Financial Analysis Utilizing Machine Learning

Nithish Warren

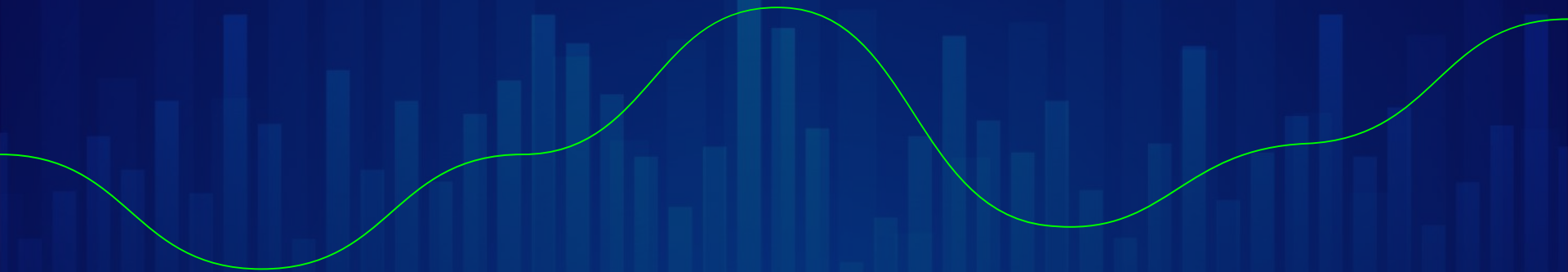


Table of contents

01 Introduction

02 Project Goals and Contents

03 Dataset 1

04 Dataset 2

05 Dataset 3

06 Conclusion

01

Introduction

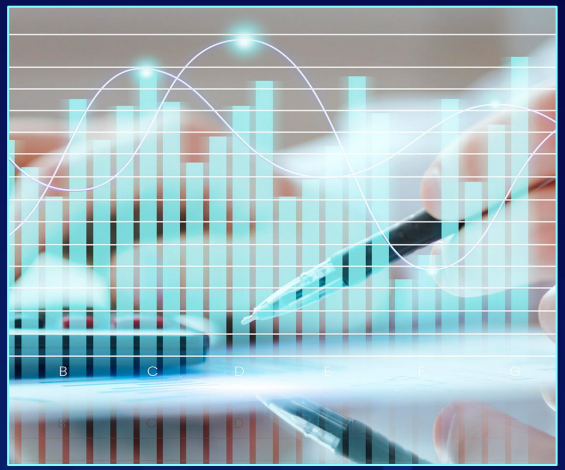
Who I am as well as the motivation behind this project.

Who am I?

- Graduate Student in Rutgers ECE
- Specializing in ML/AI

Motivation

- Creating a multimodal dataset of financial data.
- Building a model to use this dataset to beat the stock market.



02

Goals and Contents

What I hope to accomplish, as well as how to do it.

Objectives

Goals

- Understand more about financial features in the stock market, as well as applying machine learning techniques to them.
- Gain experience handling financial datasets, including preprocessing and extracting features from these datasets.

Approach

- Use clustering to find relationships in stocks that aren't inherently obvious.
- Use classification techniques to determine market sentiment from tweets.
- Use a random forest and hidden markov model to find market regimes.

03

Dataset 1

Using clustering to find patterns in stocks.

Dataset Overview/Data Exploration

- "200+ Financial Indicators of US stocks (2014-2018)" - Kaggle
  - https://www.kaggle.com/datasets/cnic92/200-financial-indicators-of-us-stocks-20142018

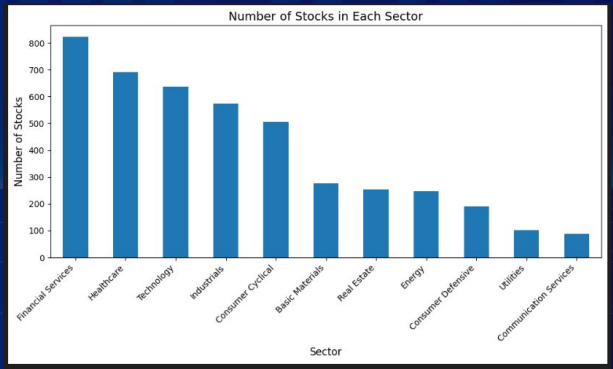
From our dataset exploration, we can see that there are 4392 stocks, and 225 features correlating to each stock. Examples of these features are Revenue, Revenue Growth, Gross Profit, and more. The majority of the features are numerical, with the two categorical ones being the ticker (label) of the stock, and the sector, which refers to what industry the stock is in.

