Large Language Models (LLMs) and their financial application for US Communication Service Industry Sector

A extended research project report submitted to The University of Manchester
for the degree of **Master of Science in Data Science (Business & Management)**
in the Faculty of Humanities

Year of submission 2024

Student ID
11351804

School of Social Sciences

Table of Contents

WORD COUNT: 7,326

List of Abbreviations

BERT	*Bidirectional Encoder Representations from Transformers*
CRSP	*Center for Research in Security Prices*
DistilBERT	*Distilled BERT*
DistilRoBERTa	*Distilled RoBERTa*
EW	*Equal-Weighted*
FinBERT	*Financial BERT*
GPU	*Graphics Processing Unit*
IQ	*IQ Key Development*
L	*Long strategy*
LLMs	*Large Language Models*
LS	*Long-Short strategy*
RNN	*Recurrent Neural Network*
RoBERTa	*Robustly optimized BERT approach*
S	*Short strategy*
SR	*Sharpe Ratio*
US	*United States*
VW	*Value-Weighted*
WRDS	*Wharton Research Data Services*

List of Tables

List of Figures

Abstract

This study investigates the application of Large Language Models (LLMs) to predict stock returns to build portfolios in the US Communication Service Industry sector by combining sophisticated LLMs (such as BERT, RoBERTa, DistilBERT, DistilRoBERTa, and FinBERT) with standard econometric approaches. By using a comprehensive method that includes collecting and combining data, analysing sentiment, and doing rolling window analysis, the models were optimised to more accurately reflect market perceptions and patterns. The findings indicate that fine-tuning greatly enhances the capabilities of the models, with BERT and DistilBERT exhibiting greater accuracy and risk-adjusted returns. The efficiency of these models in balancing risk and return in long-short portfolios is highlighted by detailed performance assessments, which include accuracy measures and Sharpe Ratios. The results emphasise the significance of adapting LLMs for financial purposes, indicating significant implications for portfolio management, automated trading systems, and wider financial analysis. Although the research shows positive results, it also recognizes limits such as variations in performance across different models and the need for more extensive datasets. It is suggested that future study should focus on improving LLMs, investigating more sophisticated fine-tuning methods, and dealing with ethical and regulatory issues in algorithmic trading..

Acknowledgements

First and foremost, I thank Allah SWT for His blessings and guidance throughout this journey.

I am deeply grateful to my little family—my wife Lovita and my son Barra—for their unwavering support and love. To my father, who I hope is proud and watching over me from above, and to my mother, whose encouragement has been invaluable, thank you. I also extend my gratitude to my brothers, sisters, and extended family for their constant support.

A special gratitude to my supervisor, Dr. Eghbal Rahimikia, for his guidance and mentorship. Lastly, I want to acknowledge my friends in the UK, Gusti and Cokro, for their companionship and support over the past year.

Declaration

No portion of the work referred to in this extended research project report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning

Intellectual property statement

i The author of this extended research project report (including any appendices and/or schedules to this report) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii Copies of this report, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

iii The ownership of certain Copyright, patents, designs, trademarks, and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the report, for example graphs and tables ("Reproductions"), which may be described in this report, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv Further information on the conditions under which disclosure, publication and commercialisation of this report, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see https://documents.manchester.ac.uk/display.aspx?DocID=24420), in any relevant dissertation restriction declarations deposited in the University Library, The University Library's regulations (see https://www.library.manchester.ac.uk/about/regulations/) and in The University's Guidance for the Presentation of dissertations.

Chapter 1: Introduction

1.1 Background & Problem Statement

Machine learning and natural language processing have had a significant impact on the financial industry, leading to profound transformations. Large Language Models (LLMs) have shown remarkable proficiency in processing and analysing textual data. These models possess the ability to comprehend complex patterns in language, rendering them significantly valuable for forecasting market fluctuations by analysing news and sentiment.

The US Communication Service Industry, comprising major corporations such as Google (Alphabet), Netflix, Walt Disney, and AT&T, is extremely vulnerable to market sentiment and news events. Precise forecasting of stock returns in this industry can provide substantial advantages to investors and stakeholders. Nevertheless, the task of incorporating and exploiting high-frequency textual data for financial forecasts continues to be a challenge.

This study aims to tackle this challenge by examining the use of LLMs to perform sentiment analysis on headline news. The study further aims to use this analysis to predict stock returns and construct portfolios specifically for the US Communication Service Industry. To access a comprehensive list of the companies that used in this study and their respective market capitalizations, please refer to the Appendix A.

1.2 Shape and Scope of the Study

This study addresses several broad questions:

- What is the impact of extracting specific sentiment, such as positivity or negativity, from headline news on predictions of stock returns?
- What is the comparative efficacy of full-model versus distilled LLMs in processing and analysing financial news?
- Does incorporating sentiment data into stock return data improve the accuracy of predictive models designed for portfolio construction and risk assessment?

The scope includes Top-25 communication service companies in the United States, and the data covers a substantial period of time to ensure the analysis is robust and reliable. The focus is on developing a predictive model that integrates LLM-based

sentiment analysis with conventional econometric techniques in order to predict stock returns and build into a portfolio.

1.3 Structure of the Report

The report is structured as follows:

- **Chapter 1: Introduction** This chapter introduces the background and problem statement, study's shape and scope, and presents the overall structure of the report.
- **Chapter 2: Literature Review** This chapter provides an overview of the existing research conducted on LLMs and their specific applications in the field of financial forecasting. It includes a comprehensive overview of LLMs, encompassing their models, architecture, and their use case in financial forecasting.
- **Chapter 3: Methodology** This chapter provides an explanation of the study design and methodology employed in the study. It covers the conceptual framework, specifications of the hardware, data collection and preprocessing, analysis using rolling windows, sentiment analysis, and the techniques employed for forecasting stock returns and building the portfolio.
- **Chapter 4: Results and Discussion** This chapter showcases the practical findings of the study, assesses the effectiveness of the predictive models, and explores the significance of the results to the portfolio.
- **Chapter 5: Conclusion & Future Work** This chapter concludes and summarises the key findings, examines the consequences for both theory and practice, acknowledges the study's limitations, and provides recommendations for future research.

