Introduction

In this assignment, we are required to work with the ProgrammableWeb data text file i.e. API.txt. This file contains service data and data retrieved through ProgrammableWeb APIs. We are required to implement **web service classification** and **clustering** by using features based on one or more service attributes. We also need to preprocess the web service data and run the selected algorithms in Python, present and discuss the evaluation result.

Design of the Web service data analysis system

Technologies/components used

Technology	Usage
Python	For general-purpose programming
Training Model	TFIDF, LDA, Word2Vec, BERT
Classification Algorithms	Decision Tree Classifier, Naive Bayesian Classifier, K Nearest Neighbor Classifier
Clustering Algorithms	K Means Clustering, DBSCAN Clustering

Feature Selection

I selected the name, description, and tags as part of my service description. The reason for choosing these was all these columns help to identify a service uniquely. Tags and descriptions especially add more value to an individual service. I tried first with tags and summary but that gave me less efficiency as the summary was a short form of description which tended to lose important features while training.

Data Cleaning Pipeline

To clean the data so that the model can properly perform classification and clustering, I have performed the below cleaning steps.

1. **Case normalization** - Converting all the words into small cases so that all the words will be treated the same.
2. **Remove rows having null values** - I found the rows where there were columns like rating and limits with null values. Hence I removed those rows from my analysis.
3. **Remove rows having duplicate values** - I did not find any rows with duplicate values, but I did it as a part of cleaning.
4. **Balance the data** - I used **categories** present in the input API records

to balance the data. I began by counting the occurrences of each category and sorting them in descending order of frequency. Then, I identified the category with the highest count and set a threshold of 30% of that count. The threshold value of 30% is common as I researched it over the internet. Then I iterated through the sorted categories, adding those that meet or exceed the threshold to a list. Then filter the input records based on these selected categories, creating a new data frame with balanced categories. Finally, return the balanced dataset.

5. **Remove stop words** - I used this method to remove words that don't add meaning to a sentence or help to identify a group uniquely. After applying this technique, words like “the”, “is”, to and cause were removed from the API records.

Program Flow

The code will go through the below steps

1. Load the API data and find the features that can uniquely identify the labels.
2. Clean the API data, this includes case normalization, removal of nulls and duplicates, stopwords, etc as already mentioned above.
3. Balance the data for better classification and clustering results. As well as to avoid outliers in the data set.
4. Perform modeling using the TFIDF, LDA, Word Embed, and BERT Models followed by classification.
5. Perform modeling using the TFIDF, LDA, Word Embed, and BERT Models followed by clustering.

Text Classification

For performing text classification, we have used 4 different modeling approaches. The results of these approaches are verified by using 3 classification algorithms.

Approach 1: TFIDF

This approach stands for Term Frequency/Inverse Document Frequency. It's a statistical measure used to retrieve information and mine the text to find the importance of a word in a document relative to a collection of other documents. The term, “Term Frequency” means the number of times a term occurs in a document. The term “Inverse Term Frequency” means the number of documents that contain the term.

I have gotten the below accuracies when I used different classification algorithms like KNN, Nayes Bayesian, and Decision Tree classification.

Hyper-parameters:
Decision Tree - No parameters used
Naive Bayesian - No Parameters used
KNN - n_neighbors=15 -> This value gave me better results with less overfitting

```
TFIDF - Decision Tree Classifier
        Cross-validation scores for tfidf: 0.7843936961584552
        Accuracy for tfidf:  0.782312925170068


TFIDF - Naive Bayesian Classifier
        Cross-validation scores for tfidf: 0.5090383702263324
        Accuracy for tfidf:  0.54421768707483


TFIDF - KNN Classifier
        Cross-validation scores for tfidf: 0.7237940624505391
        Accuracy for tfidf:  0.7091836734693877
```

As can be seen for the Decision Tree classifier the accuracy is 0.78 or 78% For Naives Bayesian it is 0.54 or 54% and for KNN it is 0.70 or 71%.

Observations:

1. The Decision Tree classifier achieved the highest accuracy, indicating that it was able to effectively capture the underlying patterns in the dataset without overfitting.
2. The KNN classifier achieved the second-highest accuracy, suggesting that it performed reasonably well in capturing the similarities between instances in the feature space.
3. The Naive Bayes classifier achieved the lowest accuracy, possibly due to the oversimplified assumption of feature independence.

Approach 2: LDA

In this case, the general idea is to model the latent topics in textual descriptions. And then to use the topics to model a service's functionality. There is an assumption that each topic is represented as a probabilistic distribution over terms and each document covers multiple topics.

Hyperparameters:
LDA object = **n_components**=10, **random_state**=42. This means that I am looking for 10 topics in my dataset, and ensuring that the random initialization is consistent across.
Decision Tree - No parameters used
Naive Bayesian - No Parameters used
KNN - n_neighbors=15 -> This value usually gives better results when tested

```
LDA - Decision Tree Classifier
        Cross-validation scores for lda: 0.3176690709295227
        Accuracy for lda:  0.30017006802721086


LDA - Naive Bayesian Classifier
        Cross-validation scores for lda: 0.3638033328057521
        Accuracy for lda:  0.3835034013605442


LDA - KNN Classifier
        Cross-validation scores for lda: 0.39761910145386303
        Accuracy for lda:  0.4141156462585034
```

As can be seen for the Decision Tree classifier the accuracy is 30%, For Naives Bayesian it is 38% and for KNN it is 41%.