Note that this data is from 2018 – while it would have been nice to get more recent information, this many features simply would not have been available elsewhere.

```
#sample and feature number
num_samples = data.shape[0]
num_features = data.shape[1]
print(f"Number of data samples: {num_samples}")
print(f"Number of features: {num_features}")
```

✓ 0.0s

```
Number of data samples: 4392
Number of features: 225
```



Data Preprocessing

Some preprocessing was needed in this dataset:

- Fix first column of dataframe (it is unnamed, so we call it 'stockTicker').
- We also want to use only stocks in the S&P 500 AND in the technology sector, as 4000+ stocks is not feasible to perform clustering on.
- Lastly, we replace any NaN values with the mean of that column, a technique called mean imputation.

```
Renamed Columns:
Index(['stockTicker', 'Revenue', 'Revenue Growth', 'Cost of Revenue',
       'Gross Profit'],
      dtype='object')
```

```
#remove NaN values (two columns contain all NaN values), replace others with mean
data = data.drop(columns=['operatingCycle', 'cashConversionCycle'])

#fill NaN values with average value from column
data = data.apply(lambda col: col.fillna(col.mean()) if col.dtype in ['float64', 'int64'] else col)
total_nan = data.isna().sum().sum()
print(f"Total NaN values in the DataFrame(after imputation): {total_nan}")
```

```
Total NaN values in the DataFrame(before imputation): 3113
Total NaN values in the DataFrame(after imputation): 0
```

Feature Extraction

- For our feature extraction, we need to reduce the number of features present, as 220+ features isn't feasible to perform clustering on.
- We find the 25 most relevant features to ‘Gross Profit’.
- Standardize the features, and then apply PCA to reduce the feature size to two dimensions for easier plotting.

```
#calculate 25 features that are highly correlated with 'gross profit'
correlation_matrix = technology_numerical_df.corr()
gross_profit_correlation = correlation_matrix['Gross Profit']
top_correlated_features = gross_profit_correlation.abs().sort_values(ascending=False).iloc[1:26]
top_features = top_correlated_features.index.tolist()
print(top_features)
```

Examples of most relevant features:

```
['Revenue', 'Total assets', 'Operating Cash Flow', 'Depreciation & Amortization',

'R&D Expenses', 'EBITDA', 'Operating Expenses', 'Other Liabilities'
```

Machine Learning Techniques

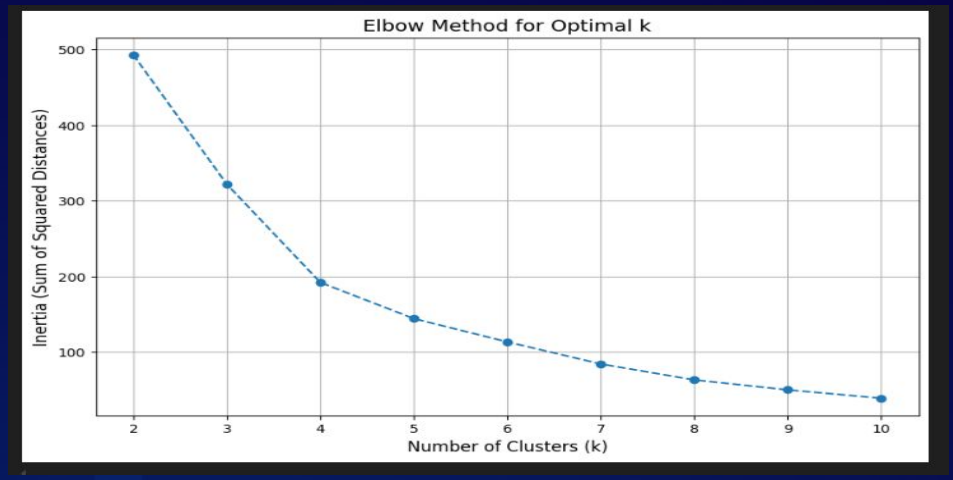
K-means Clustering

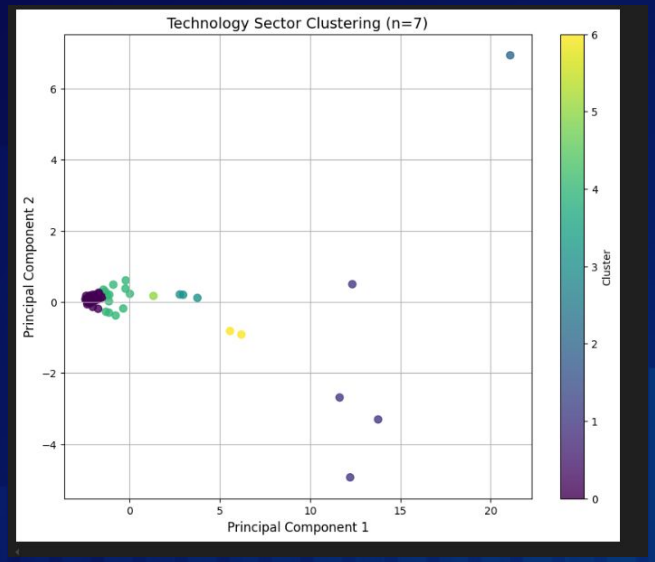
- We chose this because it's widely used and an efficient clustering algorithm. Very simplistic and interpretable.
- It works by randomly initializing k cluster centroids, assigning each data point to the nearest centroid based on some distance metric, and updating and repeating until the cetriods dont change significantly or reach some iterations.
- Implemented using KMeans class from sklearn. Firstly, we determined the optimal amount of clusters (k) by calculating inertia and silhouette scores for several k-values. With the optimal k-value of 7, we applied K-means clustering on the PCA-transformed data, and displayed the clustering results.

