1 Introduction

In today's financial landscape, access to timely and meaningful information is vital for making well-informed decisions. The world of finance is flooded with news articles and reports, each holding potential insights into economic trends and market sentiment. This project focuses on using supervised Machine Learning (ML) methods to systematically analyze sentiments in financial texts.

In chapter 2 I discuss the data structure that is used in this project. I also discuss the pre-processes that need to be done to the data before training and modeling. In chapter 3 I discuss the selected features. In chapter 4 I look at the used ML methods. And in chapter 5 I conclude the project and analyze the results.

2 Data overview and preprocessing

The data used in this project contains headlines for different financial articles. Each headline is labeled to three different categories: positive, neutral and negative. The data is obtained from *Kaggle.com* from a public dataset called *Sentiment Analysis for Financial News* [1]. It is a ML platform that contains multiple different public datasets for practicing and research.

Before analysis, the collected textual data undergoes preprocessing. This includes common preprocessing tasks that are commonly used in the field of natural language processing such as tokenization, stemming, punctuation and stop word removal [2] [3] [4]. To simplify the analysis, I drop off the ‘neutral’ column as I’m only trying to distinguish between positive and negative sentiments. That makes the data binary and allows me to use binary ML models.

The data contains 4837 data points and it is split into 2 different sections; into training and test sets. The ratio in which they are split follows the same-kind of ratio as discussed during the course. I am using *train_test_split* from the library *sklearn.model_selection* and use the *shuffle* mode on. 20% of the dataset will be reserved for testing and the rest will be used for training purposes. The data is quite unbalanced as there are 1363 positive sentiments and 604 negative sentiments. Due to this I balance the data using oversampling to match the amount of negative sentiments to match to the amount of positive sentiments.

In addition to oversampling, I also use the k-fold cross validation with 10 folds to improve the suitability of the training and validation data. I use the *StratifiedKFold* from the library *sklearn.model_selection*. It is particularly useful when dealing with imbalanced datasets, where one class has significantly fewer samples than another. It ensures that each fold has a similar proportion of examples of each class as the whole dataset. The reason for the use of both oversampling and k-fold cross validation is that I compare how the models perform when using either of these techniques.

3 Feature selection

I have to use some feature engineering in order to extract features out of the text. I use a concept called Bag of Words (BoW) to represent the text data as a matrix of word frequencies, where each row corresponds to a document, and each column corresponds to a unique word in the entire corpus. This

allows me to convert the unstructured text into a format suitable for ML models, enabling them to analyze and classify the text data based on these extracted features.

I also use a technique called TF-IDF (Term Frequency-Inverse Document Frequency) that measures the frequency of terms within a document and its repetition in the whole data. It helps in identifying the significance of terms. Terms that appear more frequently within a document are given higher TF values. So in practice, both of these techniques allow me to transform the textual data into a numerical format that I can analyze with the ML models.

In addition to using Bag of Words (BoW) and TF-IDF, the process of feature selection involves domain-specific knowledge as I am using custom stopwords for financial headlines, certain common words may not provide useful information. These stop words were selected by manually inspecting the data, aiming to exclude words that occur often that don't possess relevant information for the decision if the headline is positive or negative.

4 Methods

4.1 Multinomial Naive Bayes

Multinomial Naive Bayes (NB) is a popular choice for text classification tasks like sentiment analysis. That is why the first model I am using is Naive Bayes. It is a simple yet effective algorithm for text classification. It's computationally efficient and works well even on large datasets. This makes it a practical choice, especially for tasks involving text analysis. [6]

As a loss function, I use logistic loss that is also sometimes called binary cross entropy or cross entropy loss. I choose it because it is usually the default loss function and very used in binary probability problems and only to be changed to another if problems occurred when using it. I am not using for example hinge loss, because it is more suited for Support Vector Machines (SVM). [7]

4.2 Logistic Regression

Logistic Regression (LR) is another commonly used model in language processing. I chose to use this as the second model because it is a good fit for this problem that requires a binary solution. It has also performed well in comparison between different ML models when evaluating short texts. [8] [9] That’s why it's an excellent choice for the second ML method.

As a loss function I use logistic loss simply because it is what LR uses and also because the implementation of LR of the library *sklearn*, that I am intending to use, uses logistic loss as its default loss function. [10] [11]

5 Model evaluation & conclusion

I do the modeling on two different models first without k-folding. I use class balancing by oversampling the minority class (negative sentiments) to match the majority class (positive sentiments) Both of the chosen models are evaluated with the following metrics: accuracy, precision, recall and f1-score.