Chapter 2: Literature Review

2.1 Overview of Large Language Models

Large Language Models (LLMs) have transformed the field of natural language processing (NLP) by offering advanced capabilities in understanding and generating human language. The concept of LLMs was first introduced with the development of Eliza in the 1960s, which was the pioneering chatbot in the world. Eliza initiated the exploration of NLP, establishing the groundwork for subsequent, more intricate LLMs (Weizenbaum, 1966). Models like BERT, GPT, and their variations are trained on extensive datasets and use deep learning methods to extract intricate language patterns and contextual correlations in text. LLMs, or Language Models, are more successful than conventional word-based approaches because they take into account the syntactic and semantic intricacies of language. This makes them well-suited for tasks like sentiment analysis, machine translation, until text summarization. (Devlin et al., 2018; Radford et al., 2019).

LLMs are referred to as "large" because of their extensive parameter count, often ranging from hundreds of millions to billions. The wide range of parameters available to them enables them to acquire complex patterns and connections within the data, resulting in their exceptional performance in different NLP tasks (Brown et al., 2020).

The emergence of LLMs started with the introduction of models such as Word2Vec (Mikolov et al., 2013) and GloVe (Pennington et al., 2014). These models portrayed words as unchanging vectors in a space with several dimensions. The initial models served as the foundation for more advanced architectures such as BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer). These architectures employ transformer networks to process text in both directions and generate output, respectively (Devlin et al., 2018; Radford et al., 2019).

2.2 LLM Models and Architecture

LLMs often use transformer networks, which have encoders and decoders arranged in many layers. Vaswani et al. (2017) created Transformers, which represented a notable advancement in LLMs by surpassing earlier models such as Recurrent Neural Networks (RNNs) in their capacity to properly handle long-range

dependencies in text. Transformers, in contrast to RNNs, use self-attention mechanisms to concurrently analyse all tokens in a sequence, enabling improved parallelization and comprehension of context (Vaswani et al., 2017).

Table 1 presents a comparison of several LLM architecture and their respective characteristics used in this study:

Table 1. Models architecture and characteristics comparison

Model	Number of Parameters	Pre-training Data	Architecture and Characteristic
BERT	110M	BooksCorpus, Wikipedia	Bidirectional, Masked Language Modelling
RoBERTa	125M	BooksCorpus, Wikipedia	Enhanced training procedure, larger batch sizes
FinBERT	85M	Financial News Corpus, FinBERT	Domain-specific pre-training for financial texts
Distil BERT	66M	BooksCorpus, Wikipedia	Distilled version of BERT, smaller and faster
Distil RoBERTa	82M	BooksCorpus, Wikipedia	Distilled version of RoBERTa, focused on efficiency

Note: *This table compares different LLMs based on their number of parameters, pre-training data sources, and key architectural characteristics. These factors highlight the strengths and specialised features of each model, providing a foundation for understanding their application in sentiment analysis and financial forecasting.*

BERT, a model created by Devlin et al. (2018), employs a bidirectional strategy for masked language modelling, enabling it to comprehend the contextual meaning of a word in both forward and backward directions. The bidirectional nature is essential for understanding intricate language patterns and nuances. BERT undergoes pre-training using the BooksCorpus and Wikipedia datasets, which include a total of 110 million parameters. This extensive pre-training equips BERT with the ability to comprehend and incorporate a wide variety of linguistic elements and contextual information.

RoBERTa, a model developed by Liu et al. (2019), improves upon BERT by including bigger batch sizes and more training data into the training process. The objective of this technique is to enhance the resilience and precision of the model. RoBERTa, which has been pre-trained on BooksCorpus and Wikipedia, consists of 125 million parameters, making it one of the largest and most powerful models in terms of parameter size.

FinBERT is a specialised model designed specifically for analysing financial language. FinBERT, created by Yang et al. (2019), utilises a financial news corpus and FinBERT data during pre-training to enhance its ability to comprehend and analyse financial language and terms with more efficiency. FinBERT, which has 85 million parameters, is specifically designed for jobs involving financial sentiment analysis and forecasting.

DistilBERT, a condensed iteration of BERT, was developed by Sanh et al. (2019) with the objective of reducing size and enhancing speed, while still maintaining a significant portion of BERT's precision. DistilBERT, which has 66 million parameters, undergoes pre-training using the same datasets as BERT. Similarly, DistilRoBERTa, created by Sanh et al. (2019), is a condensed iteration of RoBERTa. The major objective is to enhance efficiency without compromising performance, using a total of 82 million parameters.

2.3 LLM Application in Stock Forecasting

The use of LLMs in stock forecasting is an expanding field of study. Conventional methods of financial prediction often depend on numerical data and technological indicators. Nevertheless, including textual data, such as news articles, earnings reports, and social media postings, might provide supplementary information that can improve prediction models (Bybee et al., 2020; Gu et al., 2020).

