

SENTIMENT CLASSIFICATION

Priyanka Gopi
Dept. of Computer Science, University of Central Florida
priyankagopi@knights.ucf.edu

I. OBJECTIVE

To develop a classifier to recognize the positive and negative reviews/sentiments of a book and ultimately perform the modelling on 300d Glove embeddings using the mean-pooling method.

II. METHODOLOGY

The assignment was begun with importing necessary libraries like Pandas, Numpy, SKLearn, Codecs, MLPClassifier, and for all the required evaluation metrics like F1, Accuracy, and Recall - Precision scores.

```
[2] import pandas as pd
import numpy as np
import sklearn
import codecs
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
```

Fig 1 – Imported Libraries

Following this, the given snippet in “sentiment_reader.py” was run, which consists of two functions and a class named *SentimentCorpus*.

Two files, namely, Positive.review and Negative.review are being read by the function *build_dicts*, which builds a feature-dictionary, “feat_dict” based on the word frequency in the files and also eliminating all the words from dataset which occurs less than 5 times.

```
def build_dicts():
    """
    builds feature dictionaries
    """
    feat_counts = {}

    # build feature dictionary with counts
    nr_pos = 0
    with codecs.open("/content/drive/MyDrive/NLP/dataset/positive.review", 'r', 'utf8') as pos_file:
        for line in pos_file:
            nr_pos += 1
            toks = line.split(" ")
            for feat in toks[0:-1]:
                name, counts = feat.split(":")
                if name not in feat_counts:
                    feat_counts[name] = 0
                feat_counts[name] += int(counts)

    nr_neg = 0
    with codecs.open("/content/drive/MyDrive/NLP/dataset/negative.review", 'r', 'utf8') as neg_file:
        for line in neg_file:
            nr_neg += 1
            toks = line.split(" ")
            for feat in toks[0:-1]:
                name, counts = feat.split(":")
                if name not in feat_counts:
                    feat_counts[name] = 0
                feat_counts[name] += int(counts)
```

Fig 2 – Function build_dicts() code

```
# remove all features that occur less than 5 (threshold) times
to_remove = []
for key, value in feat_counts.items():
    if value < 5:
        to_remove.append(key)
for key in to_remove:
    del feat_counts[key]
```

Fig 3 – Removal of values with occurrences less than 5

The *split_train_dev_test* function returns the data divided into three sets: train, development (dev), and test. It accepts a feature matrix, X and its corresponding labels y, as well as a train:dev:test ratio for splitting. The function initially determines if the ratios added together are less than 1, and if they are, it prints a warning notice. As per the splitting ratio, the data is then divided into three sets. However, if the dev set's ratio is 0, we will get only train and test sets.

The class *SentimentCorpus* has functions, *build_dicts* and *split_train_dev_test* to produce a sentiment analysis dataset. When an object of the class is created, the *__init__* function is invoked, to initialize the object by creating feature dictionaries and dividing the data into train, dev, and test sets. The object contains feature matrix X, labels Y, feature dictionary, featurecounts, and train:dev:test data.

1. Data Exploration

The stats of positive and negative words in each train and test dataset are as below:

	Train	Test
Positive class	788	212
Negative class	812	188

Code output:

```
#Train data stats
pos_train = np.sum(corp.train_y == 0) # Positive words
neg_train = np.sum(corp.train_y == 1) # Negative words

#Test data stats
pos_test = np.sum(corp.test_y == 0) # Positive words
neg_test = np.sum(corp.test_y == 1) # Negative words

#Result
print(f"Count of positive samples in train data: {pos_train} \nCount of negative samples in train data: {neg_train}")
```

Count of positive samples in train data: 788
Count of negative samples in train data: 812
Count of positive samples in test data: 212
Count of negative samples in test data: 188

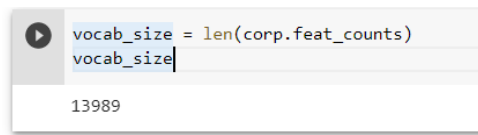
Fig 4 – Count of train/test data in each class

Below image, Fig 5, represents the dictionary representation of the words with their frequency of occurrence.

```
corpus.feats_dict
{
  'holes': 0,
  'must': 1,
  'he': 2,
  'center': 3,
  'a_lot': 4,
  'this_book': 5,
  '<num>_feet': 6,
  'put_down': 7,
  'ten': 8,
  'pressure': 9,
  'a_day': 10,
  'johnson_is': 11,
  'strange': 12,
  'explored': 13,
  'support': 14,
  'read': 15,
  ...
}
```

Fig 5 – Dictionary representation

The total **vocabulary size** is **13989**, as obtained in Fig 6.



```
vocab_size = len(corp.feab_counts)
vocab_size
```

13989

Fig 6 – Vocabulary size

2 – Modelling and Evaluation

2.1.

