using applying **TfidfTransformer** to the obtained vectors.

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer()
data_tfidf = tfidf_transformer.fit_transform(data_counts)
#data_counts is the vector created using count vectorizer
```

Results for different classifiers

DecisionTreeClassifier

- Confusion matrix for 0.33 test set split
  [34 2]
  [0 30]
  Accuracy: 0.97

- 10CV Accuracy: 0.95 (+/- 0.10)

NeighborsClassifier

- Confusion matrix for 0.33 test set split
  [34 2]
  [5 25]
  Accuracy: 0.89

- 10CV Accuracy: 0.91 (+/- 0.10)

MultinomialNB

- Confusion matrix for 0.33 test set split
  [35 1]
  [0 30]
  Accuracy: 0.98

- 10CV Accuracy: 0.96 (+/- 0.08)

SVM

- Confusion matrix for 0.33 test set split
  [35 1]
  [0 30]
  Accuracy: 0.98

- 10CV Accuracy: 0.98 (+/- 0.07)

By considering both accuracies I selected SVM as the classifier for my implementation. Multinomial Naive bayes was also gave better accuracies.

Accuracy of submitted test set predictions = 97%

Selected classifier algorithm

“Support Vector Machine” (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyperplane that differentiate the two classes very well.

```
clf = svm.SVC(kernel='linear', C=1,gamma=1).fit(data_tfidf, text_data.target)
df_test = pd.read_csv("testsetwithoutlabels.csv", names = ['heading','date','text'])

for i in range(df_test.shape[0]):
    if df_test.iloc[i,2] == "(NO TEXT)":
        df_test.iloc[i,2] = df_test.iloc[i,0]

df_test_new_counts = data_vec.transform(df_test["text"])
df_test_new_tfidf = tfidf_transformer.transform(df_test_new_counts)
pred = clf.predict(df_test_new_tfidf)
```

Code

e14240assignment.py