After the use of only oversampling, I introduce the use of k-fold cross-validation using the *StratifiedKFold* technique. The purpose is to determine how well the models generalize to unseen data when the dataset remains unbalanced. Again, similar as with the use of only oversampling, I explore the impact of different combinations of classifiers and vectorizers, but now with cross-validation.

5.1 Multinomial Naive Bayes with oversampling

I evaluate the performance of Multinomial Naive Bayes with BoW and TF-IDF representations for both the original and resampled training data. The key findings are that, using the resampled training data significantly improves accuracy, precision, recall, and F1-score for both BoW and TF-IDF. Also, BoW generally performs better than TF-IDF, especially when using the resampled data.

The exact results are for the BoW representation: accuracy for original data 0.8401 and for resampled data 0.8680, precision for original data positive (0.87), negative (0.77) and for resampled data positive (0.97), negative (0.73). The recall scores in the same order are: positive (0.89), negative (0.73) and positive (0.83), negative (0.95) and the f1-scores in the same order are: positive (0.88), negative (0.75) and positive (0.89), negative (0.83).

The exact results are for the TF-IDF representation: accuracy for original data 0.7284 and for resampled data 0.8883, precision for original data positive (0.71), negative (0.96) and for resampled data positive (0.97), negative (0.77). The recall scores in the same order are: positive (1.00), negative (0.19) and positive (0.86), negative (0.95) and the f1-scores in the same order are: positive (0.83), negative (0.32) and positive (0.91), negative (0.85).

5.2 Logistic Regression with oversampling

I also assess the Logistic Regression model's performance with BoW and TF-IDF representations for both the original and resampled training data. Here, the main findings are that Logistic Regression consistently outperforms Naive Bayes in terms of accuracy and F1-score and that the resampled training data leads to better model performance across the board, with Logistic Regression displaying particularly strong results.

The exact results are for the BoW representation: accuracy for original data 0.8350 and for resampled data 0.9492, precision for original data positive (0.83), negative (0.85) and for resampled data positive (0.97), negative (0.92). The recall scores in the same order are: positive (0.95), negative (0.61) and positive (0.96), negative (0.93) and the f1-scores in the same order are: positive (0.88), negative (0.71) and positive (0.96), negative (0.92).

The exact results are for the TF-IDF representation: accuracy for original data 0.7741and for resampled data 0.9442, precision for original data positive (0.76), negative (0.85) and for resampled data positive (0.96), negative (0.91). The recall scores in the same order are: positive (0.97), negative (0.39) and positive (0.95), negative (0.92) and the f1-scores in the same order are: positive (0.85), negative (0.53) and positive (0.96), negative (0.92).

5.3 Multinomial Naive Bayes & Logistic regression with k-folding

I examine the mean accuracy, precision, recall, and F1-score across k-folds for Multinomial Naive Bayes and Logistic Regression models with both BoW and TF-IDF representations. The main findings are that Logistic Regression consistently outperforms Multinomial Naive Bayes across all configurations. Also, it appears that the choice of vectorizer has a significant impact on model performance, with Count Vectorization often producing better results. Before all, it seems that using k-fold cross-validation helps ensure the models generalize well, even with class imbalance.

Here are the exact results. NB with CountVectorizer: mean accuracy: 0.8226, mean precision: 0.8334, mean recall: 0.8226 and mean F1-score: 0.8254. NB with TfidfVectorizer: mean accuracy: 0.7793, mean precision: 0.8174, mean recall: 0.7793 and mean F1-score: 0.7362. LR with CountVectorizer: mean accuracy: 0.8455, mean precision: 0.8432, mean recall: 0.8455 and mean F1-score: 0.8400. LR with TfidfVectorizer: mean accuracy: 0.8109, mean precision: 0.8188, mean recall: 0.8109 and mean F1-score: 0.7909.

5.4 Conclusion

This report outlines the application of machine learning techniques to address the problem of classifying financial headlines as either positive or negative, a task crucial for sentiment analysis in the domain of finance. The features considered for classification encompass various natural language processing and text analysis methods. Data preprocessing, including text tokenization, removal of stopwords, stemming, and vectorization, is performed to prepare the textual data for model training. The dataset is then divided into training and testing subsets for model evaluation. Four different models are employed: NB with both Count Vectorization and TF-IDF Vectorization, as well as LR with BoW and TF-IDF Vectorization.

The results of the model evaluation indicate that the LR model with Count Vectorization and k-folding produces the highest accuracy and F1-Score, making it the preferred choice for this problem. Nevertheless, these results open up discussions for potential improvements. The achieved accuracy of approximately 84.55% is respectable, but there is still room for enhancement. In practical, real-world financial applications, achieving a nearly perfect classification accuracy can be challenging due to the complexity and subjectivity of financial language. Collecting more diverse data, including a wider range of financial topics and sources, as well as addressing factors such as different writing styles, could potentially lead to better model generalization.

