Raaghav_94_NLP2_Sentiment_Analysis

January 26, 2024

1 Sentiment Analysis

1.1 1. Explain the pipeline for developing sentiment analysis task.

Sentiment Analysis is the process of determining whether a piece of text is positive, negative or neutral. The pipeline for this task is as follows:

- **Loading the Data**
  - Selecting and Downloading the appropriate dataset.
- **Data Cleaning and Preprocessing**
  - Retain only the text and their sentiments.
  - Remove Stopwords and Punctuations.
  - Lowercase the text.
  - Tokenize the dataset.
  - Lemmatize the words.
- **Feature Representation**
  - Vectorize the text using BoW, TF-IDF or word embeddings like Word2Vec.
- **Model Building**
  - Classifier such as Logistic Regression, SVM, Decision Tree, Naive Bayes etc.
- **Model Evaluation**
  - A test set is preprocessed with the same techniques and input to the model to classify and evaluate the performance.

1.2 2. Perform cleaning and preprocessing of text.

[ ]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from gensim.parsing.preprocessing import remove_stopwords
from gensim.utils import simple_preprocess
from nltk.stem.wordnet import WordNetLemmatizer
```

1.2.1 Load Dataset

[ ]:
```
data = pd.read_csv("archive/train.csv", encoding='unicode_escape')
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 27481 entries, 0 to 27480
```

```
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   textID            27481 non-null  object
 1   text              27480 non-null  object
 2   selected_text     27480 non-null  object
 3   sentiment         27481 non-null  object
 4   Time of Tweet     27481 non-null  object
 5   Age of User       27481 non-null  object
 6   Country           27481 non-null  object
 7   Population -2020  27481 non-null  int64
 8   Land Area (Km²)   27481 non-null  float64
 9   Density (P/Km²)   27481 non-null  int64
dtypes: float64(1), int64(2), object(7)
memory usage: 2.1+ MB
```

[ ]:
```
data = data[['selected_text', 'sentiment']]
data.columns = ['sentences', 'sentiment']
data.dropna(inplace=True)
data.head(3)
```

[ ]:
```
                            sentences sentiment
0  I`d have responded, if I were going   neutral
1                            Sooo SAD  negative
2                         bullying me  negative
```

[ ]:
```
le = LabelEncoder()
data['sentiment'] = le.fit_transform(data['sentiment'])
data.head(3)
```

[ ]:
```
                            sentences  sentiment
0  I`d have responded, if I were going          1
1                            Sooo SAD          0
2                         bullying me          0
```

1.3 Preprocessing the Data

[ ]:
```
lemma = WordNetLemmatizer()

def preprocess(text):
    text = simple_preprocess(remove_stopwords(text), deacc=True)
    return [lemma.lemmatize(str(word)) for word in text]
```

[ ]:
```
data['sentences'] = data['sentences'].apply(preprocess)
sentences = data["sentences"].values.tolist()
```

[ ]:
```
sentences_list = [" ".join(i) for i in sentences]
```

1.4 3.Generate representations using

[ ]:
```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from gensim.models.word2vec import Word2Vec
```

1.4.1 Bag of Words

[ ]:
```
cv = CountVectorizer()
count_matrix = cv.fit_transform(sentences_list)
count_array = count_matrix.toarray()
```

1.4.2 TF-IDF

[ ]:
```
tfidf = TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(sentences_list)
tfidf_array = tfidf_matrix.toarray()
```

1.4.3 Word2Vec - Continuous Bag of Words

[ ]:
```
cbow = Word2Vec(sentences, vector_size=100, window=5, min_count=2, sg=0)
```

[ ]:
```
vocab = cbow.wv.index_to_key

def get_mean_vector(model, sentence):
    words = [word for word in sentence if word in vocab]
    if len(words) >= 1:
        return np.mean(model.wv[words], axis=0)
    return np.zeros((100,))
```

[ ]:
```
cbow_array = []
for sentence in sentences:
    mean_vec = get_mean_vector(cbow, sentence)
    cbow_array.append(mean_vec)

cbow_array = np.array(cbow_array)
```

1.4.4 Word2Vec - Skip-gram

[ ]:
```
sg = Word2Vec(sentences, vector_size=100, window=5, min_count=2, sg=1)
```

[ ]:
```
vocab = sg.wv.index_to_key

def get_mean_vector(model, sentence):
    words = [word for word in sentence if word in vocab]
    if len(words) >= 1:
        return np.mean(model.wv[words], axis=0)
    return np.zeros((100,))
```

[ ]:
```
sg_array = []
for sentence in sentences:
    sg_array.append(get_mean_vector(sg, sentence))

sg_array = np.array(sg_array)
```

1.5 4. Classify the data using appropriate machine learning techniques to generate labels.

[ ]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

y_train = data['sentiment'].values
```

[ ]:
```
bow_model = MultinomialNB()
bow_model.fit(count_array, y_train)
bow_model.score(count_array, y_train)
```

[ ]: 0.8568413391557497

[ ]:
```
tfidf_model = MultinomialNB()
tfidf_model.fit(tfidf_array, y_train)
tfidf_model.score(tfidf_array, y_train)
```

[ ]: 0.8592430858806405

[ ]:
```
cbow_model = DecisionTreeClassifier()
cbow_model.fit(cbow_array, y_train)
cbow_model.score(cbow_array, y_train)
```

[ ]: 0.9702328966521107

[ ]:
```
sg_model = DecisionTreeClassifier()
sg_model.fit(sg_array, y_train)
sg_model.score(sg_array, y_train)
```

[ ]: 0.9702328966521107

1.6 5. Analyze the labels and explain the impact of embedding techniques in misclassification.