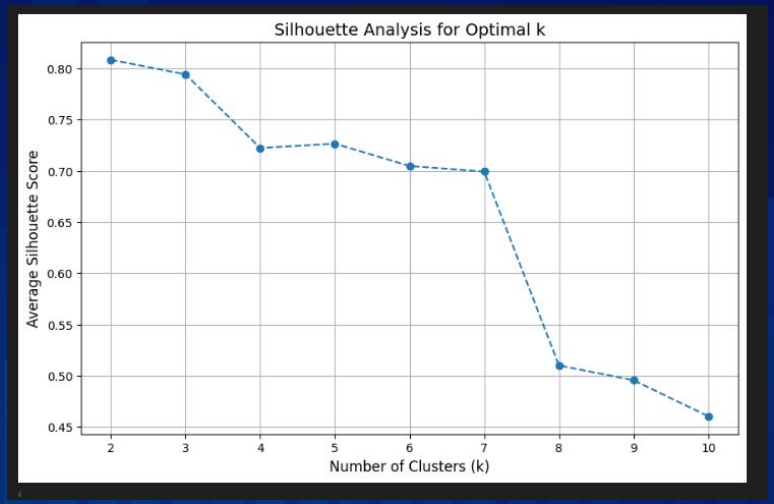
Hierarchical Clustering

- We chose this because it provides a dendrogram, which helps us visualize the clustering hierarchy and cluster structures.
- It works by calculating the pairwise distance matrix for all data points, and merging the closest two clusters based on a linkage criterion. We build the hierarchical tree and form the dendrogram.
- Implemented using AgglomerativeClustering from sklearn, grouping the PCA data into 7 clusters, as well as using Ward's linkage to minimize variance in the cluster. Displayed the results in a dendrogram, as well as grouping them with text.

Graphical Analysis (KMeans)

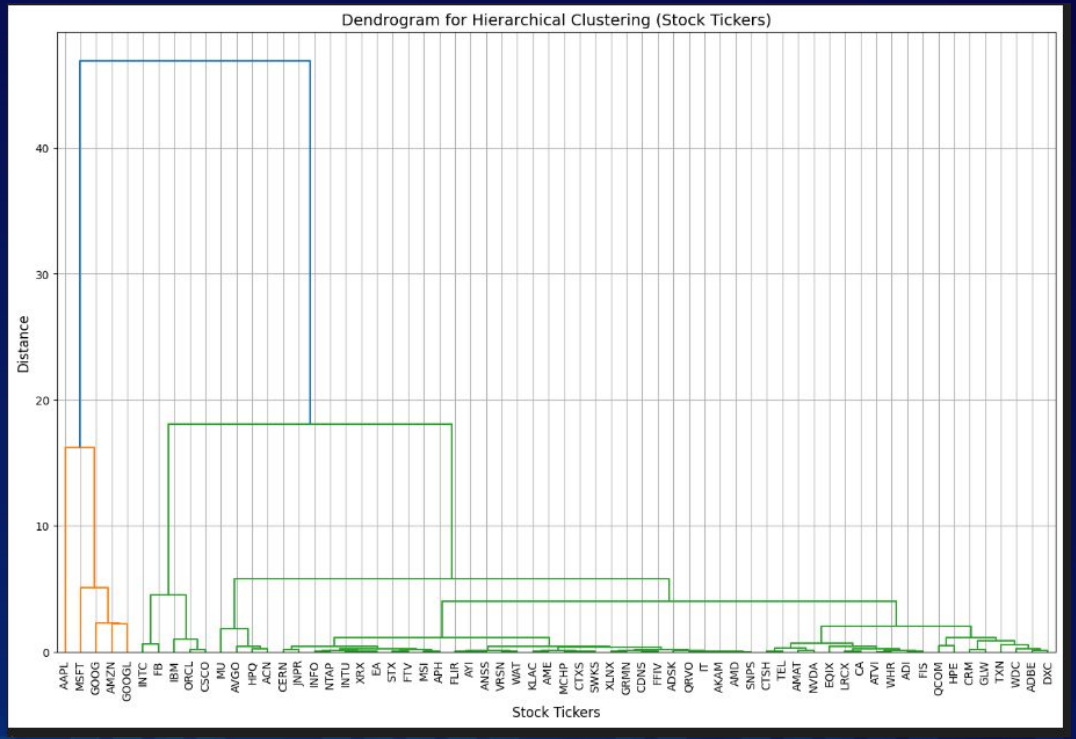






Graphical Analysis (Hierarchical)

From this dendrogram, some of the clear clustering can be seen around the larger technology companies, such as Apple, Microsoft, and Amazon. There are also some smaller sub-clusters like with Oracle, IBM, and CSCO.