Furthermore, the analysis reveals that different machine learning algorithms and techniques could be explored to enhance robustness against noisy data. Financial news often contains diverse elements beyond sentiment, such as numerical data, facts, and industry-specific jargon, which might require more advanced feature engineering and modeling approaches.

References

1. Kaggle. Sentiment Analysis for Financial News. https://www.kaggle.com/datasets/ankurzing/sentiment-analysis-for-financial-news. Accessed [21.09.2023].

2. Kumar KK, Harish BS. Classification of Short Text Using Various Preprocessing Techniques: An Empirical Evaluation. In: Sa P, Bakshi S, Hatzilygeroudis I, Sahoo M, editors. Recent Findings in Intelligent Computing Techniques. Advances in Intelligent Systems and Computing, vol 709. Springer; 2018. https://doi.org/10.1007/978-981-10-8633-5_3

3. Srividhya V, Anitha R. Evaluating preprocessing techniques in text categorization. International journal of computer science and application. 2010 Apr;47(11):49-51.

4. Kadhim AI. An evaluation of preprocessing techniques for text classification. International Journal of Computer Science and Information Security (IJCSIS). 2018 Jun;16(6):22-32.

5. Hung Lai, Rayner Alfred, Mohd Hijazi. A Review on Feature Selection Methods for Sentiment Analysis. Advanced Science Letters. 2015; 21:2952-2956. DOI: 10.1166/asl.2015.6475.

6. Wei Zhang, Feng Gao. An Improvement to Naive Bayes for Text Classification. Procedia Engineering. 2011; 15: 2160-2164. ISSN 1877-7058. doi: 10.1016/j.proeng.2011.08.404.

7. Brownlee J. How to Choose Loss Functions When Training Deep Learning Neural Networks. [https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learningneural-networks/]. Accessed [21.09.2023].

8. Aliman GB, Nivera TF, Olazo JC, Ramos DJ, Sanchez CD, Amado TM, Arago NM, Jorda Jr RL, Virrey GC, Valenzuela IC. Sentiment Analysis using Logistic Regression. Journal of Computational Innovations and Engineering Applications JULY. 2022;35:40.

9. Nguyen H, Veluchamy A, Diop M, Iqbal R. Comparative study of sentiment analysis with product reviews using machine learning and lexicon-based approaches. SMU Data Science Review. 2018;1(4):7.

10. Setia M. Binary Cross Entropy aka Log Loss - The cost function used in Logistic Regression. [https://www.analyticsvidhya.com/blog/2020/11/binary-cross-entropy-aka-log-loss-the-cost-function-used-in-logistic-regression/]. Accessed [21.09.2023].

11. sklearn.linear_model.LogisticRegression. scikit-learn documentation. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html. Accessed [21.09.2023].

Appendices

October 11, 2023

[25]:
```
# Import all dependencies
import pandas as pd
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪precision_recall_fscore_support
from sklearn.linear_model import LogisticRegression
from sklearn.utils import resample

import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

# Import the necessary libraries for Naive Bayes, Count Vectorization (BoW) and␣
 ↪TfidfVectorizer (TD-IDF)
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

nltk.download('stopwords')
nltk.download('punkt')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/suonieo1/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /home/suonieo1/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

[25]:
```
True
```

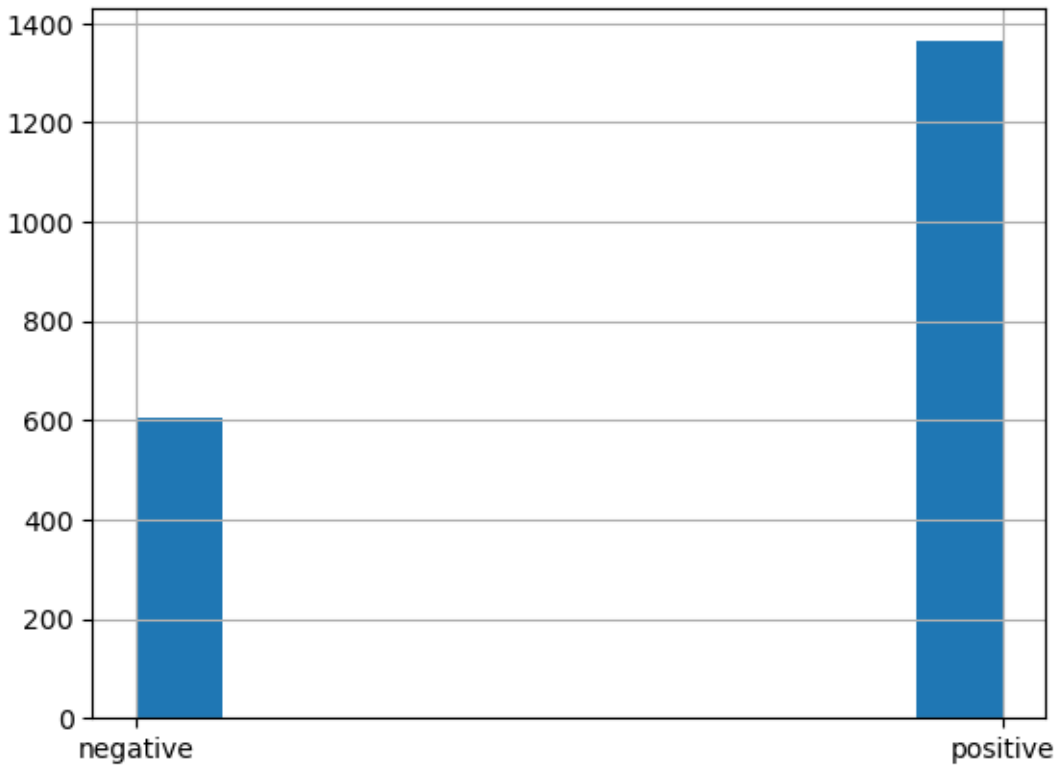
0.0.1 Data introduction

Here we read the data from the file and rename the columns to “Text” and “Sentiment”. Then we drop off the column ‘neutral’.

[26]:
```
df = pd.read_csv('data.csv', delimiter=',', encoding='latin-1', header=None)
df = df.rename(columns={0: 'Sentiment', 1: 'Text'})
df = df[['Text', 'Sentiment']]

