Homework 3

Author: Nicholas M. Synovic

Table of Contents

About

The homework assignment description can be found in hw3.pdf. The `hw3.py` script is the executable code to run to generate results.

The dataset used for this assignment was downloaded from here.

Dependencies

To run this code, you will need:

- `Python 3.10`
- `scikit-learn`
- `numpy`
  - These can be installed by running `pip install -r requirements.txt`

How To Run

- `python3.10 hw3.py`

NOTE: This code takes a long time to complete. There are various places within the code that can be commented out in order to speed up the process. Please review the `main()` function to identify which portions of the code to comment out.

Methodology

I utilized `scikit-learn`'s SVM implementation for my analysis of the dataset. I implemented a term-document frequency analysis for my custom vectorizor, and utilized the `CountVectorizer` class from `scikit-learn`.

Data Preprocessing

Stop words **were not removed from datasets** and all documents were made lowercase. Additionally, all non alphabetical charachters were removed from the dataset prior to usage.

Data Splitting

Data was split into *70-15-15* splits where:

- 70% of the data was used for training
- 15% of the data was used for development evaluation
- 15% of the data was used for model evaluation

Data was split using the same random seed (*42*) and with `scikit-learn`'s `train_test_split()` function.

Custom Vectorizer

I implemented a vectorizer based off of term-document frequency. This vectorizers counts the number of times a word appears in a given document. The data structure of the vectorizers output is a **sparse** `numpy ndarray` consisting of *13515* columns, with *13514* containing data (where each column is the dictionary of words from the training dataset) and the first column (index *0*) being the label of the document, and n rows where n is the number of documents within the training | development | testing dataset.

The code for my vectorizer is below:

```
from typing import List
from collections import Counter


def termDocumentFrequency(
    data: List[str], wordSet: set[str], label: int
) -> List[List[int]]:
    tdfDict: dict[str, List[int]] = {document: [label] for document in data}
    tdfList: List[List[str]] = []

    for document in tdfDict:
        c: Counter = Counter(document.split(" "))
        for word in wordSet:
            tdfDict[document].append(c[word])
```

```
        tdfList.append(tdfDict[document])

    return tdfList
```

Preprocessing Data for Training

Due to the high dimensionality of the data, I opted to find ways to reduce the dimensionality of the data prior to training. To do this, I utilized principle component analysis (PCA) using `sciki-learn`'s `PCA` class. I found that reducing the dataset from *13515* to *100* dimensions resulted in sufficient accuracy results.

Prior to performing a PCA, I standardized the data using `sciki-learn`'s `StandardScaler` class.

The pipeline function for this data preprocessing is below:

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler


def scaleData(
    fitData: ndarray, transformData: ndarray, numberOfComponents: int = 10
) -> ndarray:
    scaler: StandardScaler = StandardScaler()
    scaler.fit(fitData)
    scaledData: ndarray = scaler.transform(transformData)
    pca: PCA = PCA(n_components=numberOfComponents)
    pca.fit(X=scaledData)

    return pca.transform(X=transformData)
```

NOTE: This was done on my custom vectorized data. This was not done for the data processed using `scikit-learn`'s `CountVectorizer` class.

Optimal SVM Model Selection

To find the optimal SVM model to classify the data, I opted to use grid searching to find the best model. To do this, I utilized `scikit-learn`'s `GridSearchCV` method and created a pipeline to train and test many different models.

The pipeline code is below:

```
pipeline: Pipeline = make_pipeline(SVC(random_state=42))

parameterRange: List[float] = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
parameterGrid: List[dict] = [
    {"svc__C": parameterRange, "svc__kernel": ["linear"]},
    {
        "svc__C": parameterRange,
```

```
        "svc__gamma": parameterRange,
        "svc__kernel": ["rbf"],
    },
]
gridSearch: GridSearchCV = GridSearchCV(
    estimator=pipeline,
    param_grid=parameterGrid,
    scoring="accuracy",
    cv=10,
    refit=True,
    n_jobs=-1,
)
gridSearch.fit(X=trainingData, y=trainingLabels)
bestModel = gridSearch.best_estimator_
```

The best model had the following hyper-parameters:

- `C`: 100.0
- `gamma`: 0.1
- `kernel`: rbf (this was the only kernel tested)
- `random_state`: 42

The best model achieved an accuracy of *65.11%* on the training dataset and *49.41%* on the development dataset.

Best Model Test Dataset Results

The test dataset (that was vectorized using my custom vectorizer), when classified using my best model, achieved an accuracy of *50.13%*.

A table of the results can be found below:

	Development Dataset	**Testing Dataset**
Custom Vectorizer	49.41%	50.13%

Best Model Results with CountVectorizer

Using `scikit-learn`'s `CountVectorizer` class, I recreated the data sets and trained a new model using the same hyper-parameters.

The table below compares the results between my custom vectorizer and `CountVectorizer`:

	Development Dataset	**Testing Dataset**
Custom Vectorizer	49.41%	50.13%
CountVectorizer	73.24%	70.88%