Sentiment Analysis

using semi-supervised text classification

Fettache Dina
Lebgaa Hanane

Introduction

Sometimes, obtaining large labeled datasets for supervised learning can be impractical, Which makes it go to semi-supervised text classification, a paradigm that harnesses the power of both labeled and unlabeled data. This report navigates through the theoretical foundations and practical implementations of semi-supervised text classification, spotlighting self-training, and Graph-based semi-supervised learning. A case study will show a real-world application. By the end, we will grasp the potential of leveraging unlabeled data, overcoming challenges, and enhancing text classification models.

Challenges of Text Classification

Text classification involves assigning predefined categories or labels to textual data. Traditional supervised approaches rely on labeled examples for training, limiting scalability and applicability. Semi-supervised text classification addresses these challenges by incorporating unlabeled data, making the model more robust and adaptable to diverse text sources.

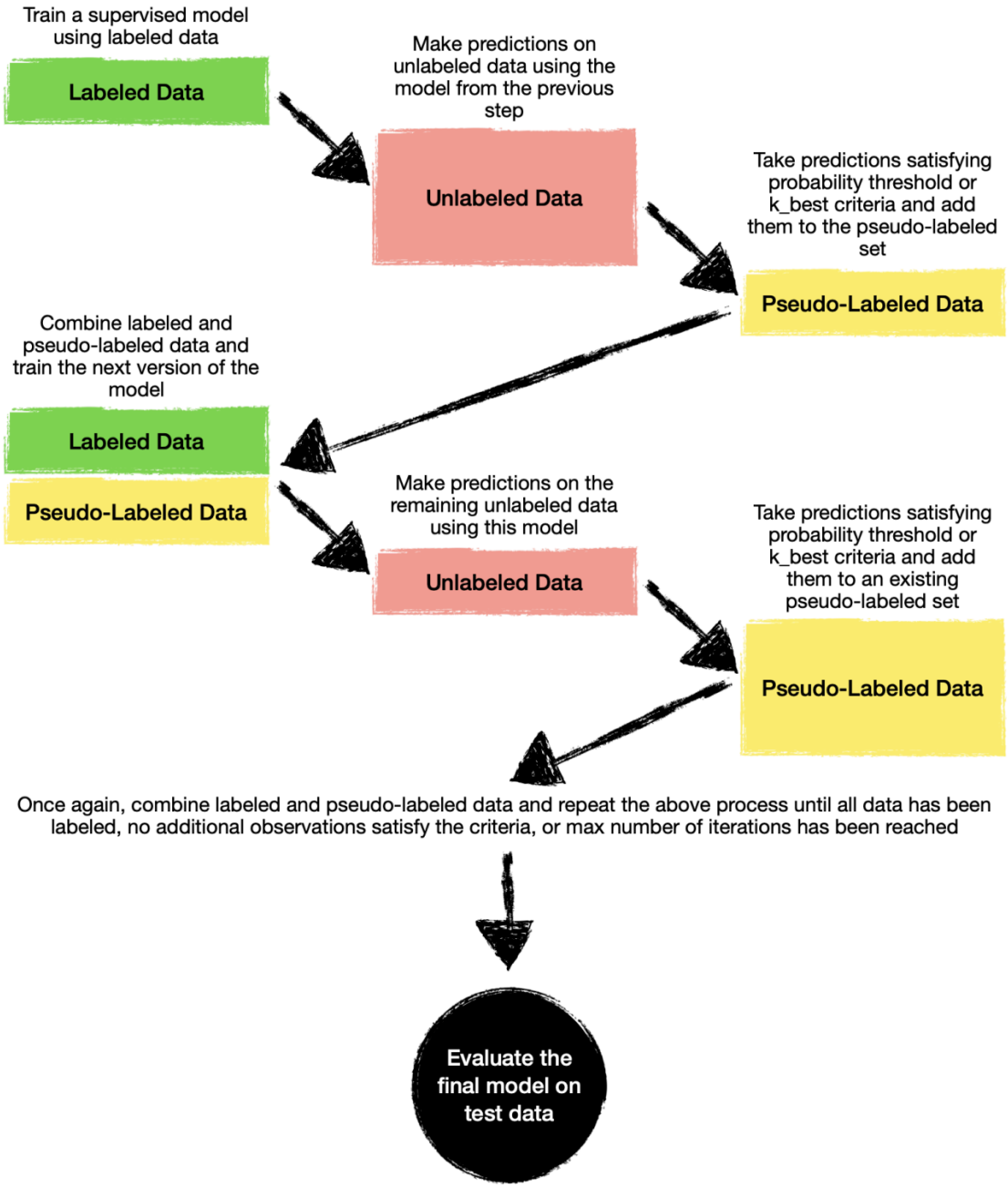
Semi-Supervised Machine Learning

Semi-supervised learning lies between supervised and unsupervised learning, making use of both labeled and unlabeled data during model training. Unlike traditional supervised learning, where models rely only on labeled examples, semi-supervised methods leverage the additional information from unlabeled instances to improve performance and address challenges associated with limited labeled datasets.