df = df[df['Sentiment'] != 'neutral']
```

```
print("Shape of data:")
print(df.shape)

print()
print("Head:")
print(df.head())

print()

sentiment_counts = df['Sentiment'].value_counts()

# Access the counts for 'positive' and 'negative'
positive_count = sentiment_counts['positive']
negative_count = sentiment_counts['negative']

print("Positive Sentiments:", positive_count)
print("Negative Sentiments:", negative_count)

print()

print(df['Sentiment'].hist())
```

```
Shape of data:
(1967, 2)

Head:
                                                 Text Sentiment
2  The international electronic industry company …  negative
3  With the new production plant the company woul…  positive
4  According to the company 's updated strategy f…  positive
5  FINANCING OF ASPOCOMP 'S GROWTH Aspocomp is ag…  positive
6  For the last quarter of 2010 , Componenta 's n…  positive

Positive Sentiments: 1363
Negative Sentiments: 604

Axes(0.125,0.11;0.775x0.77)
```



0.0.2 Data preprocessing

Here we preprocess the data. We tokenize, remove punctuations and stopwords and stem the words. Then we split the data into training and testing sets with 80/20 split.

[27]:
```
# Create the preprocessing function
def preprocess_text(text, custom_stopwords=None):
    # Tokenization (split the text into words)
    words = nltk.word_tokenize(text)

    # Remove punctuation and convert to lowercase
    words = [word.lower() for word in words if word.isalpha()]

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word not in stop_words]

    # Remove custom stopwords
    if custom_stopwords:
        words = [word for word in words if word not in custom_stopwords]

    # Stemming (reducing words to their root form)
```

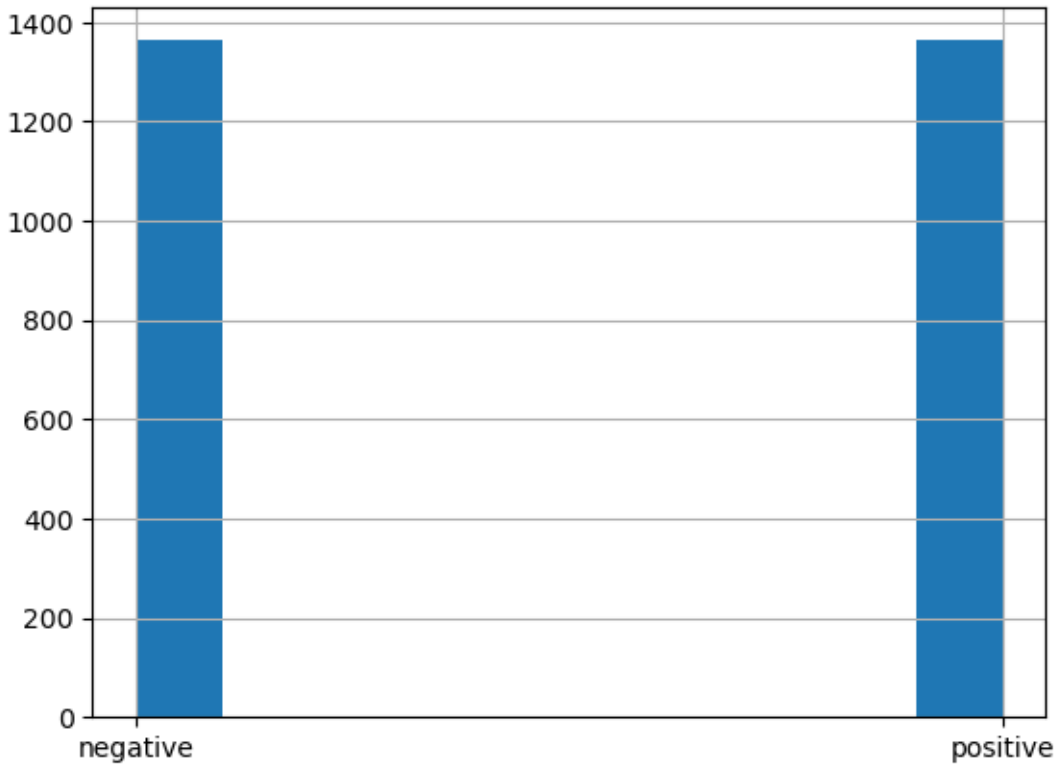
```
    stemmer = PorterStemmer()
    words = [stemmer.stem(word) for word in words]