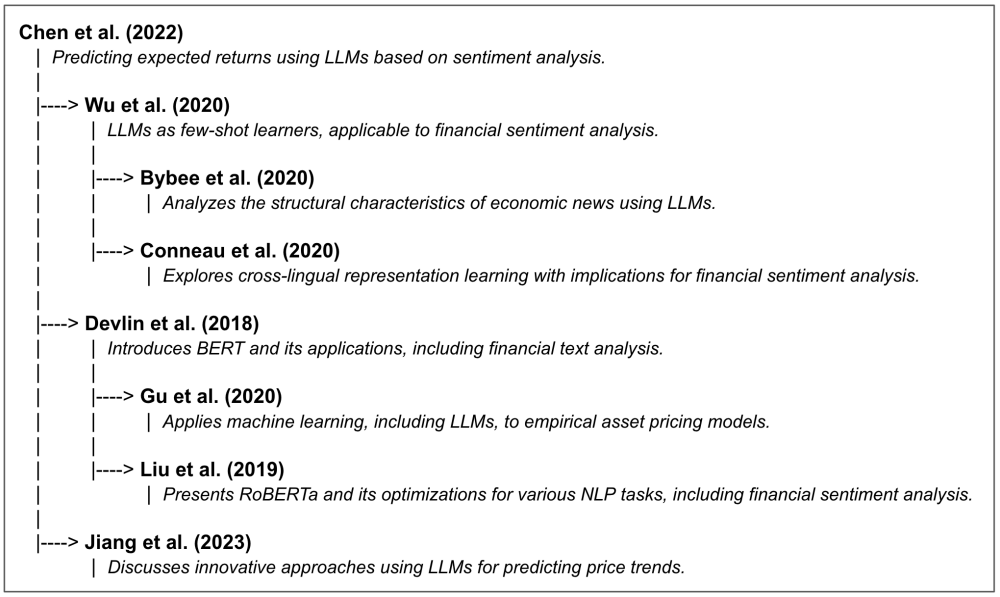


Figure 1. Flowchart of key studies and findings related to LLMs in Financial Forecasting.

Multiple studies have shown that LLMs are successful at extracting sentiment from text data and applying it to forecast stock returns. Figure 1 provides a concise overview of key studies and findings about the use of LLMs in financial prediction. Chen et al. (2022) demonstrated that using contextualised representations of news content via LLMs yields superior results compared to conventional word-based approaches for forecasting stock returns. Their research included data from several global stock markets and emphasised the capacity of LLMs to collect intricate information that impacts market sentiment and consequent stock prices.

LLMs provide a more advanced analysis of text, taking into account elements like negation, sarcasm, and complex syntactic patterns that less complicated models may overlook. Wu et al. (2020) investigated the use of LLMs as few-shot learners for financial sentiment research, while Bybee et al. (2020) examined the structural attributes of economic news using LLMs. Conneau et al. (2020) expanded on this by investigating cross-lingual representation learning, which has implications for financial sentiment analysis. As a result, there is an improvement in sentiment analysis accuracy and enhanced predictive capabilities in stock forecasting models (Devlin et al., 2018; Liu et al., 2019).

Moreover, the use of sentiment data into econometric models has shown enhanced resilience and precision in forecasting stock returns. Through the use of LLMs, researchers have the ability to construct models that not only enhance the accuracy of return predictions, but also provide a more profound understanding of the fundamental reasons that influence market fluctuations (Gu et al., 2020; Jiang et al., 2023).

Chapter 3: Methodology

3.1 Conceptual Framework

The study's conceptual framework combines LLMs with conventional econometric techniques, such as logistic regression, to forecast stock returns and conduct sentiment analysis specifically for the US Communication Service Industry. The purpose of this integration is to utilise the sophisticated capabilities of LLMs in processing and interpreting textual data, while also leveraging the strength and reliability of conventional econometric approaches to achieve accurate financial projections.

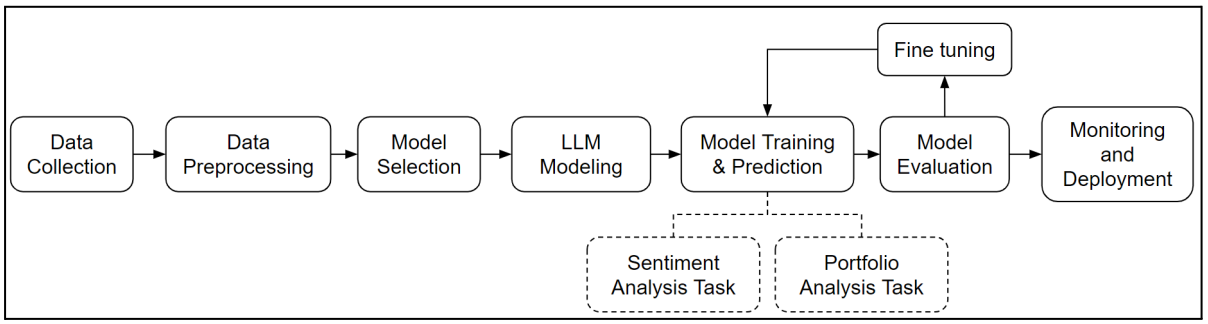


Figure 2. Methodology of LLM Application for Communication Industry Stock Forecasting

The diagram shown in Figure 2 provides the complete methodology of this study, starting by collecting data from Wharton Research Data Services (WRDS), including extensive datasets from Centre for Research in Security Prices (CRSP) for stock returns and IQ Key Development for headline news. After collecting the data, the next step is data preparation, which includes cleaning and transforming the data. This involves the process of selecting the top 25 companies by their market capitalization, eliminating duplicate headlines, managing any missing data, transforming daily returns into weekly returns, until merging the datasets to form a unified dataset for analysis.

The process of model selection has significance as it involves choosing suitable LLMs and econometric models for analysis. LLM modelling is the use of sophisticated LLMs to analyse the emotion of headline news. Pre-trained models include the extraction of average embeddings and the application of logistic regression to estimate sentiment. On the other hand, fine-tuned models do sentiment

analysis directly, without the need for extracting embeddings as an intermediary step.

The framework employs rolling window analysis to ensure the stability and correctness of the model over time. This systematic methodology guarantees a comprehensive analysis, using the advantages of both LLMs and conventional econometric approaches to provide precise financial forecasts and valuable insights.