Observations.

1. Since LDA transforms the original text data into a lower-dimensional space that represents the topics, the decision tree may have difficulties finding meaningful and unique splits in this space. As a result, the LDA-transformed features might not be well-suited for decision trees. Naive Bayes classifiers due to violations of their underlying assumptions and difficulties in decision boundary delineation cannot perform well.
2. KNN was able to achieve slightly better accuracy, indicating that the LDA transformation might have preserved enough information for KNN to effectively classify instances.

Approach 3: Word Embedding

This approach models each word as a vector of a fixed number of dimensions, like embedding each word in a space. And then encode continuous similarities between words as distance or angle between word vectors. This is where the concept of cosine similarity comes into play.

Hyper-parameters:
Word2Vec = sentences, window=5, min_count=1, epochs=10. Sentences here are the split documents, window value is the Maximum distance between the current word and the predicted word within a sentence, min_count is the Minimum frequency threshold for words to be included in the vocabulary, and epochs are the number of iterations over the entire dataset during training.
Decision Tree - No parameters used
Naive bayesian - No Parameters used
KNN - n_neighbors=15 -> This value usually gives better results

```
Word Embedding - Decision Tree Classifier
        Cross-validation scores for word embedding: 0.4890273556231003
        Accuracy for word embedding:  0.4413265306122449


Word Embedding - Naive Bayesian Classifier
        Cross-validation scores for word embedding: 0.5441389491966999
        Accuracy for word embedding:  0.5382653061224489


Word Embedding - KNN Classifier
        Cross-validation scores for word embedding: 0.6313988999855261
        Accuracy for word embedding:  0.6232993197278912
```

As can be seen for the Decision Tree classifier the accuracy is 44% For naive Bayesian it is 53% and for KNN it is 62%.

Observations:

1. The results suggest that word embeddings might not be good for decision tree classification due to difficulties in finding meaningful splits in the high-dimensional embedding space similar to that of LDA.
2. Naive Bayes, despite its simplicity and the independence assumption, can still perform reasonably well with word embeddings, attributed to its robustness.
3. KNN achieved the highest accuracy among the other classifiers, indicating that word embeddings might be effective for classification methods which is similarity-based like this one.

Approach 4: BERT

Bert can be used to extract language features from words or sentence vectors. The model is context-aware and generates dynamic vectors which is different from static vectors in Word2Vec. It can directly be used or fine-tuned on a specific task like classification or clustering.

Hyper-parameters:
Decision Tree - No parameters used
Naive Bayesian - No Parameters used
KNN - n_neighbors=15 -> This value usually gives better results

```
BERT Embedding - Decision Tree Classifier
        Cross-validation scores for bert embedding: 0.325
        Accuracy for bert embedding:  0.35
BERT Embedding - Naive Bayesian Classifier
        Cross-validation scores for bert embedding: 0.6475000000000001
        Accuracy for bert embedding:  0.665
BERT Embedding - KMeans Classifier
        Cross-validation scores for BERT embedding: 0.6300000000000001
        Accuracy for BERT embedding:  0.625
```

Note: this accuracy is calculated only for the first 5000 data records due to memory issues in my laptop. Also tried to run it on Google Collab but ran over runtime and crash error multiple times.

As can be seen for the Decision Tree classifier the accuracy is 0.35 or 35% For Naives Bayesian it is 0.66 or 66% and for KNN it is 0.62 or 62%.

Observation:

1. Naive Bayes performed exceptionally well, likely due to its simplicity and ability to effectively utilize high-dimensional feature spaces.
2. KNN also achieved high accuracy, indicating that BERT embeddings captured semantic similarities between instances that were useful for classification.
3. The results above show that BERT embeddings are well-suited for classification tasks, as evidenced by the relatively high accuracies achieved by Naive Bayes and KNN as compared to Decision Tree Classifier

Text Clustering

For performing text clustering, we have used 4 different modeling approaches again just like above. The data cleaning task remains the same as already discussed above in the design section The results of these approaches are verified by using 2 clustering algorithms namely DBSCAN and K means clustering. The accuracy metric used for all the models by the clustering algorithms is the silhouette score. The silhouette score evaluates the quality of clustering in unsupervised learning. It finds how similar an object is to its cluster compared to other clusters in the dataset. The score usually is between -1 to 1

K Means Clustering:

K-means clustering divides a dataset into K groups by iteratively adjusting cluster centers to minimize the distance between data points and their assigned clusters. It aims to find clusters that minimize within-cluster variance.