    # Join the cleaned words back into a single string
    cleaned_text = ' '.join(words)

    return cleaned_text

custom_stopwords = [
    "company", "stock", "€", "$", "Apple", "Samsung", "Kone", "Nokia", "UPM",␣
 ↪"Sampo"
]

# Apply preprocessing to each text in the data
df['Text'] = df['Text'].apply(lambda x: preprocess_text(x, custom_stopwords))

X = df['Text']
y = df['Sentiment']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
 ↪shuffle=True, random_state=42)
```

As we can see from the histogram above that the data is quite unbalanced. Because of this, we are going to oversample the negative class to match the majority class (the positive class).

[28]:
```
# Split the training data into 'positive' and 'negative' classes
positive_class = df[df['Sentiment'] == 'positive']
negative_class = df[df['Sentiment'] == 'negative']

# Check the class with the majority of samples
if len(positive_class) > len(negative_class):
    majority_class = positive_class
    minority_class = negative_class
else:
    majority_class = negative_class
    minority_class = positive_class

# Oversample the minority class to match the majority class
minority_class_resampled = resample(minority_class, replace=True,␣
 ↪n_samples=len(majority_class), random_state=42)

# Concatenate the resampled minority class with the majority class
train_data_resampled = pd.concat([majority_class, minority_class_resampled])

# Shuffle the resampled data
train_data_resampled = train_data_resampled.sample(frac=1, random_state=42)

# Split the data into X and y
```

```
X_train_resampled = train_data_resampled['Text']
y_train_resampled = train_data_resampled['Sentiment']

print("shape of resampled data:")
print(X_train_resampled.shape)
print(y_train_resampled.shape)

print(train_data_resampled['Sentiment'].hist())
```

```
shape of resampled data:
(2726,)
(2726,)
Axes(0.125,0.11;0.775x0.77)
```



Now we can see, that the data is balanced.

0.0.3 Feature engineering

Then we feature engineer our data. We first initialize Count Vectorizer to convert text data into a Bag of Words (BoW) representation. The the vectorizer will tokenize the text data, create a vocabulary of up to 10,000 unique words (features), and count the occurrences of each word in the text.

Then we fit the vectorizer to the original training data X_train using the fit_transform method. This step creates a vocabulary based on the text in X_train and transforms X_train into a matrix where each row represents a document (text) from the training data, and each column represents a word from the vocabulary. The values in this matrix are the counts of how many times each word appears in each document.

Then we transform the resampled training data into the BoW representation using the same vectorizer. This is necessary to ensure that the BoW representation of both the original and resampled data are consistent.

Finally we also transform the original test data X_test into the BoW representation.

[29]:
```
# Initialize the Count Vectorizer for BoW
count_vectorizer = CountVectorizer(max_features=10000)

# Apply Count Vectorization to the original training data
X_train_bow = count_vectorizer.fit_transform(X_train)
# Apply Count Vectorization to the resampled training data
X_train_bow_resampled = count_vectorizer.transform(X_train_resampled)