Approaches

1. Self-Training:

Self-training involves iteratively training a model on the labeled data and then using it to predict labels for unlabeled instances. The predicted labels are added to the labeled set for subsequent training iterations.



2. Graph-Based Approaches (Label Propagation and Label Spreading):

These graph-based methods propagate labels from labeled to unlabeled instances based on the relationships or similarities between data points in a graph representation

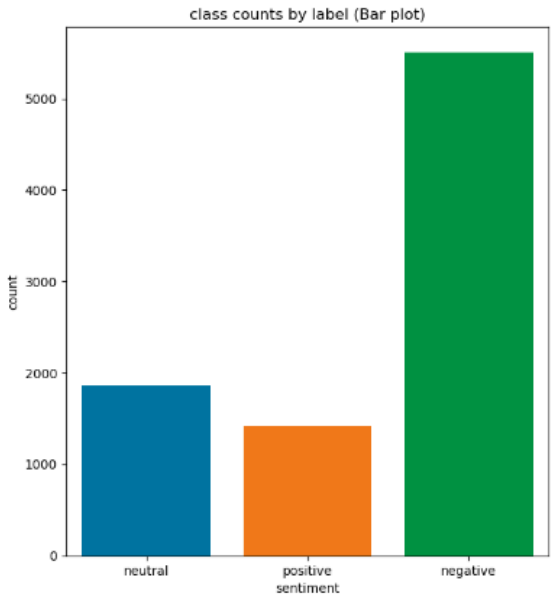
Dataset

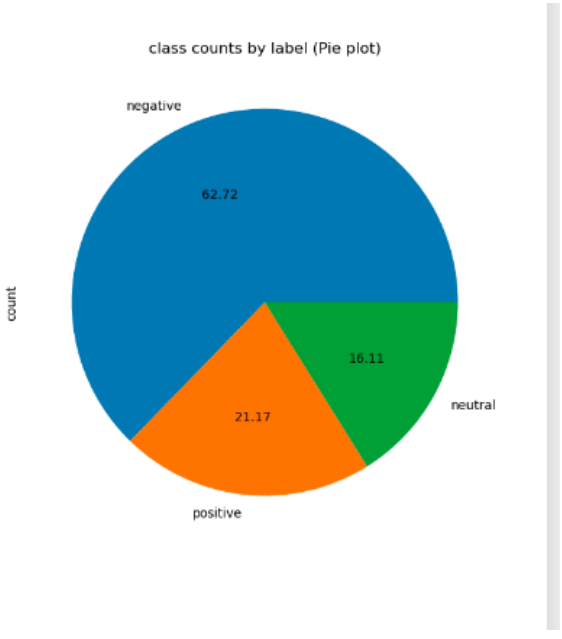
The dataset used for sentiment analysis comprises 3 classes(Negative, Positive, Neutral), an overall sentiment of customers towards flights with fully labeled data that will be handled later in preprocessing.

Performance metric

Imbalanced Dataset Challenge

In the context of sentiment analysis, the dataset exhibits class imbalance, with varying numbers of instances across the Negative, Positive, and Neutral classes. This poses a challenge as traditional accuracy may be misleading in such scenarios, where the model could achieve high accuracy by predominantly predicting the majority class.





Precision, Recall, and F1 Score

To address the imbalanced dataset challenge, a careful selection of performance metrics is essential. The F1 score, which is the harmonic mean of precision and recall, proves to be particularly effective in scenarios with imbalanced classes.

- Precision: Measures the accuracy of positive predictions, which is the ratio of correctly predicted positive observations to the total predicted positives.
- Recall (Sensitivity): Measures the ability of the model to capture all the relevant positive instances, i.e., the ratio of correctly predicted positive observations to all actual positives.
- F1 Score: Strikes a balance between precision and recall. It is especially useful when there is an uneven class distribution, as it considers false positives and false negatives.

Advantages of F1 Score

Sensitivity to Minority Classes: The F1 score takes into account both false positives and false negatives, providing a balanced evaluation that is sensitive to the performance of minority classes.
Model Generalization: The F1 score is beneficial in ensuring that the model generalizes well across all sentiment classes, preventing biases toward the majority class.

Interpretation

- A higher F1 score indicates a balanced trade-off between precision and recall, demonstrating the model's effectiveness in capturing positive instances while minimizing false positives.
- The use of the F1 score aligns to achieve a sentiment analysis model that performs well across all sentiment classes, irrespective of their initial imbalances.

Given the imbalanced nature of the sentiment dataset, the F1 score stands out as a reliable performance metric, providing a comprehensive assessment of the model's predictive capabilities across different sentiment classes

Preprocessing

1. Text Cleaning:

- A comprehensive text cleaning process involves removing emojis, punctuations, links, mentions, and newline characters. Hashtags were cleaned, and multiple spaces were reduced to a single space. and then Cleaned text was added as a new column ('cleaned_text') for further analysis.