Comparison of Methods

- In our comparison of results, we see that there is a slightly higher silhouette score for hierarchical clustering, making it the better method. However, it isn't significant enough of a difference to say that it's miles better than the other.
- However, there is a pretty big issue in both clustering results – the imbalance of values in each cluster. There is one cluster that the models have grouped the majority of stocks in, with the others seemingly being outliers.
- This mainly seems to be due to the fact that a lot of these outliers are the biggest tech companies currently (Apple, Meta, Microsoft, etc.). Potential improvement could be found by removing these stocks and finding clusters without them.
- While we weren't able to find the best patterns between stocks, we can still use these results in a meaningful way, as demonstrated after the comparison.

```
# Silhouette score for k-means
kmeans_silhouette = silhouette_score(technology_pca_data, technology_data['Cluster'])
print(f"Silhouette Score for k-means: {kmeans_silhouette:.2f}")

# Silhouette score for hierarchical clustering
hierarchical_silhouette = silhouette_score(technology_pca_data, technology_data['Hierarchical_Cluster'])
print(f"Silhouette Score for Hierarchical Clustering: {hierarchical_silhouette:.2f}")

# Cluster sizes for k-means
kmeans_cluster_sizes = technology_data['Cluster'].value_counts()

# Cluster sizes for hierarchical clustering
hierarchical_cluster_sizes = technology_data['Hierarchical_Cluster'].value_counts()
```

```
Silhouette Score for k-means: 0.60
Silhouette Score for Hierarchical Clustering: 0.65
```

Practical Application

Using the hierarchical clustering algorithm output, we are able to generate a diversified portfolio of investments and percentages. By calculating the percentage of a stock in its respective cluster, we can come up with a reasonable amount to invest in a certain stock.

This is an effective way to use our findings, despite the results not being exactly what we were expecting.

```
Portfolio Allocation:
MU: 4.1667%
HPQ: 4.1667%
AVGO: 4.1667%
ACN: 4.1667%
AMZN: 5.5556%
GOOGL: 5.5556%
GOOG: 5.5556%
AAPL: 16.6667%
ORCL: 5.5556%
CSCO: 5.5556%
IBM: 5.5556%
MSFT: 16.6667%
INTC: 8.3333%
FB: 8.3333%
```

04

Dataset 2

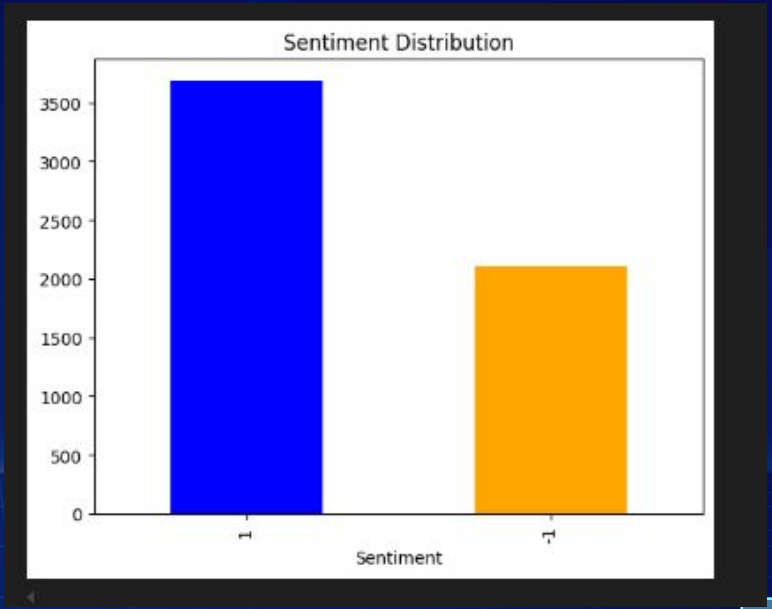
Determining sentiment of tweets using classification.

Dataset Overview/Data Exploration

- Stock-Market Sentiment Data
  - https://www.kaggle.com/datasets/yash612/stockmarket-sentiment-dataset

From our dataset exploration, we can see that there are 5971 tweets in the dataset, with 3685 positively correlated ones and 2106 being negative.