# Apply Count Vectorization to the original test data
X_test_bow = count_vectorizer.transform(X_test)
```

Here we create an instance of TfidfVectorizer with a maximum of 10,000 features. This means that the vectorizer will tokenize the text data, create a vocabulary of up to 10,000 unique words (features), and compute the TF-IDF values for each word in the text.

Then we fit the vectorizer to the original training data X_train using the fit_transform method. This step creates a vocabulary based on the text in X_train and transforms X_train into a matrix where each row represents a document (text) from the training data, and each column represents a word from the vocabulary. The values in this matrix are TF-IDF values for each word in each document.

Then we transform the resampled training data X_train_resampled into the TF-IDF representation using the same vectorizer.

Finally we also transform the original test data X_test into the TF-IDF representation.

[30]:
```
# Initialize the TF-IDF Vectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=10000)

# Apply TF-IDF Vectorization to the original training data
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
# Apply TF-IDF Vectorization to the resampled training data
X_train_tfidf_resampled = tfidf_vectorizer.transform(X_train_resampled)

# Apply TF-IDF Vectorization to the original test data
X_test_tfidf = tfidf_vectorizer.transform(X_test)
```

0.0.4 Modelling

Here we perform text classification using a Multinomial Naive Bayes (NB) classifier on two different versions of the data: one with the original training data, and one with the resampled training data.

We create two instances of the Multinomial Naive Bayes classifier. One is for the original Bag of Words (BoW) transformed training data, and the other is for the BoW-transformed resampled training data.

Then we train the models and predict the labels.

Finally we evaluate the model performances and print out metrics for both models to compare them.

[31]:
```
# Initialize a Multinomial Naive Bayes classifier for BoW
nb_model_bow = MultinomialNB()
# Initialize a Multinomial Naive Bayes classifier for BoW (for resmapled data)
nb_model_bow_resampled = MultinomialNB()


# Train the Naive Bayes model on the BoW-transformed training data
nb_model_bow.fit(X_train_bow, y_train)
# Train the Naive Bayes model on the BoW-transformed resampled training data
nb_model_bow_resampled.fit(X_train_bow_resampled, y_train_resampled)


# Predict the labels on the original test data (BoW)
y_test_pred_nb_bow = nb_model_bow.predict(X_test_bow)
# Predict the labels on the resmapled test data (BoW)
y_test_pred_nb_bow_resampled = nb_model_bow_resampled.predict(X_test_bow)


# Evaluate the Naive Bayes model's performance on the original test data (BoW)
print("Original Test Results (Naive Bayes with BoW):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_nb_bow))
print(classification_report(y_test, y_test_pred_nb_bow))


# Evaluate the Naive Bayes model's performance on the resmapled test data (BoW)
print("Resmapled Test Results (Naive Bayes with BoW):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_nb_bow_resampled))
print(classification_report(y_test, y_test_pred_nb_bow_resampled))
```

```
Original Test Results (Naive Bayes with BoW):
Accuracy: 0.8401015228426396
              precision    recall  f1-score   support

    negative       0.77      0.73      0.75       131
    positive       0.87      0.89      0.88       263
```

```
    accuracy                           0.84       394
   macro avg       0.82      0.81      0.82       394
weighted avg       0.84      0.84      0.84       394

Resmapled Test Results (Naive Bayes with BoW):
Accuracy: 0.868020304568528
              precision    recall  f1-score   support

    negative       0.73      0.95      0.83       131
    positive       0.97      0.83      0.89       263

    accuracy                           0.87       394
   macro avg       0.85      0.89      0.86       394
weighted avg       0.89      0.87      0.87       394
```

We follow the same tradition as above for each cases:

- Multinomial Naive Bayes classifier for TF-IDF
- Multinomial Naive Bayes classifier for TF-IDF (for resmapled data)
- Logistic Regression model for BoW
- Logistic Regression model for BoW (for resampled data)
- Logistic Regression model for TF-IDF
- Logistic Regression model for TF-IDF (for resampled data)

[32]:
```
# Initialize a Multinomial Naive Bayes classifier for TF-IDF
nb_model_tfidf = MultinomialNB()
# Initialize a Multinomial Naive Bayes classifier for TF-IDF (for resmapled␣
 ↪data)
nb_model_tfidf_resampled = MultinomialNB()


# Train the Naive Bayes model on the TF-IDF-transformed training data
nb_model_tfidf.fit(X_train_tfidf, y_train)
# Train the Naive Bayes model on the TF-IDF-transformed resampled training data
nb_model_tfidf_resampled.fit(X_train_tfidf_resampled, y_train_resampled)

# Predict the labels on the original test data (TF-IDF)
y_test_pred_nb_tfidf = nb_model_tfidf.predict(X_test_tfidf)
# Predict the labels on the resmapled test data (TF-IDF)
y_test_pred_nb_tfidf_resampled = nb_model_tfidf_resampled.predict(X_test_tfidf)