3.2 Hardware Specifications

The study used computing resources offered by Google Colab Pro. This platform provides powerful hardware capabilities that facilitate the processing, analysis, and training of models required for this study. The Table 2 below provides a comprehensive breakdown of the specs for the GPU model utilised:

Table 2. Google Colab Pro L4 GPU Specification

Specification	Description
GPU Model	NVIDIA L4
GPU Memory	24 GB GDDR6
GPU Temperature	Typically between 60-80°C
GPU Power Usage (Max)	120W
GPU Memory Usage	Up to 24 GB
System RAM	52 GB
Disk Space	200 GB SSD

Note: *This table outlines the hardware specifications of the Google Colab Pro L4 GPU, including GPU model, memory, operating temperature, power usage, and system specifications. These details are crucial for understanding the computational resources used for training and fine-tuning the Large Language Models (LLMs) in this study.*

For this study, over 300 Collab units were used. The substantial use of computer resources facilitated the effective management of large datasets, training of complex models, and execution of the requisite calculations for sentiment analysis and return forecasts. The combination use of the high-performance NVIDIA L4 GPU and system RAM were crucial in training the deep learning models, resulting in a substantial reduction in processing time compared to using just the CPU.

3.3 Data Collection

The data for this research is obtained from the WRDS, which consists of the CRSP for stock returns and IQ Key Development for headline news. The dataset spans

from 2005 to 2023, with data from 2005 to 2015 used for training (in-sample) and data from 2016 to 2023 used for validation (out-of-sample).

CRSP Dataset: The CRSP data offers extensive financial information, as shown in the Table 3 below:

Table 3. CRSP Dataset Variables

Field	Description
Date	The specific date of each record
Ticker	The stock ticker symbol of the company
Company Name	The name of the company
PERMCO	A unique identifier for each company
Price	The closing price of the stock
Volume	The number of shares traded
Return	The stock return, calculated as the percentage change in price
Shrout	The number of shares outstanding

Note: *This table outlines the key variables included in the CRSP dataset, which are used for the analysis of stock market performance. Each variable provides essential information about the stock trading activities and financial metrics of the listed companies.*

The IQ Key Development dataset comprises comprehensive news and event data, as shown in the Table 4 below:

Table 4. IQ Key Development Dataset Variables

Field	Description
Company Name	The name of the company involved in the event
Event Type	The type of event (e.g., earnings announcement, merger, acquisition)
Announce Date	The date the event was announced
Headline	The headline of the news article
Situation	The full body of the headline, providing more context and details about the event
Keydevid	A unique identifier for each event

Note: *This table presents the variables included in the IQ Key Development dataset. These variables capture essential details about significant company events, which are crucial for understanding the context and impact of news articles on stock market performance.*

3.4 Data Preprocessing

Data preparation involves a series of crucial tasks aimed at preparing the data for analysis. The method starts with choosing the Top-25 companies from the CRSP dataset, based on their largest market capitalization, that were listed as of the year 2005. This selection guarantees that the study concentrates on established and important companies in the US Communication Service Industry sector.

After selecting the companies, the daily returns are converted into weekly returns and are accomplished by using adjusted pricing. It is essential to accurately account for stock splits and other company activities when calculating the adjusted price of the company and ensuring the returns in order to retain accuracy. This phase guarantees that the returns accurately represent the actual economic worth and fluctuations in stock prices.

Data normalisation is then used to standardise numerical data across disparate scales. Normalisation guarantees that the analysis is not influenced by the differing scales of various characteristics, enabling more significant comparisons and model training.

The subsequent step is to eliminate identical headlines that occur on the same day by retaining the latest '*situation*' variable that has the most complete information in order to prevent repetition in sentiment analysis. It is crucial to do this step in order to guarantee that each headline makes a distinct contribution to the sentiment score or probability, without adding any bias via repetition.

Dealing with absent headlines is another crucial component of preprocessing. Dates that do not have headlines are replaced with a '*[No_Headline]*' placeholder in order to ensure the consistency of the dataset. This method guarantees the systematic documentation of the lack of news, which may serve as a helpful indication in sentiment analysis.

Ultimately, the process of combining the CRSP and IQ Key Development datasets is performed in order to create a unified dataset that can be used for analysis. This ensures that financial data and headline news are appropriately synced by aligning each input based on date and its respective company.

3.5 Rolling Windows Analysis

The use of rolling window analysis ensures the preservation of model stability and correctness as time progresses. This methodology entails partitioning the dataset into several windows, each including a training phase and a validation period. More precisely, the study utilises 8 windows, with each period consisting of 10 years of training data and 1 year of validation data. Figure 3 below depicts the process of rolling windows analysis in this study, displaying the periods of training and validation across the whole series spanning from 2005 to 2023.

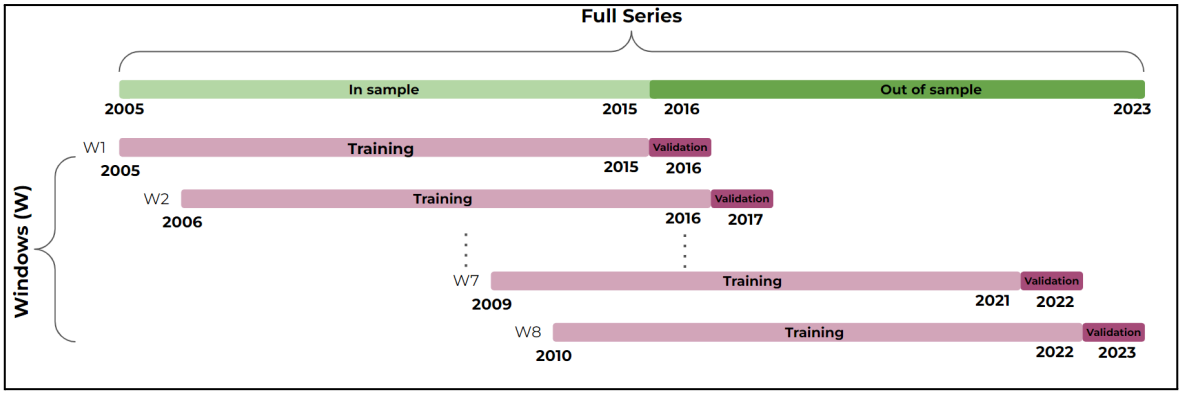


Figure 3. Rolling Windows Analysis

The approach starts by delineating each window to include a training time of 10 years, followed by a validation phase of 1 year. During the training phase, the models are trained using the data from the specified training period. Afterwards, the validation phase consists of evaluating the model's performance by utilising the data from the succeeding validation period. Following each validation period, the models are refreshed with the latest data, resulting in a one-year advancement of the time horizon.