Initially wanted to use an unlabeled dataset -> https://www.kaggle.com/datasets/equinxx/stock-tweets-for-sentiment-analysis-and-prediction - however, there was no feasible way to label the dataset other than manually(Advantage of this dataset - dates were included, so you could visualize how often a positive tweet correlated to a bull market and vice versa).



Data Preprocessing

Some preprocessing was needed in this dataset:

- Since we are using text data, we need to prepare it for tokenization. This involves cleaning the data to remove certain characters and make all the letters lowercase.

- Not only that, we want to remove the possibility of any ethical concerns with this dataset(since the practical application would be data-scraping tweets in real-time). So, we remove any mentions from the tweets as well.

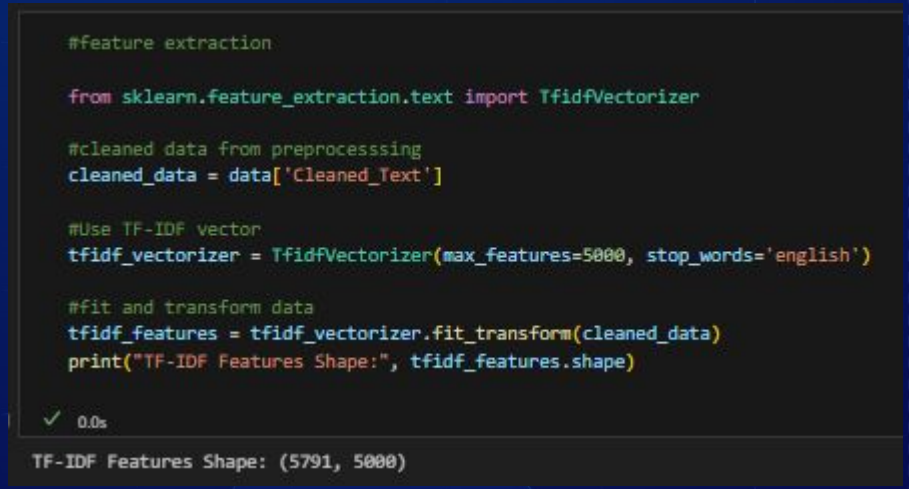
```
#data preprocessing

def basic_cleaning(text):
    text = text.lower()  #convert to lowercase
    text = re.sub(r'@\w+', '', text)  #remove mentions
    text = re.sub(r'[^a-zA-Z\s]', '', text)  #remove special characters
    text = re.sub(r'\s+', ' ', text).strip()  #remove spaces
    return text

data['Cleaned_Text'] = data['Text'].apply(basic_cleaning)
```

Feature Extraction

- Our feature extraction involves converting the text data into numerical features so that ML models can process them. We use the TfidfVectorizer function from sklearn, which works by assigning each word a term frequency (how often it shows up in a tweet), and then adjusting this value based on how often it shows up across all tweets. We maximize the features to 5000 and remove any common english stopwords. From this, we geta matrix where each row represents a tweet, and each column represents the importance of a term.



Machine Learning Techniques

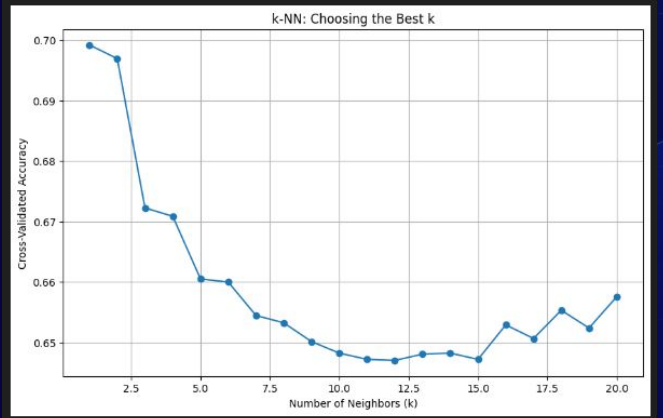
Naive Bayes

- We chose this because it works well with text classification, since it assumes feature independence, aligning with the nature of TF-IDF.
- It works by looking at the probabilities of each word appearing in positive and negative tweets form the training data. These probabilities are then combined to decide the overall sentiment.
- Implemented using MultinomialNB from sklearn. We also use k-folds cross validation in order to ensure robustness, setting k=5 as a standard. Since our accuracy was very similar each time, we know our model performs well.

```
Mean Accuracy: 0.7508177349004039
```

K-Nearest-Neighbors

- We chose this because it makes decisions based on its similarity to nearby datapoints, making it a useful binary classifier.
- It works by assigning a test sample to the class with the greatest number of k-neighbors. The distance used to measure between the datapoint and its neighbors is (in this case) Euclidean.
- Implemented using KNeighborsClassifier from sklearn, first trying the classifier at k=5. Then, we find the best k value by plotting the classifier at k values from 1-20. From this, we can see k=3 is a good balance between accuracy and overfitting, so we train the final KNN accuracy using that.