In [7]:
```
# Clean  text
def clean_text_from_emojis(text):
    return demoji.replace(text, '')
```

In [8]:
```
#Remove punctuations, links, mentions and \r\n new line characters
def strip_all_entities(text):
    text = text.replace('\r', '').replace('\n', ' ').replace('\n', ' ').lower() #remove \n and \r and lowercase
    text = re.sub(r"(?:\@|https?\://)\S+", "", text) #remove links and mentions
    text = re.sub(r'[^\x00-\x7f]',r'', text) #remove non utf8/ascii characters such as '\x9a\x91\x97\x9a\x97'
    banned_list= string.punctuation + 'Ã'+'±'+'ã'+'¼'+'å'+'»'+'§'
    table = str.maketrans('', '', banned_list)
    text = text.translate(table)
    return text
```

In [9]:
```
# Clean hastags '#' Symbol
def clean_hashtags(tweet):
    new_tweet = " ".join(word.strip() for word in re.split('#(?!(?:hashtag)\b)[\w-]+(?=(?:\s+#[\w-]+)*\s*$)', tweet)) #remove las
    new_tweet2 = " ".join(word.strip() for word in re.split('#|_', new_tweet)) #remove hashtags symbol from words in the middle
    return new_tweet2
```

In [10]:
```
# Remove multiple spaces
def remove_mult_spaces(text):
    return re.sub("\s\s+" , " ", text)
```

In [11]:
```
#dataset1:df
dataset1_texts_new = []
for t in df.text:
    dataset1_texts_new.append(remove_mult_spaces(clean_hashtags(strip_all_entities(clean_text_from_emojis(t)))))
```

2. Unlabled:

A strategic portion of labeled data in the Airline-sentiment-2 dataset was randomly selected and turned into unlabeled data to simulate a semi-supervised learning scenario.

In [13]:
```
# Randomly select a portion of the data to turn into unlabeled
percentage_to_turn_unlabeled = 0.4  # Adjust as needed
num_samples_to_turn_unlabeled = int(len(df1) * percentage_to_turn_unlabeled)
samples_to_turn_unlabeled = df1.sample(num_samples_to_turn_unlabeled)
```

In [14]:
```
# Set the 'label' column for the selected samples to NaN
df1.loc[samples_to_turn_unlabeled.index, 'sentiment'] = None

df1= df1[['cleaned_text','sentiment']]
df1.head()
```

3. Handling Class Imbalance with SMOTE:

To address the class imbalance, the SMOTE (Synthetic Minority Over-sampling Technique) was applied inside the pipeline to augment the minority class data and create a more balanced training set.

4. Scaling using StandardScaler:

StandardScaler is a preprocessing step used in the pipeline to standardize the features by removing the mean and scaling to unit variance. In the context of your text data, this step helps ensure that the TF-IDF values for each feature have a mean of 0 and a standard deviation of 1. Standardization is particularly important when working with machine learning models that are sensitive to the scale of input features, such as Support Vector Machines or k-nearest Neighbors.

Reason for with_mean=False in StandardScaler: In the case of TF-IDF vectors or other sparse data representations, setting with_mean=True might not be appropriate. Sparse matrices are often used to represent text data due to the large number of zeros in the feature space. Sparse data cannot be centered as it may result in a dense matrix, defeating the purpose of using a sparse representation. Therefore, setting with_mean=False is essential when dealing with sparse data, indicating that the mean should not be subtracted during the standardization process.

5. Text Vectorization using TF-IDF:

The TF-IDF (Term Frequency-Inverse Document Frequency) vectorization technique was employed in the pipeline to convert text data into a numerical format suitable for machine learning models.

SMOTE

To mitigate the impact of class imbalance, the Synthetic Minority Over-sampling Technique (SMOTE) was employed. SMOTE is a data augmentation method designed to oversample the minority class by generating synthetic instances, thereby achieving a more balanced class distribution.

SMOTE was seamlessly integrated into the training pipeline alongside the chosen machine learning model. The SMOTE algorithm was applied specifically to the minority

class instances during the training phase. This integration aimed to provide the model with a more diverse set of examples from the minority class, enhancing its ability to generalize and make accurate predictions for under-represented instances.the default parameter for SMOTE is k=5 as the number of nearest neighbors

The nearest neighbors are used to define the neighborhood of samples to use to generate the synthetic samples. You can pass:

- an int corresponding to the number of neighbors to use. A ~sklearn. neighbors.NearestNeighbors instance will be fitted in this case.
- an instance of a compatible nearest neighbors algorithm that should implement both methods kneighbors and kneighbors_graph. For instance, it could correspond to a Nearest Neighbor but could be extended to any compatible class.

