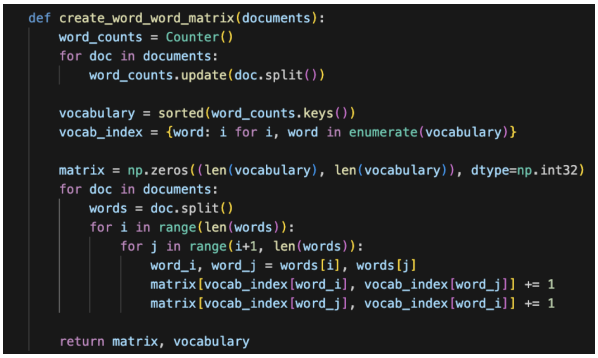
NLP Course Project 3 - Logistic Regression & Vector Semantics

In this project I have implemented tasks which are listed below:

1. Create term-document and word-word matrix. Visualize term-document and word-word matrixes in MATRIX form
2. Calculate tf-IDF features
3. Calculate PMI featires
4. Apply Logistic Regression for classification of text based on tf-IDF and PMI features and compare with results of Naive Bayes method.
5. Find 10 most similar words for the given word by using Cosine similarity.

FIrst we need to have utility functions to calcualte the word-word matrix which is very expensive in terms of the computation as we have three `for` loops. These are essenetial to calculate next feature matrices such as TF-IDF and PMI.

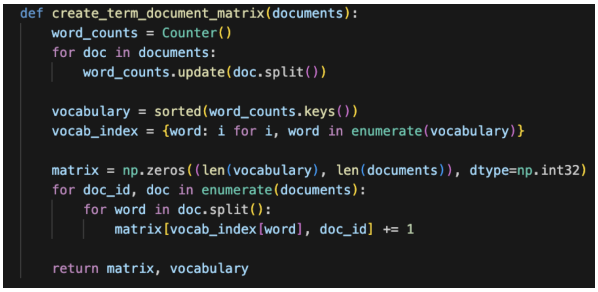


```python
def create_word_word_matrix(documents):
    word_counts = Counter()
    for doc in documents:
        word_counts.update(doc.split())

    vocabulary = sorted(word_counts.keys())
    vocab_index = {word: i for i, word in enumerate(vocabulary)}

    matrix = np.zeros((len(vocabulary), len(vocabulary)), dtype=np.int32)
    for doc in documents:
        words = doc.split()
        for i in range(len(words)):
            for j in range(i+1, len(words)):
                word_i, word_j = words[i], words[j]
                matrix[vocab_index[word_i], vocab_index[word_j]] += 1
                matrix[vocab_index[word_j], vocab_index[word_i]] += 1

    return matrix, vocabulary
```



```python
def create_term_document_matrix(documents):
    word_counts = Counter()
    for doc in documents:
        word_counts.update(doc.split())

    vocabulary = sorted(word_counts.keys())
    vocab_index = {word: i for i, word in enumerate(vocabulary)}

    matrix = np.zeros((len(vocabulary), len(documents)), dtype=np.int32)
    for doc_id, doc in enumerate(documents):
        for word in doc.split():
            matrix[vocab_index[word], doc_id] += 1

    return matrix, vocabulary
```

Most frequent items and pairs

On the left we can see the most frequent tokens (which are words in our case) and on the right we can see the pairs that has been most used. I am using context