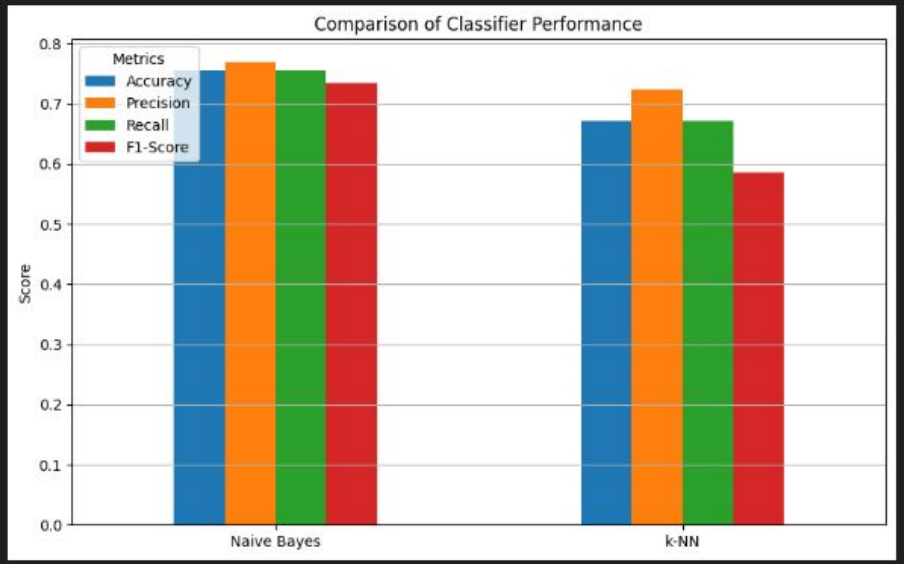


Comparison of Methods

- In our comparison of results, we calculate common evaluation metrics for both the KNN and Naive Bayes classifiers, with those metrics being accuracy, precision, recall, and F1 score.
- We can see that Naive Bayes performs better in every single category – this is most likely due to the fact that it excels at identifying the likelihood of words for each class, making it well-suited for text classification tasks. On the other hand, k-NN struggles in high-dimensional spaces due to the curse of dimensionality, where distances between points lose meaning.
- We can once again find a practical application using our trained naive bayes classifier (since it performed better).

Comparison Table:

	Accuracy	Precision	Recall	F1-Score
Naive Bayes	0.755824	0.768642	0.755824	0.733341
k-NN	0.671268	0.722836	0.671268	0.585426



Practical Application

Lastly, as a sort of "practical" test (since the ideal scenario would be to data-scrape tweets in real-time and predict sentiment), we retrain the Naive Bayes classifier on all the data, and give it a positive and negative tweet that I have made up to see if classifies them correctly.

This shows our model works, and when coupled with real-time data scraping, shows exactly how it would be used.

```
#train naive bayes on full dataset
tfidf_features = tfidf_vectorizer.transform(data['Cleaned_Text'])
nb_classifier = MultinomialNB()
nb_classifier.fit(tfidf_features, data['Sentiment'])

#example tweets
tweets = [
    "Professor Bajwa says the Rutgers stock will skyrocket!",
    "Professor Bajwa says that we should short the Rutgers stock."
]

#classify tweets function based on naive bayes
def classify_tweet(tweet, classifier, vectorizer):
    cleaned_tweet = basic_cleaning(tweet)  # Clean the tweet
    transformed_tweet = vectorizer.transform([cleaned_tweet])  # Transform using TF-IDF
    predicted_sentiment = classifier.predict(transformed_tweet)  # Predict sentiment
    return "Positive" if predicted_sentiment[0] == 1 else "Negative"

#perform classification and print
for tweet in tweets:
    sentiment = classify_tweet(tweet, nb_classifier, tfidf_vectorizer)
    print(f"The tweet: \"{tweet}\" is classified as {sentiment}.")
```

✓ 0.0s

```
The tweet: "Professor Bajwa says the Rutgers stock will skyrocket!" is classified as Positive.
The tweet: "Professor Bajwa says that we should short the Rutgers stock." is classified as Negative.
```

05

Dataset 3

Find states in the market where conditions for trading are unfavorable.

Understanding the Goal

A market regime refers to a specific state or phase of the financial market, characterized by distinct behaviors such as low or high volatility and differing levels of risk and return. For example, a low-volatility regime typically indicates stable markets with smaller price fluctuations, while a high-volatility regime reflects unstable markets with larger swings, often associated with higher risk.

The goal of this notebook is *not* to perform regression by predicting specific stock prices or trends but to classify each time period into one of these regimes.