The rolling window technique guarantees that the models are consistently updated and verified using current data, capturing any evolving trends and patterns. By using this approach, it enhances the resilience and adaptability of the models, enabling more precise forecasts in a constantly changing market setting.

3.6 Sentiment Analysis

3.6.1 Pre-trained

Large Language Models (LLMs) are used to do sentiment analysis on headline news, extracting the emotional tone associated with the text. At first, five pre-trained models are employed: BERT, RoBERTa, FinBERT, DistilBERT, and DistilRoBERTa.

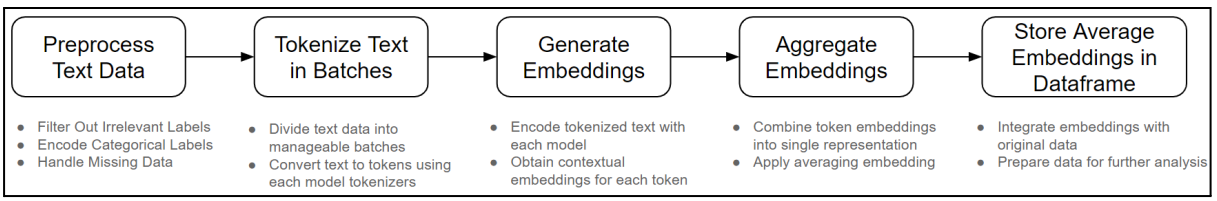


Figure 4. Average Embeddings Extraction Process

Pre-trained models start the process by obtaining average embeddings and then use logistic regression to determine sentiment probability and the price direction compared to the previous and the next week (up or down). This process involves

many stages as shown in Figure 4 above. Initially, the text data undergoes preprocessing to eliminate extraneous labels, encode category categories, and address missing data. Subsequently, the text is partitioned into batches, breaking down the text data into manageable chunks and transforming it into tokens using the tokenizers specific to each model. After the text is divided into tokens, the embeddings are created by encoding the tokenized text using each model. This process results in obtaining contextual embeddings for each individual token. The token embeddings are consolidated into a unified representation by using an averaging procedure. Ultimately, the average embeddings are saved in a dataframe, merging the embeddings with the original data and making it ready for further analysis.

3.6.2 Logistic Regression

Logistic Regression is used to create a model that predicts the likelihood of a binary event, such as whether stock price will go up or down, based on sentiment probabilities obtained from the LLMs. The equation below describes the connection between the labels and features:

$$E(y_i, t \mid x_i, t) = \sigma(x_i, t\beta)$$

In this context, σ(x) represents the logistic link function, which transforms the characteristics into a numerical value ranging from 0 to 1. This process ensures that the measurement of sentiment is standardised. Using this strategy, sentiment scores can be computed for each row in the testing sample, with the tone of a news article determined by its sentiment score. By incorporating Logistic Regression with pre-trained models, it improves the capacity to reliably categorise sentiment, which is vital for forecasting stock fluctuations.

3.6.3 Fine-tuned

Unlike the process of obtaining average embeddings and utilising logistic regression, the fine-tuned models immediately carry out sentiment analysis. The fine-tuned SequenceClassifier model predicts both the price direction and the likelihood of negative or positive emotion. This efficient method utilises the improved functionalities of finely-tuned models to provide precise sentiment forecasts.

The efficacy of the sentiment analysis is assessed by Accuracy, which guarantees the models' proficiency in capturing the sentiment expressed in the headlines. The

Table 5 outlines the distinctions between pre-trained models and fine-tuned models for the sentiment analysis task:

Table 5. Pre-trained (Untuned) vs Fine Tuned Models for Sentiment Analysis Task

Model	Pre-trained Models	Fine Tuned Models for Sentiment Analysis Task
BERT	'BertModel'	'BertForSequenceClassification'
RoBERTa	'RoBERTaModel'	'RobertaForSequenceClassification'
Distil BERT	'DistilBertModel'	'DistilBertForSequenceClassification'
Distil RoBERTa	'DistilRobertaModel'	'DistilRobertaForSequenceClassification'
FinBERT	'finbert-pretrain'	Finbert-tone 'BertForSequenceClassification'

Note: *This table contrasts pre-trained and fine-tuned models used for sentiment analysis in this study. Pre-trained models are initially trained on general datasets, while fine-tuned models are further trained on domain-specific (in this study is sequence classification) data to enhance their performance in predicting sentiment from financial texts.*

3.7 Portfolio Construction & Return Prediction

The final phase is forecasting stock returns by incorporating sentiment data into econometric models. The sentiment analysis results from the LLMs are combined with cumulative log returns to improve the forecasting precision of the models. More precisely, the models are trained to predict stock returns by using cumulative log returns, which provide a more reliable measure of returns over a period of time in comparison to plain arithmetic returns. Cumulative Logarithmic (log) return is calculated using the following formula:

$$Cumulative\ Log\ Return = \sum_{t=0}^{n} log(1 + Weekly\ Return_t)$$

This measure incorporates the compounding impact of returns over many time periods, resulting in a more precise depiction of investment success over time.

A total of 12 methods are used for portfolio development, including both untuned and fine-tuned models from five distinct LLMs (BERT, RoBERTa, DistilBERT, DistilRoBERTa, FinBERT). The strategies are outlined below:

- Equal Weighted Long: Creating a portfolio by allocating equal amounts of investment to equities that are expected to provide positive returns.
- Equal Weighted Short: Creating a portfolio by evenly investing in equities that are expected to have negative returns.