[ ]:
```
x_test = ["What is not to like about this product.",
"Not bad.",
"Not an issue.",
"Not buggy.",
"Not happy.",
"Not user-friendly.",
```

```
"Not good.",
"Is it any good?",
"I do not dislike horror movies.",
"Disliking horror movies is not uncommon.",
"Sometimes I really hate the show.",
"I love having to wait two months for the next series to come out!",
"The final episode was surprising with a terrible twist at the end.",
"The film was easy to watch but I would not recommend it to my friends.",
"I LOL'd at the end of the cake scene."]

y_test = [2, 1, 1, 1, 0, 0, 0, 1, 2, 0, 0, 2, 0, 0, 1]
sentiment = {0:"Negative", 1:"Neutral", 2:"Positive"}
```

[ ]:
```
bow_test = [' '.join(preprocess(sentence)) for sentence in x_test]
bow_test = cv.transform(bow_test).toarray()

tfidf_test = [' '.join(preprocess(sentence)) for sentence in x_test]
tfidf_test = tfidf.transform(tfidf_test).toarray()

cbow_test_embeds = [(preprocess(sentence)) for sentence in x_test]
cbow_test = []
for sentence in cbow_test_embeds:
    cbow_test.append(get_mean_vector(cbow, sentence))
cbow_test = np.array(cbow_test)

sg_test_embeds = [(preprocess(sentence)) for sentence in x_test]
sg_test = []
for sentence in sg_test_embeds:
    sg_test.append(get_mean_vector(sg, sentence))
sg_test = np.array(sg_test)
```

[ ]:
```
y_bow = bow_model.predict(bow_test)
print([sentiment[i] for i in y_bow])
print(classification_report(y_test, y_bow))
```

```
['Neutral', 'Negative', 'Negative', 'Neutral', 'Positive', 'Negative',
'Negative', 'Positive', 'Neutral', 'Negative', 'Negative', 'Neutral', 'Neutral',
'Neutral', 'Neutral']
              precision    recall  f1-score   support

           0       0.67      0.57      0.62         7
           1       0.29      0.40      0.33         5
           2       0.00      0.00      0.00         3

    accuracy                           0.40        15
   macro avg       0.32      0.32      0.32        15
weighted avg       0.41      0.40      0.40        15
```

[ ]:
```
y_tfidf = tfidf_model.predict(tfidf_test)
print([sentiment[i] for i in y_tfidf])
print(classification_report(y_test, y_tfidf))
```

```
['Neutral', 'Negative', 'Negative', 'Neutral', 'Positive', 'Neutral',
'Negative', 'Neutral', 'Neutral', 'Negative', 'Negative', 'Neutral', 'Neutral',
'Neutral', 'Neutral']
              precision    recall  f1-score   support

           0       0.60      0.43      0.50         7
           1       0.33      0.60      0.43         5
           2       0.00      0.00      0.00         3

    accuracy                           0.40        15
   macro avg       0.31      0.34      0.31        15
weighted avg       0.39      0.40      0.38        15
```

[ ]:
```
y_cbow = cbow_model.predict(cbow_test)
print([sentiment[i] for i in y_cbow])
print(classification_report(y_test, y_cbow))
```

```
['Neutral', 'Negative', 'Positive', 'Negative', 'Negative', 'Negative',
'Negative', 'Negative', 'Positive', 'Negative', 'Neutral', 'Neutral', 'Neutral',
'Neutral', 'Negative']
              precision    recall  f1-score   support

           0       0.50      0.57      0.53         7
           1       0.00      0.00      0.00         5
           2       0.50      0.33      0.40         3

    accuracy                           0.33        15
   macro avg       0.33      0.30      0.31        15
weighted avg       0.33      0.33      0.33        15
```

[ ]:
```
y_sg = sg_model.predict(sg_test)
print([sentiment[i] for i in y_sg])
print(classification_report(y_test, y_sg))
```

```
['Negative', 'Negative', 'Negative', 'Negative', 'Negative', 'Negative',
'Negative', 'Positive', 'Neutral', 'Neutral', 'Negative', 'Neutral', 'Positive',
'Neutral', 'Neutral']
              precision    recall  f1-score   support

           0       0.50      0.57      0.53         7
           1       0.20      0.20      0.20         5
           2       0.00      0.00      0.00         3
```

```
    accuracy                           0.33        15
   macro avg       0.23      0.26      0.24        15
weighted avg       0.30      0.33      0.32        15
```

1.7 6. Discuss the limitations of each embedding technique and explain the techniques that rectify it.

- BOW
  - Sparse Representation.
  - Word Order is not considered.
  - Does not capture semantic meaning of the text.
  - Computationally Intensive.
- TF-IDF
  - Sparse Representation.
  - Does not capture semantic meaning of the text.
  - Computationally Intensive.
- Word2Vec
  - Semantic and Dense Representation.
  - CBow: Faster and Better Representation for Frequent Words.
  - SkipGram: Works well with small amount of data and Represent Rare words well.

All these still face Out-Of-Vocabulary (OOV) problem that can be resolved by using FastText which uses N-grams of the words.