By identifying market regimes, we can adapt trading strategies to changing market conditions, such as avoiding high-risk periods or maximizing returns during stable phases. Since market regimes are not explicitly labeled in the data, models like the Hidden Markov Model (HMM) are employed to infer hidden states, while classifiers like Random Forest provide a comparative perspective. This classification helps manage risk, optimize returns, and improve overall trading strategies.

Dataset Overview/Data Exploration

- Huge Stock Market Dataset
  - https://www.kaggle.com/datasets/borismarjanovic/price-volume-data-for-all-us-stocks-etfs?resource=download

From our dataset exploration, we can see that there are 8016 rows and 7 columns, with the start corresponding to first available record of the SPY ETF.

We DO NOT use the dataset above – contained data mainly for individual stocks, rather than for the SPY ETF. As a result, we generate data using the yfinance package and the script to the right (given in GitHub)

```
Number of samples (rows): 8016
Number of raw features (columns): 7
```

```python
#generate dataset in correct format

def generate_spy_data(csv_filepath, start_date, end_date):
    spy_data = yf.download("SPY", start=start_date, end=end_date)
    spy_data.reset_index(inplace=True)
    spy_data = spy_data.rename(columns={
        "Date": "Date",
        "Open": "Open",
        "High": "High",
        "Low": "Low",
        "Close": "Close",
        "Adj Close": "Adj Close",
        "Volume": "Volume"
    })

    spy_data.to_csv(csv_filepath, index=False)
    print(f"S&P 500 data saved to {csv_filepath}")

generate_spy_data(
    csv_filepath="spy_data_1993_2024.csv",
    start_date="1993-01-01",
    end_date="2024-12-1"
)
```

✓ 0.1s

```
[**********100%**********]  1 of 1 completed
S&P 500 data saved to spy_data_1993_2024.csv
```

Data Preprocessing

Not much preprocessing was needed in this dataset, since we generated/formatted the values already. We simply check for any NaN values.

We do include some preprocessing when making the dataset, as we have to calculate returns, since that is what the HMM is trained on.

What we are hoping to accomplish from this dataset is to identify hidden states where there is heightened market volatility or unfavorable conditions (ex, 2008, 2020).


```
#preprocessing

#ensure no missing values
missing_fraction = data.isnull().mean()  # Fraction of mis
missing_fraction = missing_fraction[missing_fraction > 0]
print(missing_fraction) #should be empty

print(data.head())
```

```
#load data function
def load_data(csv_filepath):
    df = pd.read_csv(csv_filepath, index_col="Date", parse_dates=True)
    df["Returns"] = df["Adj Close"].pct_change()
    return df.dropna()
```

Feature Extraction

- For feature extraction, again, we are solely using stock data. Right now, we want to define the training and test data split for both the logistic regression and HMM model. The reason that we include the split in feature extraction is for several reasons. One, since this is a time-series dataset, performing cross-validation is tricky/not feasible, since we need to maintain chronological order. Two, by splitting the data now, we prevent any potential data leakage from occurring later when calculating values like moving average.

```
def feature_extraction_and_split(df):
    #ensure no missing values (again)
    df.dropna(inplace=True)

    #split train and test
    train_data = df.loc[:'2010-12-31']  #data up to 2010
    test_data = df.loc['2011-01-01':]  #data from 2011 onward

    return train_data, test_data

#apply this split
train_data, test_data = feature_extraction_and_split(data)

print("Training Data:")
print(train_data.head())

print("\nTesting Data:")
print(test_data.head())
```

Machine Learning Techniques

Hidden Markov Model

- We chose this because it excels in analyzing and modeling sequential data, and in our case, is time-series stock market data. The market goes between different regimes, like highly volatile bear markets and low volatility bull markets. These states aren't directly observable, which makes this model very effective.

- It works by assuming hidden states that define the process - the aforementioned market regimes. Each return is generated by one of the hidden states, and the model transitions between these states based on a state transition matrix, specifying the probabilities of moving from one state to another.

- Implemented using GaussianHMM class from the hmmlearn library. We initialized it to find two market regimes, full covariance matrices, and up to 1000 iterations for convergence. The trained HMM was applied to training and test datasets using .predict(), and this created a sequence of hidden states for each dataset, representing market regimes over time. Adding these states as a new column in the train and test datasets let us use the Random Forest model as a comparison.

Random Forest

- We chose the Random Forest algorithm because it allows us to directly compare its performance against the Hidden Markov Model (HMM). By training the Random Forest model using the HMM-predicted states as targets, we can evaluate its ability to predict the same market regimes. This gives us a practical way to assess whether a supervised machine learning approach, like Random Forest, can perform as well as the HMM in identifying hidden market states.

- It works by creating multiple decision trees during training, where each tree is trained on a random subset of the data. When making predictions, the model aggregates the outputs of all these trees.