For the modelling step, the model Multi-layer Perceptron classifier (MLPClassifier) was used to connect to the neural network. Unlike other classification algorithms like Support Vector classifier, MLPClassifier uses an underlying Neural Network to carry out the classification step.

The below code, in Fig 7 creates a loop to train and test an MLP classifier model using various combinations of hyperparameters. The activation function (*activation_functions*) and the number of layers (*n_layers*) are the hyperparameters that will change over each loop.

The code constructs an instance of the MLPClassifier model using the specified number of hidden layers (*n*) and activation function for each combination of *n* layers and activation functions ('act' looping over logistic, tanh, relu functions). The method is then used to fit the model to the training dataset. Based on the test data, predictions are made by the model, and its accuracy, precision, recall, and f1-score are then computed.



```
# Define the hyperparameters
n_layers = [1, 2, 3]
hidden_unit_size = 100
activation_functions = ["logistic", "tanh", "relu"]

# Train and evaluate a MLPClassifier for each configuration
for n in n_layers:
    for act in activation_functions:
        model = MLPClassifier(hidden_layer_sizes=(hidden_unit_size,)*n, activation=act)
        model.fit(corpus.train_X, corpus.train_y)
        pred_y = model.predict(corpus.test_X)

        # Calculate and print the performance metrics
        acc = accuracy_score(corpus.test_y, pred_y)
        prec = precision_score(corpus.test_y, pred_y)
        rec = recall_score(corpus.test_y, pred_y)
        f1 = f1_score(corpus.test_y, pred_y)
        print(f"n_layers: {n}, activation: {act}, acc: {acc:.4f}, prec: {prec:.4f}, rec: {rec:.4f}, f1: {f1:.4f}")
```

Fig 7 – Modelling using MLPClassifier and evaluating its performance.

2.2.

The evaluation metrics require the libraries to be imported from *sklearn.metrics*, as done in the beginning.

(i) Accuracy: It is a measure of evaluation, establishes the fraction of data points that were accurately predicted. It is calculated by dividing the total amount of points by the points that were correctly obtained.

(ii) Recall: A metric that counts the fraction of all positive labels that were accurately predicted as positive. Formally, it is the ratio of genuine positive responses to all incorrect negative predictions that were mistakenly categorized as positive reactions (false negative).

(iii) Precision: The binary classifier statistic gives the measure of positive predictions that were relevantly classified.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

(1) - Precision-Recall calculation

(iv) F1-Score: Produced by computing the harmonic mean of precision and recall scores of a classifier.

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

(2) - F1 score equation

For the sentiment evaluation, feed forward neural network tuning is essential. Considering the choice of hyperparameters, such as the activation function and number of hidden layers, can have a substantial impact on how effectively a neural network model performs. Here, 9 different configurations were evaluated on the test data as shown in Fig 7. The model with hyperparameters of 3 layers and Logistic activation function showed to be the best with 86.5% accuracy, 85.7% F1 score with Precision: Recall as 0.8490:0.8670 ratio. This implies that the model was able to predict the values in the appropriate class well with a score 86.7% and the percentage of true results were returned in a score of 84.9%, which is decent.

```
[128] # Set the hyperparameters
n_layers = [1, 2, 3]
hidden_unit_size = 100
activation_functions = ["logistic", "tanh", "relu"]

# Training and evaluation of MLPClassifier model for each combination of hyperparameters
for n in n_layers:
    for act in activation_functions:
        model = MLPClassifier(hidden_layer_sizes=(hidden_unit_size,)*n, activation=act, random_state=0)
        model.fit(corp.train_X, corp.train_y)
        pred_y = model.predict(corp.test_X)

        # Calculate and print the performance metrics
        acc = accuracy_score(corp.test_y, pred_y)
        prec = precision_score(corp.test_y, pred_y)
        rec = recall_score(corp.test_y, pred_y)
        f1 = f1_score(corp.test_y, pred_y)
        print(f'n_layers: {n}, activation: {act}, acc: {acc:.4f}, prec: {prec:.4f}, rec: {rec:.4f}, f1: {f1:.4f}")

/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 1, activation: logistic, acc: 0.8600, prec: 0.8367, rec: 0.8723, f1: 0.8542
/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 1, activation: tanh, acc: 0.8500, prec: 0.8107, rec: 0.8883, f1: 0.8477
/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 1, activation: relu, acc: 0.8500, prec: 0.8200, rec: 0.8723, f1: 0.8454
/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 2, activation: logistic, acc: 0.8350, prec: 0.8020, rec: 0.8617, f1: 0.8308
/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 2, activation: tanh, acc: 0.8525, prec: 0.8308, rec: 0.8617, f1: 0.8460
/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 2, activation: relu, acc: 0.8575, prec: 0.8429, rec: 0.8564, f1: 0.8496
/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 3, activation: logistic, acc: 0.8650, prec: 0.8490, rec: 0.8670, f1: 0.8579
/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 3, activation: tanh, acc: 0.8325, prec: 0.8201, rec: 0.8245, f1: 0.8223
/usr/local/lib/python3.8/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:1109: DataConversionWarning: A column-
y = column_or_id(y, warn=True)
n_layers: 3, activation: relu, acc: 0.8575, prec: 0.8429, rec: 0.8564, f1: 0.8496

From the above evaluation metrics, it seems like the results (acc: 0.8650, prec: 0.8490, rec: 0.8670, f1: 0.8579) are good with logistic activation
function run with 3 layers.
```

