Portfolio Assignment: Text Classification

Cady Baltz (cmb180010)

4/2/2023

Importing a text classification data set

```
# necessary imports for data processing
import pandas as pd
import seaborn as sb
```

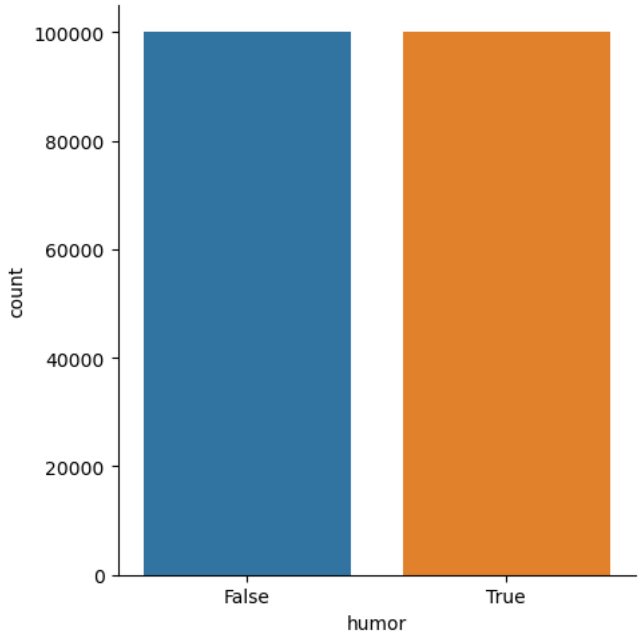
```
# using a humor text dataset from Kaggle
# source: https://www.kaggle.com/datasets/deepcontractor/200k-short-texts-for-humor-detection

input_filename = '/kaggle/input/200k-short-texts-for-humor-detection/dataset.csv'
df = pd.read_csv(input_filename)
```

Target class distribution

```
# create a graph showing the distribution of target classes in the dataset
sb.catplot(x="humor", kind='count', data=df)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f0d1f3c3ad0>
```



Data set description:

This dataset contains 200,000 instances of short pieces of text, as a well as a boolean label indicating whether or not the text is humorous. As you can see in the graph above, this dataset is evenly divided between the two target classes. The ML classification model should be able to predict whether or not a piece of short text is humorous.

Divide into a train/test dataset

```
# split the data into train and test sets
from sklearn.model_selection import train_test_split

# in this case, the predictor is the text and the target is whether or not the text is humorous
X = df.text
```

```
y = df.humor

# have 20% of the data go to the test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
```

Pre-Processing

```
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer

# remove stopwords during pre-processing
stopwords = set(stopwords.words('english'))

# create a tf-idf representation of the data so it can be represented with discrete values
vectorizer = TfidfVectorizer(stop_words=stopwords)

# apply the tfidf vectorizer to fit and transform the training data
X_train_tfidf = vectorizer.fit_transform(X_train)

# only transform the test data
X_test_tfidf = vectorizer.transform(X_test)

# also create a binary tf-idf representation for comparison
vectorizer = TfidfVectorizer(stop_words=stopwords, binary=True)

# apply the tfidf vectorizer to fit and transform the training data
X_train_tfidf_binary = vectorizer.fit_transform(X_train)

# only transform the test data
X_test_tfidf_binary = vectorizer.transform(X_test)
```

Naive Bayes Approach

Training the model

```
# we can use a multinomial Bayes classifier using the discrete tf-idf representation
from sklearn.naive_bayes import MultinomialNB

# fit the training data using the classifier with default settings
nb = MultinomialNB()
nb.fit(X_train_tfidf, y_train)
```

```
MultinomialNB()
```

Evaluating the model

```
# make predictions on the test data
pred = nb.predict(X_test_tfidf)

# print the confusion matrix for the test data
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, pred)
```

```
array([[17994,  2073],
       [ 1955, 17978]])
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

print('accuracy score: ', accuracy_score(y_test, pred))

print('\nprecision score (not humorous): ', precision_score(y_test, pred, pos_label=0))
print('precision score (humorous): ', precision_score(y_test, pred))

print('\nrecall score: (not humorous)', recall_score(y_test, pred, pos_label=0))
print('recall score: (humorous)', recall_score(y_test, pred))

print('\nf1 score: ', f1_score(y_test, pred))
```

```
accuracy score:  0.8993

precision score (not humorous):  0.9020001002556519
precision score (humorous):  0.8966136352301631

recall score: (not humorous) 0.8966960681716251
recall score: (humorous) 0.9019214368133246

f1 score:  0.8992597038815525
```

Now, we will try a binary Naive Bayes approach to compare the performance with the multinomial approach.

```
from sklearn.naive_bayes import BernoulliNB

nb_2 = BernoulliNB()
nb_2.fit(X_train_tfidf_binary, y_train)
```

```
BernoulliNB()
```

```
# make predictions on the test data
pred = nb_2.predict(X_test_tfidf_binary)

# print confusion matrix
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, pred)
```

```
array([[17732,  2335],
       [ 1632, 18301]])
```

```
# print various metrics for comparison
print('accuracy score: ', accuracy_score(y_test, pred))

print('\nprecision score (not humorous): ', precision_score(y_test, pred, pos_label=0))
print('precision score (humorous): ', precision_score(y_test, pred))

print('\nrecall score: (not humorous)', recall_score(y_test, pred, pos_label=0))
print('recall score: (humorous)', recall_score(y_test, pred))

print('\nf1 score: ', f1_score(y_test, pred))
```

```
accuracy score:  0.900825

precision score (not humorous):  0.9157198925841769
precision score (humorous):  0.8868482264004652

recall score: (not humorous) 0.8836398066477301
recall score: (humorous) 0.9181257211659057

f1 score:  0.902215977716976
```

The binary approach had a better performance. Therefore, for the rest of my analyses, I will use the binary tf-idf representation.