Smote on self-train classifier:

applying SMOTE to a Self-Training Classifier in a semi-supervised text classification scenario with unbalanced data can have several advantages and considerations. The advantages include improved class balance, increased model sensitivity, enhanced generalization, and reduced bias. However, potential challenges include the risk of overfitting.

Smote on label-propagation and label-spreading classifier:

When you apply SMOTE to a dataset before label propagation, you are essentially creating synthetic samples to balance the class distribution. While this can be beneficial in addressing the class imbalance issue, it may also introduce noise which leads to misclassifications, and reduce the overall performance. which is why we will only consider models that do not include smote.

TF-IDF

TF-IDF, which stands for Term Frequency-Inverse Document Frequency, is a numerical statistic used in natural language processing and information retrieval to evaluate the importance of a word in a document relative to its frequency in a collection of documents, or corpus. Here's a breakdown of how TF-IDF works and its purpose in text classification:

Term Frequency (TF):

- **Definition:** Measures the frequency of a term in a document.
- **Calculation:** Ratio of the number of occurrences of a term to the total number of terms in the document.
- **Purpose:** Reflects how often a term appears within a specific document.

Inverse Document Frequency (IDF):

- **Definition:** Measures how important a term is across multiple documents.
- **Calculation:** The logarithm of the ratio of the total number of documents to the number of documents containing the term.
- **Purpose:** Highlights the uniqueness or rarity of a term across the entire corpus.

TF-IDF Score:

- **Definition:** Obtained by multiplying TF and IDF.
- **Calculation:** TF-IDF = TF * IDF
- **Purpose:** Quantifies the importance of a term in a specific document within the context of the entire corpus.

Purpose of TF-IDF in Text Classification:

- **Capturing Significance of Words:** TF-IDF helps address the challenge of representing textual data in a way that captures the significance of individual words.
- **Discriminative Power:** It assigns higher scores to terms that are frequent within a document but relatively rare in the entire corpus. This emphasizes the discriminative power of terms that are unique to certain documents.
- **Text Classification:** TF-IDF is commonly used in text classification tasks where the goal is to automatically categorize or label documents based on their content. By assigning weights to terms based on their importance in a document and across the corpus, TF-IDF helps in creating feature vectors that can be used as input for machine learning algorithms.

Models

Self-train text classifier:

The core concept of a Self-Training Classifier involves utilizing a standard classification algorithm iteratively to progressively label initially unlabeled data points. The process unfolds through the following sequential steps:

1. Data Gathering:

- Combine both labeled and unlabeled data, utilizing only the labeled observations for the initial training of a supervised model.

2. Model Prediction:

- Employ the trained model to predict the class labels for the unlabeled data.

3. Pseudo-Labeling:

- Select observations based on predefined criteria (high prediction probability or inclusion in the top 10% with the highest probabilities).

criteria: it could be set to 'threshold', which means pseudo-labels with prediction probabilities above the threshold are added to the dataset. or 'k_best', the k_best pseudo-labels with the highest prediction probabilities are added to the dataset

threshold: ***default=0.75,*** k_best: ***default=10.***

- Concatenate these pseudo-labels into the labeled data, expanding the training set.

4. Iterative Model Training:

- Train a new supervised model using the merged dataset containing both labeled and pseudo-labeled observations.

- Repeat the prediction process, integrating newly selected observations into the pool of pseudo-labeled data.

5. Iteration Continuation:

- Continue iterating through these steps until one of the following conditions is met: all data points are labeled, no additional unlabeled observations meet the pseudo-labeling criteria, or a specified maximum number of iterations is reached.

In essence, the Self-Training Classifier refines its predictions iteratively by leveraging both initially labeled and progressively pseudo-labeled data. This iterative approach aims to improve the model's performance over successive cycles, effectively incorporating unlabeled data into the training process and enhancing the classifier's ability to generalize to the entire dataset.

by changing the bese algorithms inside the Self-train model ,we have generated 8 models

K-Nearest Neighbors (KNN): a non-parametric, lazy supervised learning algorithm used for classification and regression. It classifies data points based on the majority class of their k-nearest neighbors. It is intuitive and easy to understand, making it suitable for various applications.

Results with Knn:

before Smote:

```
In [11]: f1 = f1_score(y_test, y_pred, average='weighted')
         print("F1 Score:", f1)

         F1 Score: 0.17452686133113168

In [12]: print(accuracy_score(y_test, y_pred))

         0.2214001138303927

In [13]: classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.78      0.07      0.13      1095
                    1       0.49      0.14      0.22       384
                    2       0.17      0.92      0.28       278

             accuracy                           0.22      1757
            macro avg       0.48      0.38      0.21      1757
         weighted avg       0.62      0.22      0.17      1757


In [14]: conf_matrix = confusion_matrix(y_test, y_pred)
         pd.DataFrame(conf_matrix)

Out[14]:
```