# Evaluate the Naive Bayes model's performance on the original test data␣
 ↪(TF-IDF)
print("Original Test Results (Naive Bayes with TF-IDF):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_nb_tfidf))
print(classification_report(y_test, y_test_pred_nb_tfidf))
```

```
# Evaluate the Naive Bayes model's performance on the resmapled test data␣
 ↪(TF-IDF)
print("Resmapled Test Results (Naive Bayes with TF-IDF):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_nb_tfidf_resampled))
print(classification_report(y_test, y_test_pred_nb_tfidf_resampled))
```

```
Original Test Results (Naive Bayes with TF-IDF):
Accuracy: 0.7284263959390863
              precision    recall  f1-score   support

    negative       0.96      0.19      0.32       131
    positive       0.71      1.00      0.83       263

    accuracy                           0.73       394
   macro avg       0.84      0.59      0.57       394
weighted avg       0.79      0.73      0.66       394

Resmapled Test Results (Naive Bayes with TF-IDF):
Accuracy: 0.8883248730964467
              precision    recall  f1-score   support

    negative       0.77      0.95      0.85       131
    positive       0.97      0.86      0.91       263

    accuracy                           0.89       394
   macro avg       0.87      0.90      0.88       394
weighted avg       0.90      0.89      0.89       394
```

[33]:
```
# Initialize a Logistic Regression model for BoW
lr_model_bow = LogisticRegression()
# Initialize a Logistic Regression model for BoW (for resmapled data)
lr_model_bow_resampled = LogisticRegression()


# Train the Logistic Regression model on the BoW-transformed training data
lr_model_bow.fit(X_train_bow, y_train)
# Train the Logistic Regression model on the BoW-transformed resampled training␣
 ↪data
lr_model_bow_resampled.fit(X_train_bow_resampled, y_train_resampled)


# Predict the labels on the original test data (BoW)
y_test_pred_lr_bow = lr_model_bow.predict(X_test_bow)
# Predict the labels on the resmapled test data (BoW)
y_test_pred_lr_bow_resampled = lr_model_bow_resampled.predict(X_test_bow)
```

```
# Evaluate the Logistic Regression model's performance on the original test␣
 ↪data (BoW)
print("Original Test Results (Logistic Regression with BoW):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_lr_bow))
print(classification_report(y_test, y_test_pred_lr_bow))


# Evaluate the Logistic Regression model's performance on the resmapled test␣
 ↪data (BoW)
print("Resmapled Test Results (Logistic Regression with BoW):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_lr_bow_resampled))
print(classification_report(y_test, y_test_pred_lr_bow_resampled))
```

```
Original Test Results (Logistic Regression with BoW):
Accuracy: 0.8350253807106599
              precision    recall  f1-score   support

    negative       0.85      0.61      0.71       131
    positive       0.83      0.95      0.88       263

    accuracy                           0.84       394
   macro avg       0.84      0.78      0.80       394
weighted avg       0.84      0.84      0.83       394

Resmapled Test Results (Logistic Regression with BoW):
Accuracy: 0.949238578680203
              precision    recall  f1-score   support

    negative       0.92      0.93      0.92       131
    positive       0.97      0.96      0.96       263

    accuracy                           0.95       394
   macro avg       0.94      0.94      0.94       394
weighted avg       0.95      0.95      0.95       394
```

[34]:
```
# Initialize a Logistic Regression model for TF-IDF
lr_model_tfidf = LogisticRegression()
# Initialize a Logistic Regression model for TF-IDF (for resmapled data)
lr_model_tfidf_resampled = LogisticRegression()

# Train the Logistic Regression model on the TF-IDF-transformed training data
lr_model_tfidf.fit(X_train_tfidf, y_train)
# Train the Logistic Regression model on the TF-IDF-transformed resampled␣
 ↪training data
lr_model_tfidf_resampled.fit(X_train_tfidf_resampled, y_train_resampled)
```

```
# Predict the labels on the original test data (TF-IDF)
y_test_pred_lr_tfidf = lr_model_tfidf.predict(X_test_tfidf)
# Predict the labels on the resmapled test data (TF-IDF)
y_test_pred_lr_tfidf_resampled = lr_model_tfidf_resampled.predict(X_test_tfidf)


# Evaluate the Logistic Regression model's performance on the original test␣
↪data (TF-IDF)
print("Original Test Results (Logistic Regression with TF-IDF):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_lr_tfidf))
print(classification_report(y_test, y_test_pred_lr_tfidf))


# Evaluate the Logistic Regression model's performance on the resmapled test␣
↪data (TF-IDF)
print("Resmapled Test Results (Logistic Regression with TF-IDF):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_lr_tfidf_resampled))
print(classification_report(y_test, y_test_pred_lr_tfidf_resampled))
```