- The Equal Weighted Long-Short strategy involves creating a portfolio with no net investment by going long on the top 20% of companies with the most positive sentiment ratings and shorting the bottom 20% of stocks with the most negative sentiment scores.
- Value Weighted Long: Creating a portfolio by investing in companies that are expected to have good returns, with the investment amount determined by their market capitalization.
- Value Weighted Short: Creating a portfolio by investing in companies that are expected to have negative returns, with the investment amount determined by their market capitalization.
- The Value Weighted Long-Short strategy involves creating a portfolio with no net investment by going long on the top 20% of stocks with the highest positive sentiment scores and shorting the bottom 20% of stocks with the most negative sentiment scores. The weights of these positions are determined by the market capitalization of the stocks.

The portfolios' performance is assessed using Sharpe Ratios, which quantify the risk-adjusted return of the portfolio. The Sharpe Ratio (annualised) is computed using the below formula:

$$\textit{Sharpe Ratio} = (R_p / \sigma_p) \times \sqrt{52}$$

where R_p is the average return of the portfolio and σ_p represents the standard deviation of the portfolio's return. This methodology guarantees that the portfolios are built with the intention of achieving the optimal performance. It utilises the predictive capabilities of LLMs and econometric methodologies to provide precise and dependable projections of stock returns.

Chapter 4: Results and Discussion

4.1 Statistical Overview

This section presents a comprehensive examination of the results of weekly returns, headline news, top tokens, the average number of tokens per headline across various models, and the visual representations of word clouds.

4.1.1 Weekly Return

The summary data for the weekly returns of companies in the US Communication Service Industry sector are extensive, including a range of parameters like mean, standard deviation, and quartiles, among others. This information is essential for understanding the performance and volatility of any company's stock. As an example, AT&T Inc. (Ticker: T) has an average weekly return of 0.001357 and a standard deviation of 0.029655. This suggests that AT&T's returns are relatively stable with moderate variability. In contrast, companies like Nexstar Media Group (NXST) have higher standard deviations (0.096741), indicating greater volatility in their stock returns. The weekly returns for AT&T range from a low of -0.189807 to a top of 0.140761. The quartiles for AT&T are as follows: Q1 is -0.014286, Q2 is 0.000891, and Q3 is 0.016607. This indicates that 25% of AT&T's weekly returns fall below -0.014286, while 75% fall below 0.016607. These insights are helpful for investors or portfolio managers who want to comprehend the risk and return characteristics of various equities. For a complete summary for each company, please refer to Appendix B.

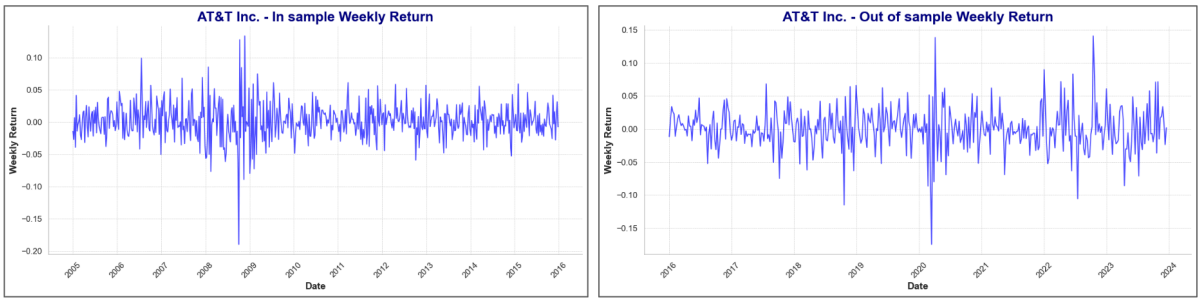


Figure 5. Weekly Return of In-sample (left) and Out-of-sample (right) for AT&T Inc.

The visualisations shown above demonstrate the weekly return for AT&T Inc. throughout the periods of analysis and validation. The chart on the left displays the returns seen during the period from 2005 to 2016, which is referred to as the in-sample period. Conversely, the chart on the right illustrates the returns observed

during the period from 2016 to 2024, known as the out-of-sample period. These charts provide a clear depiction of the fluctuation and patterns in AT&T's stock returns over a period of time. To access the comprehensive in-sample and out-of-sample statistics for all companies, please see Appendix C.