Logistic Regression

```
# imports needed for logistic regression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
```

```
# create a logistic regression classifier
classifier = LogisticRegression(solver='lbfgs', class_weight='balanced', max_iter=400)

# fit the classifier to the training data
classifier.fit(X_train_tfidf_binary, y_train)
```

```
LogisticRegression(class_weight='balanced', max_iter=400)
```

```
# evaluate the classifier by attempting to classify the test data
pred = classifier.predict(X_test_tfidf_binary)

# print confusion matrix
print(confusion_matrix(y_test, pred))
```

```
# print metrics to compare with other models
print('\naccuracy score: ', accuracy_score(y_test, pred))

print('\nprecision score (not humorous): ', precision_score(y_test, pred, pos_label=0))
print('precision score (humorous): ', precision_score(y_test, pred))

print('\nrecall score: (not humorous)', recall_score(y_test, pred, pos_label=0))
print('recall score: (humorous)', recall_score(y_test, pred))

print('\nf1 score: ', f1_score(y_test, pred))
```

```
[[18143  1924]
 [ 1971 17962]]

accuracy score:  0.902625

precision score (not humorous):  0.9020085512578304
precision score (humorous):  0.9032485165443025

recall score: (not humorous) 0.9041211940000997
recall score: (humorous) 0.9011187478051472

f1 score:  0.9021823752479972
```

Neural Networks

```
# first, set up a classifier like we did in the example code from GitHub
from sklearn.neural_network import MLPClassifier

classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(15, 2), random_state=1)

# attempt to fit the training data with this classifier
classifier.fit(X_train_tfidf_binary, y_train)
```

```
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(15, 2), random_state=1,
               solver='lbfgs')
```

```
# observe a very low accuracy with this classifier configuration
pred = classifier.predict(X_test_tfidf_binary)

print('accuracy score: ', accuracy_score(y_test, pred))
```

```
accuracy score:  0.498325
```

After observing this low accuracy using the MLPClassifier initialization code from class, I experimented with a different number of nodes in the second hidden layer and found that it greatly increased the accuracy.

```
# now, adjust the number of nodes in the second hidden layer
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(15, 10), random_state=1)
classifier.fit(X_train_tfidf_binary, y_train)
```

```
/opt/conda/lib/python3.7/site-packages/sklearn/neural_network/_multilayer_perceptron.py:549: ConvergenceWarning: lbfgs failed
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(15, 10), random_state=1,
               solver='lbfgs')
```

```
pred = classifier.predict(X_test_tfidf_binary)

# print confusion matrix
print(confusion_matrix(y_test, pred))

print('\naccuracy score: ', accuracy_score(y_test, pred))

print('\nprecision score (not humorous): ', precision_score(y_test, pred, pos_label=0))
print('precision score (humorous): ', precision_score(y_test, pred))

print('\nrecall score: (not humorous)', recall_score(y_test, pred, pos_label=0))
```

```
print('recall score: (humorous)', recall_score(y_test, pred))

print('\nf1 score: ', f1_score(y_test, pred))
```

```
[[18078  1989]
 [ 1932 18001]]

accuracy score:  0.901975

precision score (not humorous):  0.9034482758620690
precision score (humorous):  0.9005002501250625

recall score: (not humorous) 0.9008820451487517
recall score: (humorous) 0.9030753022625796

f1 score:  0.9017859379305161
```

Analysis

Overall, the three ML algorithms produced similar accuracies (in the range of 89% to 90%). The algorithm with the highest accuracy was logistic regression at 90.26%.

However, the models had different relative performances based on the scoring formula, as a result of different confusion matrices. While this is subjective and would depend on the exact context of the implementation of this data, I would place a higher cost on false negatives in classifying this dataset. This is because if you think a piece of text is serious when it is meant to be a joke, you may react negatively to the information. If we are placing a higher cost on false negatives, then we should place greater emphasis on the recall score for humorous text. In this case, the Naive Bayes Bernoulli approach had the best performance at 91.81%.

Additionally, if we decided to evaluate based on the F1 score to balance precision and recall, the Naive Bayes approach would be highest again, with an F1 score of 90.22%.

Finally, if we wanted to place a higher cost on false positives, then logistic regression would have the best performance (as it had greatest precision when analyzing humorous text).

I believe that the limitation of my ML approach was how I discretized the data. In this assignment, I simply used a TF-IDF approach to convert the text into a numerical representation. With the addition of more complex features, I likely could have achieved a higher accuracy. When browsing Kaggle, I saw that others were able to achieve a neural net accuracy of over 95% with the same dataset by applying more pre-processing steps, calculating additional features, and fine-tuning the model's parameters.

If I were to continue working on this project, I would attempt to improve on the neural net implementation by experimenting with different numbers of hidden layers and solver algorithms, as well as additional pre-processing and improved feature selection.