window of 2 here so we are looking at one precinding and one future token to calculate this.

```
bir: 14476
və: 12408
bu: 6513
üçün: 4449
çox: 3260
film: 2364
ki: 2133
ilə: 2112
kimi: 2078
də: 1909
yaxşı: 1900
```

```
filmin: 909
yalnız: 903
sonra: 871
böyük: 863
pis: 851
bütün: 801
həqiqətən: 796
nə: 783
etmək: 759
olduğunu: 755
da: 729
lakin: 712
onu: 710
mən: 702
az: 661
tərəfindən: 637
```

```
('bir', 'və'): 134401
('bir', 'bu'): 61222
('bu', 'və'): 52059
('bir', 'üçün'): 46534
('və', 'üçün'): 40098
('bir', 'çox'): 34350
('və', 'çox'): 29609
('bir', 'ilə'): 23996
('bir', 'kimi'): 22727
('bir', 'film'): 22208
('bir', 'ki,'): 21379
('ilə', 'və'): 20682
('bir', 'də'): 20145
('bir', 'daha'): 20006
('kimi', 'və'): 19766
('bu', 'üçün'): 19201
('bir', 'olan'): 18973
('film', 'və'): 18785
('bir', 'yaxşı'): 18056
```

```
('ancaq', 'və'): 8596
('bir', 'filmi'): 8559
('bu', 'də'): 8365
('bir', 'onu'): 8333
('bir', 'etmək'): 8249
('bu', 'qədər'): 8249
('amma', 'və'): 8209
('bir', 'olduğunu'): 8182
('bir', 'bütün'): 8089
('bir', 'lakin'): 7957
('və', 'yalnız'): 7881
('idi.', 'və'): 7836
('bu', 'daha'): 7823
('bir', 'o,'): 7747
('böyük', 'və'): 7665
('filmin', 'və'): 7651
```

Term Document Matrix

Using term document matrix we can take a look at how many times specific token has been appeared in the various documents. Below is some random sample from it.

	385	730	201	1404	1965	116	1938	336	1044	1622	1709	1521	1381	840	822	1967	1061	553	1256	1759
ilə	1	7	7	6	5	7	2	2	9	3	4	1	2	1	4	3	8	2	2	3
kimi	4	11	2	3	1	3	2	2	4	1	5	3	6	3	2	4	3	7	1	7
də	9	3	4	1	6	4	4	3	1	1	10	9	5	6	3	2	0	3	2	5
yaxşı	3	2	1	0	3	2	5	1	2	5	0	3	2	0	5	3	1	5	0	5
daha	3	7	3	1	2	2	4	5	4	3	7	1	5	1	3	1	10	0	1	13
qədər	2	6	1	1	6	3	3	4	2	2	3	4	1	1	2	1	2	5	0	8
hər	4	2	0	2	2	6	2	3	0	4	9	6	2	2	7	2	2	7	0	3
olan	1	5	8	2	2	5	1	1	6	4	0	4	6	2	1	5	4	1	6	3
ən	2	2	1	1	3	3	0	10	3	0	3	3	2	0	2	6	1	3	3	2
bu,	1	3	1	1	1	3	0	4	2	0	3	3	2	2	2	1	0	4	0	6
ya	2	3	3	1	4	2	2	12	3	0	6	9	4	3	1	5	1	1	0	16
heç	3	0	9	2	1	2	3	6	3	3	2	0	1	6	4	2	1	7	1	1
şey	6	2	2	1	0	1	2	3	0	2	4	4	3	1	1	3	0	7	0	4
idi.	3	2	0	1	3	1	0	0	0	2	0	0	1	1	5	1	2	4	0	0
amma	6	0	1	3	0	0	1	1	1	0	4	4	3	1	0	1	0	0	1	3
filmi	2	0	1	0	0	1	2	3	0	0	1	0	0	3	2	0	0	4	0	1
onun	10	2	1	2	4	1	1	0	1	1	1	2	3	1	3	4	1	1	2	2
ancaq	0	1	1	1	4	0	0	1	0	0	0	1	4	3	2	4	0	1	2	1

TF-IDF features

Next up we calculate TF-IDF which calculates a word's importance by multiplying its frequency in a document (TF) with the inverse log proportion of documents containing the word (IDF). It is very essential as we need to be able to transform raw string text into numerical vectors so that we can use for Machine Learning modeling. The formula is listed below:

$$TF(t,d) = \frac{number\ of\ times\ t\ appears\ in\ d}{total\ number\ of\ terms\ in\ d}$$

$$IDF(t) = log\frac{N}{1+df}$$

$$TF-IDF(t,d) = TF(t,d) * IDF(t)$$

The code below calculates the TF-IDF features utilizing the `term_doc_matrix` is as follows:

```
def calculate_tf_idf(matrix):
    tf = matrix / np.sum(matrix, axis=0)
    idf = np.log(matrix.shape[1] / np.count_nonzero(matrix, axis=1))
    return tf * idf[:, np.newaxis]
```