	0	1	2
0	79	40	976
1	16	54	314
2	6	16	256

After Smote:

```
In [25]: # Compute F1 score
         f1 = f1_score(y_test, y_pred, average='weighted')
         print("F1 Score is : ", f1)

         F1 Score is :  0.3173583336364451

In [26]: print("Accuracy is : ", accuracy_score(y_test, y_pred))

         Accuracy is :  0.32498577120091066

In [27]: classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.79      0.19      0.30      1095
                    1       0.31      0.40      0.35       384
                    2       0.21      0.77      0.33       278

             accuracy                           0.32      1757
            macro avg       0.44      0.45      0.33      1757
         weighted avg       0.59      0.32      0.32      1757
```

Stochastic Gradient Descent (SGD):

SGD is an iterative optimization algorithm commonly used for training machine learning models, particularly in the context of large datasets. It updates the model parameters by considering only a subset of the data (a batch) in each iteration, making it computationally efficient.

Results with SGD:

Bofore Smote:

```
F1 Score: 0.7269562957693754

Accuracy is : 0.7558338076266363
```

	0	1	2
0	1056	25	11
1	247	137	10
2	116	20	135

```
Classification Report:
                   precision    recall  f1-score   support

                0       0.74      0.97      0.84      1092
                1       0.75      0.35      0.48       394
                2       0.87      0.50      0.63       271

         accuracy                           0.76      1757
        macro avg       0.79      0.60      0.65      1757
     weighted avg       0.76      0.76      0.73      1757
```

After Smote:

```
In [16]: # Compute and print the F1 score
         f1 = f1_score(y_test, y_pred, average='weighted')
         print("F1 Score is : ", f1)

         F1 Score is :  0.7768118213369273

In [17]: # Evaluate the model's performance
         print("Accuracy is : ", accuracy_score(y_test, y_pred))

         Accuracy is :  0.7757541263517359

In [18]: classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.87      0.85      0.86      1124
                    1       0.59      0.61      0.60       366
                    2       0.66      0.67      0.66       267

             accuracy                           0.78      1757
            macro avg       0.71      0.71      0.71      1757
         weighted avg       0.78      0.78      0.78      1757
```

Support Vector Machine (SVM):

SVM is a supervised learning algorithm for classification and regression tasks. It finds a hyperplane that best separates classes in a high-dimensional space. SVM is effective in handling complex decision boundaries and is widely used in text classification, image recognition, and more.

Results with Vector Machine (SVM):

Before Smote:

```
In [31]: # Compute F1 score
         f1 = f1_score(y_test, y_pred, average='weighted')
         print("F1 Score is : ", f1)

         F1 Score is :  0.678515246660392

In [32]: # Print the accuracy score
         print("Accuracy Score is : ", accuracy_score(y_test, y_pred))

         Accuracy Score is :  0.6829823562891292

In [33]: # Print the classification report
         classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.76      0.82      0.79      1084
                    1       0.45      0.47      0.46       388
                    2       0.68      0.47      0.56       285

             accuracy                           0.68      1757
            macro avg       0.63      0.58      0.60      1757
         weighted avg       0.68      0.68      0.68      1757

In [34]: conf_matrix = confusion_matrix(y_test, y_pred)
         pd.DataFrame(conf_matrix)

Out[34]:
```

	0	1	2
0	885	162	37
1	182	181	25
2	91	60	134

After Smote:

```
In [19]: # Compute F1 score
         f1 = f1_score(y_test, y_pred, average='weighted')
         print("F1 Score is : ", f1)

         F1 Score is :  0.660824735919138

In [20]: # Print the accuracy score
         print("Accuracy Score is : ", accuracy_score(y_test, y_pred))

         Accuracy Score is :  0.6727376209447923

In [21]: # Print the classification report
         classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.75      0.84      0.79      1095
                    1       0.44      0.40      0.42       386
                    2       0.61      0.40      0.48       276

             accuracy                           0.67      1757
            macro avg       0.60      0.55      0.56      1757
         weighted avg       0.66      0.67      0.66      1757
```

Decision Tree:

A decision tree is a tree-like model where each node represents a decision based on the value of a specific feature. It recursively splits the data into subsets until a certain condition is met. Decision trees are interpretable and can capture complex relationships in the data.

Results with Decision Tree:

Before Smote:

	0	1	2
0	854	153	87
1	167	157	44
2	101	49	145

```
Classification Report:
                precision    recall  f1-score   support

             0       0.76      0.78      0.77      1094
             1       0.44      0.43      0.43       368
             2       0.53      0.49      0.51       295

      accuracy                           0.66      1757
     macro avg       0.57      0.57      0.57      1757
  weighted avg       0.65      0.66      0.66      1757
```

```
F1 Score: 0.6556504603212985

Accuracy : 0.6550939100739898
```

After Smote:

```
In [16]: # Compute F1 score
         f1 = f1_score(y_test, y_pred, average='weighted')  # You can change the average parameter based on your needs
         print("F1 Score is : ", f1)

         F1 Score is :  0.6984382757561852

In [17]: print("Accuracy is : ", accuracy_score(y_test, y_pred))

         Accuracy is :  0.6943653955606147

In [18]: classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.81      0.77      0.79      1139
                    1       0.46      0.50      0.48       360
                    2       0.58      0.62      0.60       258

             accuracy                           0.69      1757
            macro avg       0.62      0.63      0.62      1757
         weighted avg       0.70      0.69      0.70      1757
```

Random Forest:

Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (classification) or the mean prediction (regression) of the individual trees. It improves accuracy and reduces overfitting compared to a single decision tree.