Hyper-parameter:
K Means: n_clusters=50, init='k-means++', n_init=1, max_iter=100, tol=1e-4, random_state=0
Using 50 clusters with initialization method as k-means++. The number of iterations is 100 and the random state is 0. The tolerance level for convergence is set to a very low value. This combination and the values of the parameters gave me a slightly better silhouette score for all the models

```
TFIDF with Silhouette Score: 0.013499115805789256
/Users/shreenidhi/Desktop/webservices/mypa4/trial_new.py:217: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  data['clusters'] = kmeans.labels_
LDA with Silhouette Score: 0.36224353044129504
Word Embed with Silhouette Score: 0.06703643100988561
BERT with Silhouette Score 0.09854121
```

Clustering with TFIDF:
Note: No hyper-parameter
The technique for finding the vector for clustering is the same as classification except did not use the labels(y)
TFIDF with Silhouette Score: 0.013
This score indicates that **k means** was not able to properly cluster the data points because the score is closer to 0 than 1.

Clustering with LDA:
Hyper-parameter: n_components=13, random_state=42 (same as classification)
LDA with Silhouette Score: 0.362
This score is better than the above score(TFIDF). It means that k means was able to cluster the data points.

Clustering with Word Embedding:
Hyper-parameter: sentences, window=5, min_count=1, epochs=10 (same as classification)
Word Embed with Silhouette Score: 0.067
This score is better than the above score(TFIDF) but not better than LDA.

Clustering with BERT:
Note: Used starting 1000 points to perform clustering due to storage problems in my machine.
BERT with Silhouette Score 0.098
This score is better than the TFIDF and Word Embed but not better than LDA.

Observations:

1. LDA clustering produced the highest Silhouette Score, indicating the best clustering performance among the techniques mentioned.
2. TF-IDF clustering had the lowest Silhouette Score, suggesting the poorest clustering performance.
3. Word Embeddings and BERT clustering achieved moderate Silhouette Scores, indicating somewhat satisfactory clustering performance.

DBSCAN Clustering:

DBSCAN is a clustering method that groups nearby points. It looks for points with many neighbors close by and then expands clusters from these points. It's good at finding clusters of different shapes and handling noisy data.

Hyper-parameter:
I have used different values for **eps** and **min_samples** for each of the modeling techniques. The accuracy metric used for all the models by the clustering algorithms is the silhouette score. Some combinations of **eps** and **min_samples** are giving me negative silhouette values and some give positive values. For every modeling technique, I have used different values of eps and min_samples to get the best clustering score.

```
TFIDF with Silhouette Score -0.010509150517637912
LDA with Silhouette Score 0.004927863632870041
Word Embed with Silhouette Score: 0.21651861
BERT with Silhouette Score 0.4806833
```

Clustering with TFIDF:
Note: No hyper-parameter
Eps = 1.0, **min_samples** = 10
The technique for finding the vector for clustering is the same as classification.
TFIDF with Silhouette Score -0.01051
The score usually is between -1 to 1, and this score indicates that k means was not able to properly cluster the data points because the score is closer to -1.

Clustering with LDA:
Hyper-parameter: n_components=13, random_state=42 (same as classification)
Eps = 0.1, **min_samples** = 3
LDA with Silhouette Score 0.0049
This score is better than the above score(TFIDF). But still, It indicates that k means was not able to properly cluster the data points.

Clustering with Word Embedding:
Note: Used starting 1000 points to perform clustering due to storage problems in my machine.
Hyper-parameter: sentences, window=5, min_count=1, epochs=10 (same as classification)
Eps = 3.0, **min_samples** = 3
Word Embed with Silhouette Score: 0.256
This score is better than the above score of TFIDF and LDA. This score is not closer to 1. But still, it means that word embedding was able to cluster the data points more accurately as compared to previous techniques. Given more training data i.e. > 1000. This model would have given a better score.

Clustering with BERT:
Note: Used starting 1000 points to perform clustering due to storage problems in my machine.
BERT with Silhouette Score 0.4806833
This score is better than all the models. This score is almost halfway to 1. The BERT embedding was able to cluster the data points more accurately as compared to previous techniques. Given more training data i.e. > 1000. This model would have given a better score.

Observations:
Clustering with BERT embeddings achieved the highest Silhouette Score, indicating better clustering performance compared to TFIDF, Word embedding, and LDA. TFIDF modeling performed the worst among all models. It's important to note that, clustering with BERT and Word Embeddings used only a subset of the data points due to storage limitations in my machine, which could have impacted the results. Given more training data i.e. > 1000. Both these models i.e. BERT and word embedding would have given a better score.

Set up and Test Instructions

1. Extract and Open the folder "**Acharya_Shreenidhi_A4**" in visual studio code or pycharm.
2. Run the command "**python -m venv .**" in a terminal (**CTRL+~)** to create a virtual environment for this application.
   Note: If you get error in the above command, you can use **python3 -m venv .**
3. Run the command "**source bin/activate**" to activate the created environment.
4. Run the command "**pip install -r requirements.txt**" to install all dependencies.
   Note: if you encounter error with pip command use "**pip3 install -r requirements.txt"** instead.
5. Run the command **python Analysis.py**
   Note: If you get an error in the above command, you can use **python3 Analysis.py**