Pointwise Mutual Information (PMI) features

Another method to get the features for the text is to use PMI that measures the association between two words by comparing their joint probability to the product of their individual probabilities, using logarithms. Here is the formula for the PMI:

$$PMI(a,b) = \log(\frac{P(a,b)}{P(\mathrm{a})\mathrm{P}(\mathrm{b})})$$

```python
EPSILON = 1e-8
def calculate_pmi(matrix):
    total_word_count = np.sum(matrix)
    word_count_per_document = np.sum(matrix, axis=1)
    word_frequency = word_count_per_document / total_word_count
    word_pair_prob = matrix / total_word_count

    pmi_matrix = np.zeros_like(matrix, dtype=np.float64)
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[1]):
            pmi_matrix[i, j] = np.log((word_pair_prob[i, j] + EPSILON) / ((word_frequency[i] + EPSILON) * (word_frequency[j] + EPSILON)))

    return pmi_matrix
```

Logistic Regression

Next up we are building Logistic regression from scratch where we are optimizing and training parameters for each of the TF-IDF columns. The parameters are as folows

- Learning Rate
- Number of iterations
- L2 Regularization
- Intercept

And this is the code for Logistic Regression:

```python
class CustomLogisticRegression(BaseEstimator, ClassifierMixin):
    def __init__(self, lr=0.01, num_iter=100, fit_intercept=True, reg_param=0.1):
        self.lr = lr
        self.num_iter = num_iter
        self.fit_intercept = fit_intercept
        self.reg_param = reg_param

    def __add_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        self.theta = np.zeros(X.shape[1])

        for i in range(self.num_iter):
            z = np.dot(X, self.theta)
            h = self.__sigmoid(z)
            gradient = np.dot(X.T, (h - y)) / y.size
            reg_term = self.reg_param * self.theta / y.size
            self.theta -= self.lr * (gradient + reg_term)

    def predict_proba(self, X):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        return np.column_stack((1 - self.__sigmoid(np.dot(X, self.theta)), self.__sigmoid(np.dot(X, self.theta))))

    def predict(self, X):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        return (self.__sigmoid(np.dot(X, self.theta)) >= 0.5).astype(int)

    def score(self, X, y):
        return (self.predict(X) == y).mean()
```

Logistic Regression Evaluation

Now that we have model we can use both TF-IDF and PMI. Upon observing we see there is not too much difference in the accuracy, f1 and so on for our case.

TF-IDF

```
Train Accuracy:
0.950625

Test Accuracy:
0.7525

Train F1 Score:
0.9501577287066246

Test F1 Score:
0.7681498829039812


Test Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.69      0.73       199
           1       0.73      0.82      0.77       201

    accuracy                           0.75       400
   macro avg       0.76      0.75      0.75       400
weighted avg       0.76      0.75      0.75       400


Test Confusion Matrix:
[[137  62]
 [ 37 164]]
```

PMI

```
Train Accuracy:
0.936875

Test Accuracy:
0.7525

Train F1 Score:
0.9344581440622972

Test F1 Score:
0.7480916030534351


Test Classification Report:
              precision    recall  f1-score   support

           0       0.74      0.77      0.76       199
           1       0.77      0.73      0.75       201

    accuracy                           0.75       400
   macro avg       0.75      0.75      0.75       400
weighted avg       0.75      0.75      0.75       400


Test Confusion Matrix:
[[154  45]
 [ 54 147]]
```

Random Search Hyperparameter Tuning

I have additionally tune following parameters using random search and we see that I am getting quite good increase leap over the previous model. We also see that model overfits which could be fixed by decreasing TF-IDF features or using regularization.