Results with Random Forest:

Before Smote:

```
F1 Score is :  0.6671221044923381

Accuracy Score is :  0.7165623221400114
```

	0	1	2
0	1071	18	5
1	275	84	14
2	163	23	104

```
Classification Report:
                   precision    recall  f1-score   support

                0       0.71      0.98      0.82      1094
                1       0.67      0.23      0.34       373
                2       0.85      0.36      0.50       290

         accuracy                           0.72      1757
        macro avg       0.74      0.52      0.55      1757
     weighted avg       0.72      0.72      0.67      1757
```

After Smote:

```
In [16]: # Compute F1 score
         f1 = f1_score(y_test, y_pred, average='weighted')  # You can
         print("F1 Score is : ", f1)

         F1 Score is :  0.7150548233867742

In [17]: # Print the accuracy score
         print("Accuracy Score is : ", accuracy_score(y_test, y_pred))

         Accuracy Score is :  0.742743312464428

In [18]: # Print the classification report
         classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.76      0.94      0.84      1139
                    1       0.64      0.37      0.47       369
                    2       0.74      0.39      0.51       249

             accuracy                           0.74      1757
            macro avg       0.71      0.57      0.61      1757
         weighted avg       0.73      0.74      0.72      1757
```

Naive Bayes:

Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem. It assumes independence between features, hence the "naive" assumption. Despite its simplicity, Naive Bayes performs well in various applications, including text classification and spam filtering.

Results with Naive Bayes:

Before Smote:

```
F1 Score is : 0.4775176404407466

Accuracy is :  0.6175298804780877
```

	0	1	2
0	1077	3	0
1	382	8	0
2	286	1	0

Classification Report:

	precision	recall	f1-score	support
0	0.62	1.00	0.76	1080
1	0.67	0.02	0.04	390
2	0.00	0.00	0.00	287
accuracy			0.62	1757
macro avg	0.43	0.34	0.27	1757
weighted avg	0.53	0.62	0.48	1757

After Smote:

```
In [16]: # Compute F1 score
         f1 = f1_score(y_test, y_pred, average='weighted')  # You ca
         print("F1 Score is : ", f1)

         F1 Score is :  0.7625227812016376

In [17]: print("Acuuracy is : ", accuracy_score(y_test, y_pred))

         Acuuracy is :  0.7785998861696073

In [18]: classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.79      0.95      0.87      1076
                    1       0.67      0.45      0.54       376
                    2       0.81      0.57      0.67       305

             accuracy                           0.78      1757
            macro avg       0.76      0.66      0.69      1757
         weighted avg       0.77      0.78      0.76      1757
```

XGBoost (XGB):

XGBoost is an optimized gradient boosting algorithm designed for speed and performance. It sequentially adds weak learners (typically decision trees) to the ensemble, optimizing for both bias and variance. XGBoost is widely used for its efficiency and effectiveness in competitions and real-world applications.

Results with XGB:

Before Smote

```
F1 Score is :  0.741561538972695

Accuracy Score is :  0.7569721115537849
```

	0	1	2
0	1008	65	27
1	198	165	24
2	84	29	157

Classification Report:

	precision	recall	f1-score	support
0	0.78	0.92	0.84	1100
1	0.64	0.43	0.51	387
2	0.75	0.58	0.66	270
accuracy			0.76	1757
macro avg	0.72	0.64	0.67	1757
weighted avg	0.75	0.76	0.74	1757

After Smote:

```
In [13]: # Randomly select a portion of the data to turn into unlabeled
         percentage_to_turn_unlabeled = 0.4  # Adjust as needed
         num_samples_to_turn_unlabeled = int(len(df1) * percentage_to_turn_unlabeled)
         samples_to_turn_unlabeled = df1.sample(num_samples_to_turn_unlabeled)

In [14]: # Set the 'Label' column for the selected samples to NaN
         df1.loc[samples_to_turn_unlabeled.index, 'sentiment'] = None

         df1= df1[['cleaned_text','sentiment']]
         df1.head()
```

Logistic Regression:

Logistic Regression is a linear model for binary classification that predicts the probability of an instance belonging to a particular class. It uses the logistic function to map the output to a probability range between 0 and 1. Despite its name, logistic regression is a classification algorithm.