4.1.2 Headline News

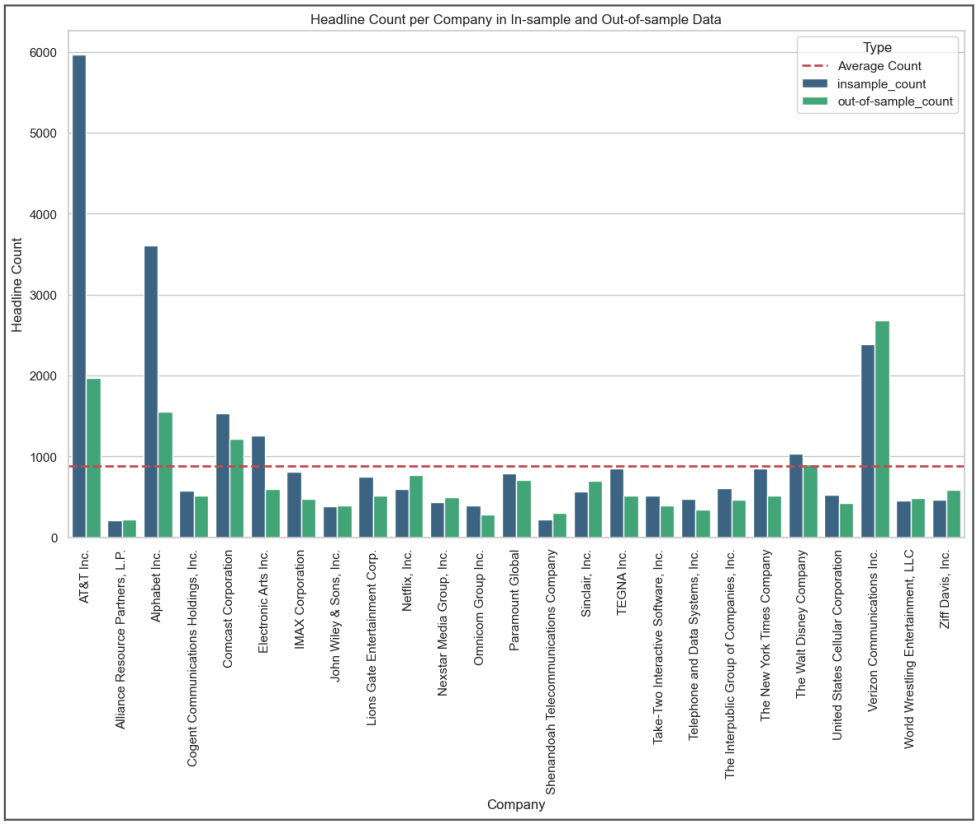


Figure 6. Headline count per-Company

The bar chart shown in Figure 6 above displays the distribution of headline counts per company. The headline count per company offers a comprehensive perspective of the news coverage that each company gets, which is a crucial aspect in sentiment research. AT&T Inc. and Alphabet (Google) are companies that have a noticeably larger number of headlines with up to 6000 and 3500 headlines respectively, suggesting that they get news attention more often. The heightened media focus might result in higher stock price volatility when news spreads quickly and market responses occur.

The variation in the number of headlines also underscores the varying degrees of media visibility across firms. Companies with lower headline counts may encounter less market volatility as a result of a smaller number of news events that significantly

impact the market. The consequences of media coverage are crucial for sentiment analysis since regular news announcements might result in more precise sentiment predictions owing to a larger dataset.

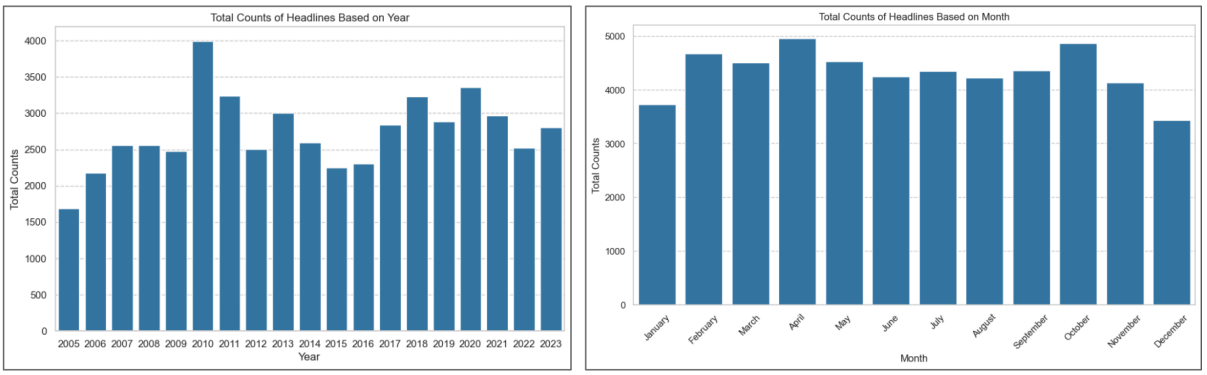


Figure 7. News Headline counts based on Year (left) and Month (right)
US Stock Communication Service Industry

In addition, examining news headlines by year and month offers further insights regarding the temporal distribution of news coverage. The left figure in Figure 7 displays the cumulative number of headlines each year from 2005 to 2023, emphasising the yearly fluctuations in media coverage. It is worth mentioning that in 2010, there was a notable increase in the number of headlines related to key events or advancements in the Communication Service Industry. The figure on the right displays the overall number of headlines by month, showing a fairly uniform distribution with spikes in April and October, reflecting seasonal patterns in news coverage.

4.1.3 Word Cloud & Top Tokens

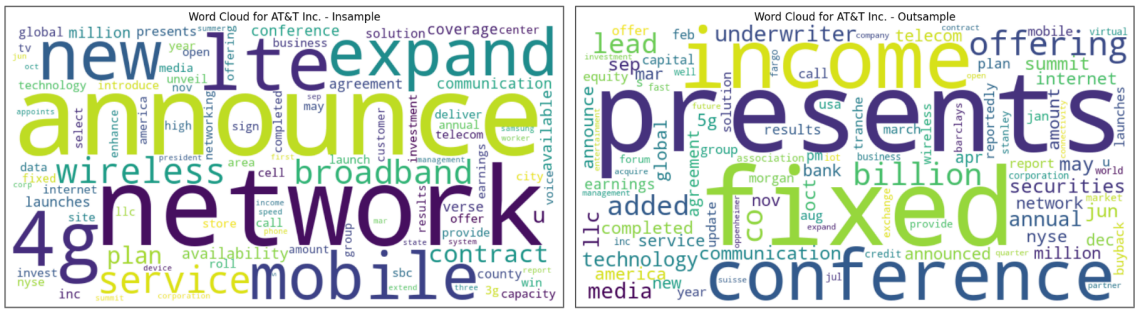


Figure 8. Word Cloud of In-sample (left) and Out-of-sample (right) for AT&T Inc.

The word clouds depict the terms that appear most often in the headlines. The example used is AT&T Inc., which is divided into two datasets: in-sample and out-of-sample (*For a complete list of word clouds for all companies, please see Appendix D*). The regular occurrence of corporate announcements and earnings

reports is reflected in the prevalence of tokens such as "conference," "announces," and "earnings" in financial news.