Fig 8 – Modelling Results with document-word matrix

Here, 3 hidden layers may be adequate, because adding more layers would increase the possibility of overfitting. The underlying pattern in the data might have been captured in this situation with 3 hidden layers and 100 hidden units, producing good performance.

3. Modelling using Embeddings

Followed by this, word embeddings zip file was downloaded from the appropriate URL using the *wget* function, following which it was unzipped, as below.

```
[10] #Embeddings
url = "https://huggingface.co/stanfordnlp/glove/resolve/main/glove.6B.zip"
file = wget.download(url)

[11] ! unzip glove.6B.zip

Archive:  glove.6B.zip
  inflating: glove.6B.100d.txt
  inflating: glove.6B.200d.txt
  inflating: glove.6B.300d.txt
  inflating: glove.6B.50d.txt
```

Fig 9 – Read embeddings from the URL.

The below function loads GloVe word embeddings that are pre-trained already and finds any missing words in a corpus *feat_dict* that don't contain pre-trained embeddings.

```
[11] embedding_file = '/content/glove.6B.300d.txt' #new 300d embeddings file path variable
with open(embedding_file, 'r', encoding='utf-8') as f:
    embeddings = {}
    for line in f:
        values = line.split()
        words = values[0]
        embedding = np.asarray(values[1:], dtype='float32')
        embeddings[words] = embedding
        #splitting each line in 300d embeddings file
        #assigning first values which are words to array 'word'
        #values from position 1 are embeddings

    embedding_dict = {} #creating a dictionary for Embeddings
```

Fig 10 – Opening the file and splitting the words and their embeddings into separate lists.

Using a corpus's feature dictionary, "*feat_dict*", the code generates a set of words after loading the embeddings, "corp". The "embeddings" dictionary is then used to construct a new set of terms. The words from the feature dictionary that are missing from the embeddings dictionary are located using set difference, and they are saved in a variable called "*missing words*."

```
#Missing words in Embeddings not present in initial dictionary
corp_words = corp.feat_dict.keys()
glove_words = embeddings.keys()
missing_words = corp_words - glove_words
```

```
missing_words
```

```
"son's",
'are_more',
'possibly_be',
'sure_you',
'just_by',
'together_with',
"can't_believe",
'see_them',
'the_current',
'against_the',
'they_should',
'are_much',
```

Fig 11 – Missing words from 'feat_dict' dictionary

The following code generates a dictionary named "embedding dict" that will hold the embeddings for the words that were recognized as being missing in the previous code.

The code initially creates an empty list called "emb vecs" and initializes it for each missing word in the set "*missing words*." The word is then split by underscores, "*_*", if any are present

and uses the "replace" method to remove all single quotes from the word before checking to see if it appears in the pre-trained embeddings dictionary. If at least one of the words in a missing word has a pre-trained embedding, the algorithm averages the individual word embeddings to provide an embedding for the *missing word*. Finally, the embeddings are normalized by dividing it by the total number of vectors. When a word is missing and none of its constituent words have pre-trained embeddings, the code checks to see if the missing word itself is in the pre-trained embeddings dictionary. If yes, the relevant embedding is directly kept in "*embedding dict*".