Results with Logistic Regression:

before Smote

	0	1	2
0	1053	37	20
1	193	164	11
2	98	25	156

Classification Report:

	precision	recall	f1-score	support
0	0.78	0.97	0.87	1100
1	0.78	0.42	0.55	376
2	0.88	0.60	0.71	281
accuracy			0.79	1757
macro avg	0.82	0.66	0.71	1757
weighted avg	0.80	0.79	0.77	1757

F1 Score of logistic regression is : 0.7641407840118125

Accuracy of logistic regression is : 0.7814456459874787

After Smote:

```
In [17]: # Compute and print the F1 score
         f1 = f1_score(y_test, y_pred, average='weighted')
         print("F1 Score is : ", f1)

         F1 Score is :  0.7986946998650918

In [18]: # Evaluate the model's performance
         print("Accuracy is : ", accuracy_score(y_test, y_pred))

         Accuracy is :  0.7973819009675583

In [19]: # Generate and print the classification report
         classification_rep = classification_report(y_test, y_pred)
         print("Classification Report:\n", classification_rep)

         Classification Report:
                       precision    recall  f1-score   support

                    0       0.87      0.87      0.87      1088
                    1       0.60      0.64      0.62       369
                    2       0.79      0.73      0.76       300

             accuracy                           0.80      1757
            macro avg       0.75      0.75      0.75      1757
         weighted avg       0.80      0.80      0.80      1757
```

when trying to print loss function, turns out self-train classifier does not have a history attributes :

```
In [9]: # Check if the model has a 'history' attribute
        if hasattr(model2, 'history'):
            history = model2.history
            if 'loss' in history:
                plt.plot(history['loss'])
                plt.title('Model Convergence')
                plt.xlabel('Epoch')
                plt.ylabel('Loss')
                plt.show()
            else:
                print("The 'history' attribute does not contain loss information.")
        else:
            print("The model does not have a 'history' attribute.")

        The model does not have a 'history' attribute.
```

Graph-based classifiers:

Rbf: The Radial Basis Function is a type of kernel function used in support vector machines (SVMs) and other machine learning models. The RBF kernel measures the

similarity between two data points in a high-dimensional space. For a pair of data points, the RBF kernel is defined as

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

where:

$\|x_i - x_j\|$ is the Euclidean distance between x_i and x_j,
σ is a parameter that controls the width of the kernel.

Label Spreading classifier:

Idea: Similar to Label Propagation, Label Spreading propagates labels through the graph. However, it uses a different approach to smooth label information across the graph.

How it works: It considers a fully connected graph and utilizes a diffusion process to spread label information.

Algorithm 11.3 Label spreading (Zhou et al., 2004)

Compute the affinity matrix $\mathbf{W}$ from (11.1) for $i \neq j$ (and $\mathbf{W}_{ii} \leftarrow 0$)
Compute the diagonal degree matrix $\mathbf{D}$ by $\mathbf{D}_{ii} \leftarrow \sum_j W_{ij}$
Compute the normalized graph Laplacian $\mathcal{L} \leftarrow \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}$
Initialize $\hat{Y}^{(0)} \leftarrow (y_1, \ldots, y_l, 0, 0, \ldots, 0)$
Choose a parameter $\alpha \in [0, 1)$
Iterate $\hat{Y}^{(t+1)} \leftarrow \alpha\mathcal{L}\hat{Y}^{(t)} + (1-\alpha)\hat{Y}^{(0)}$ until convergence to $\hat{Y}^{(\infty)}$
Label point x_i by the sign of $\hat{y}_i^{(\infty)}$

Results of Label Spreading classifier model:

Using “rbf”:

```
Accuracy is :  0.7148548662492885

F1 Score is :  0.7063275698677198
```

	0	1	2
0	942	106	57
1	162	144	53
2	86	37	170

Classification Report:

	precision	recall	f1-score	support
0	0.80	0.84	0.82	1114
1	0.54	0.44	0.49	368
2	0.63	0.66	0.65	275
accuracy			0.73	1757
macro avg	0.66	0.65	0.65	1757
weighted avg	0.72	0.73	0.72	1757

Using “Knn”:

Accuracy is : 0.7148548662492885

F1 Score is : 0.7245158089943451

	0	1	2
0	942	106	57
1	162	144	53
2	86	37	170

Classification Report:

	precision	recall	f1-score	support
0	0.80	0.84	0.82	1114
1	0.54	0.44	0.49	368
2	0.63	0.66	0.65	275
accuracy			0.73	1757
macro avg	0.66	0.65	0.65	1757
weighted avg	0.72	0.73	0.72	1757

Label Propagation classifier:

Idea: Propagate labels from labeled data points to their neighbors in the graph.

How it works: Initially, a small subset of data points is labeled, and the labels are propagated to unlabeled points based on the graph structure. This propagation continues until a steady state is reached.