- Learning Rate
- Number of iterations
- Regularization

```python
def random_search(X_train, y_train):
    param_grid = {
        'lr': uniform(0.01, 1),
        'num_iter': [500, 1000, 5000, 10000],
        'reg_param': [0.001, 0.01, 0.1, 1]
    }
    model = CustomLogisticRegression()
    random_search = RandomizedSearchCV(model, param_distributions=param_grid, n_iter=10, cv=5, scoring='accuracy', random_state=42)
    random_search.fit(X_train, y_train)
    best_params = random_search.best_params_
    return best_params

X_train, X_test, y_train, y_test = train_test_split(tf_idf_features.T, df['label'].values, test_size=0.2, random_state=42)

best_params = random_search(X_train, y_train)
print("Best parameters found:", best_params)
model = CustomLogisticRegression(lr=best_params['lr'], num_iter=best_params['num_iter'])
model.fit(X_train, y_train)

evaluation_results, losses = evaluate_model(model, X_train, X_test, y_train, y_test)
for metric, value in evaluation_results.items():
    print(f'{metric}:')
    print(value)
    print()
```

✓ 43m 52.9s

```
Accuracy: 0.82
F1 Score: 0.8260869565217391
Confusion Matrix:
 [[157  42]
 [ 30 171]]

Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.79      0.81       199
           1       0.80      0.85      0.83       201

    accuracy                           0.82       400
   macro avg       0.82      0.82      0.82       400
weighted avg       0.82      0.82      0.82       400
```

Comparison of classification with Naive Bayes

When we compare the Naive Bayes and Logistic Regression we see that latter outperformes the former one

Naive Bayes

```
Accuracy: 0.78
F1 Score: 0.7582417582417582

Classification Report:
              precision    recall  f1-score   support

           0       0.73      0.87      0.80       199
           1       0.85      0.69      0.76       201

    accuracy                           0.78       400
   macro avg       0.79      0.78      0.78       400
weighted avg       0.79      0.78      0.78       400


Confusion Matrix:
 [[174  25]
 [ 63 138]]
```

Logistic Regression

```
Accuracy: 0.82
F1 Score: 0.8260869565217391

Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.79      0.81       199
           1       0.80      0.85      0.83       201

    accuracy                           0.82       400
   macro avg       0.82      0.82      0.82       400
weighted avg       0.82      0.82      0.82       400

Confusion Matrix:
 [[157  42]
 [ 30 171]]
```

Cosine Similarity

Finally we find most similar words for the given word by using Cosine similarity that is based on the TF-IDF feartures. As we can see there is `pis` and `dəhşətli` also `bəyəndim` and `bravo`

```python
def cosine_similarity(vec1, vec2):
    dot_product = np.dot(vec1, vec2)
    norm_vec1 = np.linalg.norm(vec1)
    norm_vec2 = np.linalg.norm(vec2)
    return dot_product / (norm_vec1 * norm_vec2)

def most_similar_words(word, word_vectors, vocab, top_n=10):
    if word not in vocab:
        return []

    word_index = vocab.index(word)
    similarities = []
    for i, vec in enumerate(word_vectors):
        if i != word_index:
            sim = cosine_similarity(word_vectors[word_index], vec)
            similarities.append((vocab[i], sim))

    similarities.sort(key=lambda x: x[1], reverse=True)
    return similarities[:top_n]
```

```
second_sim(given_word='pis')
✓ 0.1s
['bu', 'qədər', 'bir', 'film', 'ən', 'və', 'üçün', 'dəhşətli', 'deyil', 'filmi']
```

10 most similar words to '**bəyəndim**':
samantha: 0.7869332032190166
bravo 0.7869332032190166
olanda.: 0.7869332032190166
stansiya: 0.7845372906768673
ləqəb: 0.7603623560825417
bilməz: 0.7464255628829782
oxumağı: 0.7174692276295543
səhnəyə: 0.6280834834525226