- Implemented using RandomForestClassifier from sklearn. Used returns as features, and the HMM-predicted states as targets. We can use K-Fold cross validation here to split the training data, since it doesn't require chronological input. Once trained, we predicted market states for the test set.

Comparison of Methods

- First, the predictions from the Random Forest model were compare directly with the HMM states with standard evaluation metrics, like accuracy, precision, recall, and F1 score.
- To compare the models directly, we create a simple backtesting strategy that only trades during a low volatility period (regime 0), resulting in the trade signal being equal to 1. Using cumulative returns, we can calculate two evaluation metrics.
- These metrics are the Sharpe ratio – the risk-adjusted returns of the strategy (higher Sharpe ratio = better returns relative to risk), and the Maximum Drawdown (maximum loss from a peak in cumulative returns).
- Our results show that Random Forest was able to achieve an accuracy of 65% in predicting HMM states, which makes it a reasonable comparison to the HMM itself.
- The more interesting metrics, the Sharpe ratio and Maximum Drawdown, show that while Random Forest had a higher Sharpe ratio, it also had a lower Maximum Drawdown, making it susceptible to loss in a volatile market.

```
Random Forest vs. HMM States Performance:
Accuracy: 0.6496763754045307
Confusion Matrix:
 [[578 166]
 [267 225]]
Classification Report:
              precision    recall  f1-score   support

           0       0.68      0.78      0.73       744
           1       0.58      0.46      0.51       492

    accuracy                           0.65      1236
   macro avg       0.63      0.62      0.62      1236
weighted avg       0.64      0.65      0.64      1236

HMM Sharpe Ratio: 0.1511, Max Drawdown: -9.9743%
RF Sharpe Ratio: 0.1683, Max Drawdown: -12.1837%
```

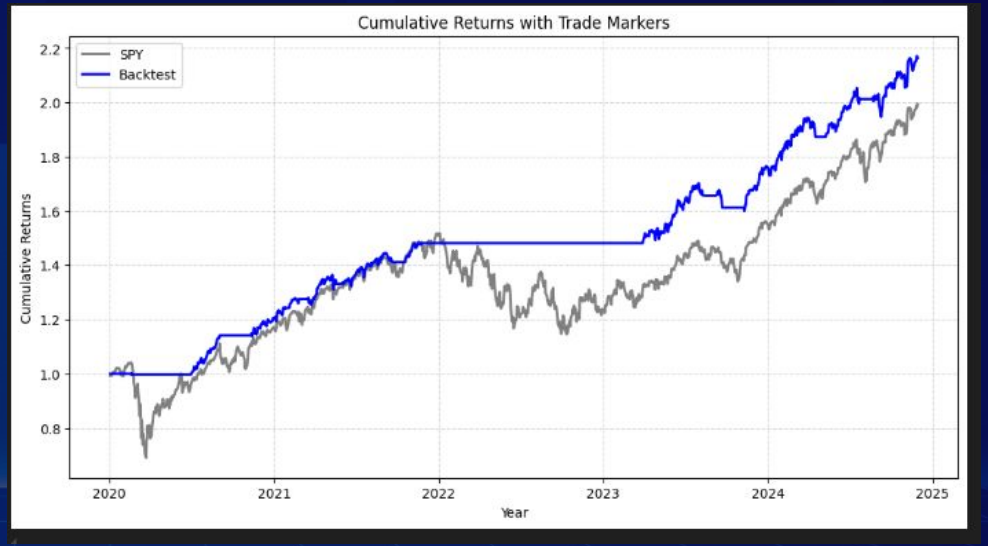
Practical Application

This implementation showcases how the Hidden Markov Model (HMM) can guide trading decisions in real-world financial markets. The model is trained on historical S&P 500 data, focusing on predicting market regimes, and is then integrated into a backtesting strategy to evaluate its effectiveness.

- We first train the HMM on data up to 2019, as to include as much as possible while still leaving enough to visually see the results.

- Our backtesting strategy uses moving average crossovers as a trading signal. A short-term (10-day) moving average and a long-term (30-day) moving average are calculated. A crossover (short-term moving above long-term) indicates a buy signal, while the reverse indicates a sell signal. We ONLY perform these trades when the HMM deems a regime as low volatility. Daily returns are then accumulated, which are plotted against the SPY (if you had just held throughout).

- As we can see, we actually perform better than the SPY – but this is just a backtest. It doesn't account for things like broker fees or the HMM not converging.



06

Dataset 3

What we have learned as well as future implementations.

Conclusion

Overall, despite some setbacks, the project was successful. We gained insight into working with machine learning models and datasets, as well as the machine learning pipeline, through cleaning data, feature extraction, hyperparameter tuning, and evaluation metrics.

A possible workflow that could come from this project – applying the sentiment analysis with the HMM to even further boost performance.

Further research into the HMM model should be done, as there is clearly potential based on the results.