```
Original Test Results (Logistic Regression with TF-IDF):
Accuracy: 0.7741116751269036
              precision    recall  f1-score   support

    negative       0.85      0.39      0.53       131
    positive       0.76      0.97      0.85       263

    accuracy                           0.77       394
   macro avg       0.81      0.68      0.69       394
weighted avg       0.79      0.77      0.75       394

Resmapled Test Results (Logistic Regression with TF-IDF):
Accuracy: 0.9441624365482234
              precision    recall  f1-score   support

    negative       0.91      0.92      0.92       131
    positive       0.96      0.95      0.96       263

    accuracy                           0.94       394
   macro avg       0.94      0.94      0.94       394
weighted avg       0.94      0.94      0.94       394
```

0.0.5 Feature engineering vol 2.

In the above code we didn't use kfolding, but instead we oversampled the minority class to match the majority class.

This time we are going to use the StratifiedKFold cross-validator to split the data into k-folds while preserving the distribution of classes. By doing this, we are leaving the data unbalanced to see how the use of k-folds compares to oversampling.

[35]:
```
count_vectorizer = CountVectorizer(max_features=10000)

X_train_bow = count_vectorizer.fit_transform(X)

X_test_bow = count_vectorizer.transform(X)

tfidf_vectorizer = TfidfVectorizer(max_features=10000)

X_train_tfidf = tfidf_vectorizer.fit_transform(X)

X_test_tfidf = tfidf_vectorizer.transform(X)

# Define the number of folds (k)
n_splits = 10

kf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)
```

0.0.6 Modelling vol 2.

Here we are using a nested loop to iterate over each combination of classifier and vectorizer. For each combination of classifier and vectorizer, we perform k-fold cross-validation.

Then for each fold we split the data into training and testing sets, vectorize the text data, fit the model, predict the labels and calculate and evaluate metrics.

[36]:
```
# Define the classifiers and vectorizers
classifiers = [MultinomialNB(), LogisticRegression()]
vectorizers = [count_vectorizer, tfidf_vectorizer]

# Iterate over classifiers and vectorizers
for classifier in classifiers:
    for vectorizer in vectorizers:
        print(f"Classifier: {classifier.__class__.__name__}, Vectorizer:␣
 ↪{vectorizer.__class__.__name__}")

        # Initialize lists to store evaluation metrics for each fold
        accuracy_scores = []
        precision_scores = []
        recall_scores = []
        f1_scores = []

        for train_index, test_index in kf.split(X, y):
            X_train_fold, X_test_fold = X.iloc[train_index], X.iloc[test_index]
            y_train_fold, y_test_fold = y.iloc[train_index], y.iloc[test_index]
```

```
            X_train_vectorized = vectorizer.transform(X_train_fold)
            X_test_vectorized = vectorizer.transform(X_test_fold)

            model = classifier
            model.fit(X_train_vectorized, y_train_fold)

            y_test_pred = model.predict(X_test_vectorized)

            accuracy = accuracy_score(y_test_fold, y_test_pred)
            precision, recall, f1, _ =␣
↪precision_recall_fscore_support(y_test_fold, y_test_pred, average='weighted')

            accuracy_scores.append(accuracy)
            precision_scores.append(precision)
            recall_scores.append(recall)
            f1_scores.append(f1)

        # Calculate and print the mean and standard deviation of evaluation␣
↪metrics across folds
        print("Mean Accuracy:", sum(accuracy_scores) / len(accuracy_scores))
        print("Mean Precision:", sum(precision_scores) / len(precision_scores))
        print("Mean Recall:", sum(recall_scores) / len(recall_scores))
        print("Mean F1 Score:", sum(f1_scores) / len(f1_scores))
```

```
Classifier: MultinomialNB, Vectorizer: CountVectorizer
Mean Accuracy: 0.8225810628820056
Mean Precision: 0.8334277095589927
Mean Recall: 0.8225810628820056
Mean F1 Score: 0.8253928667187898
Classifier: MultinomialNB, Vectorizer: TfidfVectorizer
Mean Accuracy: 0.779345799233399
Mean Precision: 0.8174185407754925
Mean Recall: 0.779345799233399
Mean F1 Score: 0.7361895579381386
Classifier: LogisticRegression, Vectorizer: CountVectorizer
Mean Accuracy: 0.8454729099761732
Mean Precision: 0.8432257760906646
Mean Recall: 0.8454729099761732
Mean F1 Score: 0.8400062306226802
Classifier: LogisticRegression, Vectorizer: TfidfVectorizer
Mean Accuracy: 0.810892986636279
Mean Precision: 0.8187844969569668
Mean Recall: 0.810892986636279
Mean F1 Score: 0.7909439629833744
```