Algorithm 11.1 Label propagation (Zhu and Ghahramani, 2002)

Compute affinity matrix $\mathbf{W}$ from (11.1)
Compute the diagonal degree matrix $\mathbf{D}$ by $\mathbf{D}_{ii} \leftarrow \sum_j W_{ij}$
Initialize $\hat{Y}^{(0)} \leftarrow (y_1, \ldots, y_l, 0, 0, \ldots, 0)$
Iterate
1. $\hat{Y}^{(t+1)} \leftarrow \mathbf{D}^{-1}\mathbf{W}\hat{Y}^{(t)}$
2. $\hat{Y}_l^{(t+1)} \leftarrow Y_l$

until convergence to $\hat{Y}^{(\infty)}$
Label point x_i by the sign of $\hat{y}_i^{(\infty)}$

Results of Label Propagation classifier model:

Using “rbf”:

```
Accuracy is :  0.6983494593056346
F1 Score is :  0.7245158089943451
```

confusion matrix:

	0	1	2
0	869	146	99
1	144	159	50
2	59	63	168

```
Classification Report:
              precision    recall  f1-score   support

           0       0.78      0.79      0.79      1077
           1       0.47      0.41      0.44       391
           2       0.55      0.63      0.59       289

    accuracy                           0.68      1757
   macro avg       0.60      0.61      0.60      1757
weighted avg       0.67      0.68      0.68      1757
```

Using “Knn”:

```
F1 Score is :  0.6940582659752538

Accuracy is :  0.6983494593056346
```

	0	1	2
0	876	126	75
1	170	170	51
2	69	39	181

```
Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.81      0.80      1077
           1       0.51      0.43      0.47       391
           2       0.59      0.63      0.61       289

    accuracy                           0.70      1757
   macro avg       0.63      0.62      0.62      1757
weighted avg       0.69      0.70      0.69      1757
```

Best Model

Logistic regression is the best model with an f1-score of 79.8%, so we export it:

```
joblib.dump(model1, 'BestModel.joblib')
```

Advantages

Interpretability: Logistic regression provides easy interpretability of model coefficients, which can be crucial for understanding the impact of different features on the classification decision.

Efficiency: Logistic regression tends to be computationally efficient and can handle large datasets with a relatively low computational cost.

Robustness to Irrelevant Features: Logistic regression is less prone to overfitting when dealing with a high number of irrelevant features compared to more complex models.

Probabilistic Output: Logistic regression provides probabilities as output, making it easy to set a threshold for classification and interpret the confidence of predictions.

Inconvenients

Linear Decision Boundary: The assumption of a linear decision boundary may limit the model's ability to capture complex relationships in the text data. If the underlying patterns are nonlinear, more complex models may offer better performance.

Limited Expressiveness: Text data often contains intricate relationships and semantics that may not be fully captured by logistic regression's linear model. More sophisticated models, such as ensemble methods or deep learning, could potentially offer better expressiveness.

Handling Unbalanced Data: Logistic regression may face challenges in handling unbalanced data, especially if one class significantly outnumbers the other.

Techniques like resampling, adjusting class weights, or exploring different evaluation metrics may be necessary to address this imbalance effectively.

Deployment

1. Model Deployment:

- Exporting the Best Model: The best-performing sentiment analysis model, trained using semi-supervised learning and TF-IDF vectorization, is exported using the serialization technique (joblib).
- Flask Integration: The exported model is integrated into a Flask application, allowing it to be used for real-time predictions.

2. Flask Application Structure:

- The Flask application comprises two main pages - "Home" and "Predict". The "Home" page features a text input form, and the "Predict" page displays the predicted sentiment.

3. Model Implementation in Flask:

- Model Loading: In the Flask application, the exported model is loaded during the application startup.
- Text Processing: When a user submits text through the form, the input text undergoes the same preprocessing steps applied during model training.
- TF-IDF Vectorization: The preprocessed text is vectorized using the TF-IDF vectorizer, ensuring compatibility with the model's input requirements.
- Sentiment Prediction: The vectorized text is then passed through the loaded sentiment analysis model, and the predicted sentiment is obtained.
- Result Presentation: The predicted sentiment is presented on the "Predict" page, offering users immediate feedback on the sentiment of the entered text.

Conclusion

Semi-supervised text classification presents a versatile and pragmatic approach to text categorization, especially in scenarios where labeled data is scarce. The selection of a specific method depends on the nature of the text data and the characteristics of the

classification task. The integration of unlabeled data enriches the training process, enhancing the model's ability to handle diverse and dynamic textual content.

References

1 - dataset: twitter-us-airline-sentiment from data.world website.

2 - a review on sentiment analysis and emotion detection from text.

3 - self-training paper by Massih-Reza Amini and Vasilii Feofanov.

4 - a paper about Learning from labeled and Unlabeled Data with Label Propagation by Xiaojin Zhu and Zoubin Ghahramani.

5 - Semi-Supervised learning Book by ‘Olivier Chapelle’, ’Bernhard scholkopf’ and ‘Alexander Zien’ .