To summarise, the method is generating embeddings for all the missing words by averaging the embeddings of their constituent parts, given they are present in the pre-trained embeddings dict, or by using those of the missing word itself, if present. This is followed by obtaining the train_X data for modelling.

```
[1] for word in corp_words:
    if word not in missing_words:
        embedding_dict[word] = embeddings[word] #assign the embedding from corpus if the word is present in new set of words

    else:
        emb_vecs = []
        emb = np.zeros(300) #create an array of 300 vectors
        split_words = word.split("_") #split bigrams by '_' and get unigrams

        for w in split_words:
            w = w.replace("'", "") #replace the single-quotes in unigrams with space
            if w in embeddings:
                emb_vecs.append(embeddings[w]) #if these words are present in the existing embeddings list, assign the same to it

        if len(emb_vecs) > 0:
            emb_vecs = np.array(emb_vecs) #convert the list to numpy object array
            emb = np.sum(emb_vecs, axis=0) #obtain the array of mean values of all the words
            emb = emb/len(emb_vecs) #normalization of the embeddings for new words not existing in the previous list

        embedding_dict[word] = emb #adding the new word embeddings to the overall list of Embeddings - dictionary
```

Fig 12 – Creating embeddings for new words in URL downloaded embeddings.

Another function named "*final_embedding*" is created to accept X train data as its only input, to create embedding vectors for each sample in the input training data. This function computes a weighted average of the pre-trained embedding vectors for each word that appears in the sample.

```
[146] def final_embedding(X_train_data):
    new_res_emb = [] #Create a list for final Embedding list to feed the model

    for data in X_train_data:
        list_of_emb = [embedding_dict[word] * data[corp.feats_dict[word]] for word in corp.feats_dict]
        norm_emb = sum(list_of_emb) / sum(data) #Sum by rows to obtain one final vector and normalize it

        new_res_emb.append(norm_emb)

    return np.array(new_res_emb)
```

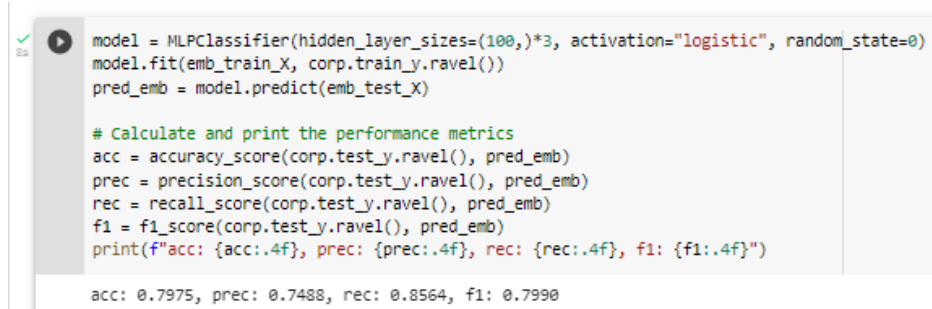
Fig 13 – Function "final_embedding" definition for obtaining final array of embeddings by mean-pooling method

Thus, the train dataset is created for the embeddings to feed to the model in the next step.

```
✓ 3m #Obtain train_X data
emb_train_X = final_embedding(corp.train_X)
emb_test_X = final_embedding(corp.test_X)
```

Fig 14 – Create test and train data for embedding.

Finally, we model using the best hyperparameters setting for good accuracy, as obtained previously which are using logistic activation function and 3 hidden layers. With this, an accuracy of 79.7%, precision - recall score of 74.8% - 85.6% and F1 score of 79.9% were obtained, as represented in the image below.



```
model = MLPClassifier(hidden_layer_sizes=(100,)*3, activation="logistic", random_state=0)
model.fit(emb_train_X, corp.train_y.ravel())
pred_emb = model.predict(emb_test_X)

# Calculate and print the performance metrics
acc = accuracy_score(corp.test_y.ravel(), pred_emb)
prec = precision_score(corp.test_y.ravel(), pred_emb)
rec = recall_score(corp.test_y.ravel(), pred_emb)
f1 = f1_score(corp.test_y.ravel(), pred_emb)
print(f"acc: {acc:.4f}, prec: {prec:.4f}, rec: {rec:.4f}, f1: {f1:.4f}")

acc: 0.7975, prec: 0.7488, rec: 0.8564, f1: 0.7990
```

Fig 15 – Model results obtained by giving embeddings as input.

As shown, the accuracy for embeddings is less than that for the document-word matrix. The obtained embeddings might not be of good quality; they might not be able to capture the key elements of the words. Here, using embeddings to represent the documents could mean losing crucial information and, as a result, having a lesser level of accuracy than the document-word matrix. We know that more data will result in good accurate generalization. The document-word matrix has received more training data than the embeddings, which explains why the former performed better. Of course, there is scope for further improvement by using different neural network for modeling and preprocessing steps as they matter in terms of the resulting metrics.