These tokens are crucial for training LLMs since they embody the fundamental topics found in the financial news corpus. The frequency of these tokens may also serve as an indicator of market sentiment trends, where certain terms may be linked to either positive or negative market movements. For example, the frequent use of the terms "earnings" and "results" may indicate a correlation with quarterly financial reports, which are crucial events that impact stock performance.

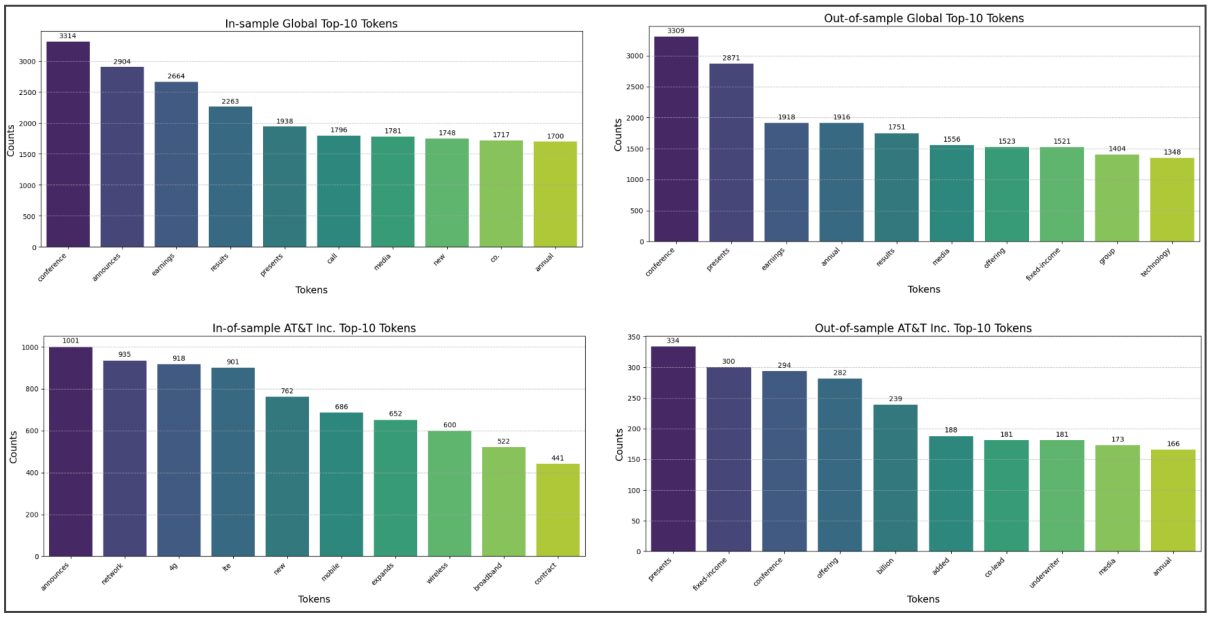


Figure 9. Top-10 Tokens in In-Sample (Top left), Out-of-Sample (Top right), and for a specific company, AT&T Inc In-sample (Bottom left) and Out-of-sample (Bottom Right)

Figure 9 displays the top-10 tokens used in headlines, both globally and particularly for AT&T Inc. as an example. The tokens are ordered based on their frequency of occurrence. During the in-sample period, the most often occurring tokens are "conference," "announces," "earnings," and "results," which suggest the prevalent subjects discussed in financial news within this specific time frame. Similarly, throughout the out-of-sample period, there is a persistent occurrence of words like "conference," "presents," "earnings," and "annual." These words indicate the continuing topics in financial reporting and market debates.

The out-of-sample top-10 tokens for AT&T Inc. include phrases such as "presents," "fixed-income," and "conference." These tokens represent distinct subjects that are pertinent to AT&T's corporate operations and financial occurrences. The frequency of these tokens in the headlines indicates a concentration on presentations, debates

connected to revenue, and conferences, which are crucial in the company's press coverage.

4.1.4 Average Tokens per Headline Across Models

The Table 6 presents a comparison of the average amount of tokens per headline across several LLMs and their respective tuning stages. The average token count per headline for BERT is 20 in its untuned condition and 19 after fine-tuning, suggesting a modest decrease in token count after fine-tuning. RoBERTa exhibits an average of 21 tokens per headline in its untuned state, which decreases to 20 tokens after fine-tuning. The average number of tokens per headline in both the untuned and fine-tuned phases of FinBERT is 18, indicating a stable token count across different tuning conditions.

Table 6. Average tokens per headline across the models

Models	BERT		RoBERTa		FinBERT	
Tuning	UT	FT	UT	FT	UT	FT
Average Tokens per Headline	20	19	21	20	18	18

Note: *This table shows the average number of tokens per headline for each model, comparing un-tuned (UT) and fine-tuned (FT) states. Fine-tuning generally reduces the average number of tokens per headline, reflecting the models' enhanced ability to capture essential information more concisely.*

The fluctuations in average token counts are noteworthy since they have a substantial influence on the model's efficiency and processing time. Models with lower token density per headline, such as FinBERT, exhibit enhanced processing speed, which is advantageous for real-time sentiment analysis. On the other hand, models that have a greater number of tokens may capture more specific information, but this comes with the drawback of requiring more processing resources and time.

The comparison reveals that fine-tuning often leads to a little decrease in the amount of tokens per headline. This may improve the model's performance by prioritising more important aspects of the text. The improvement in efficiency achieved by fine-tuning is a crucial element of model optimisation, guaranteeing that the models are both accurate and computationally efficient for practical use in financial sentiment analysis and stock prediction.

4.2 Results & Discussion

4.2.1 Model Performance

This section assesses the performance of the models using two main metrics: Accuracy for sentiment analysis in forecasting the direction of return, and Sharpe Ratios for constructing portfolios. The assessment measures used in this study include Accuracy for sentiment analysis, which gauges the models' capacity to forecast the direction of price (up or down), and Sharpe Ratios for portfolio construction.

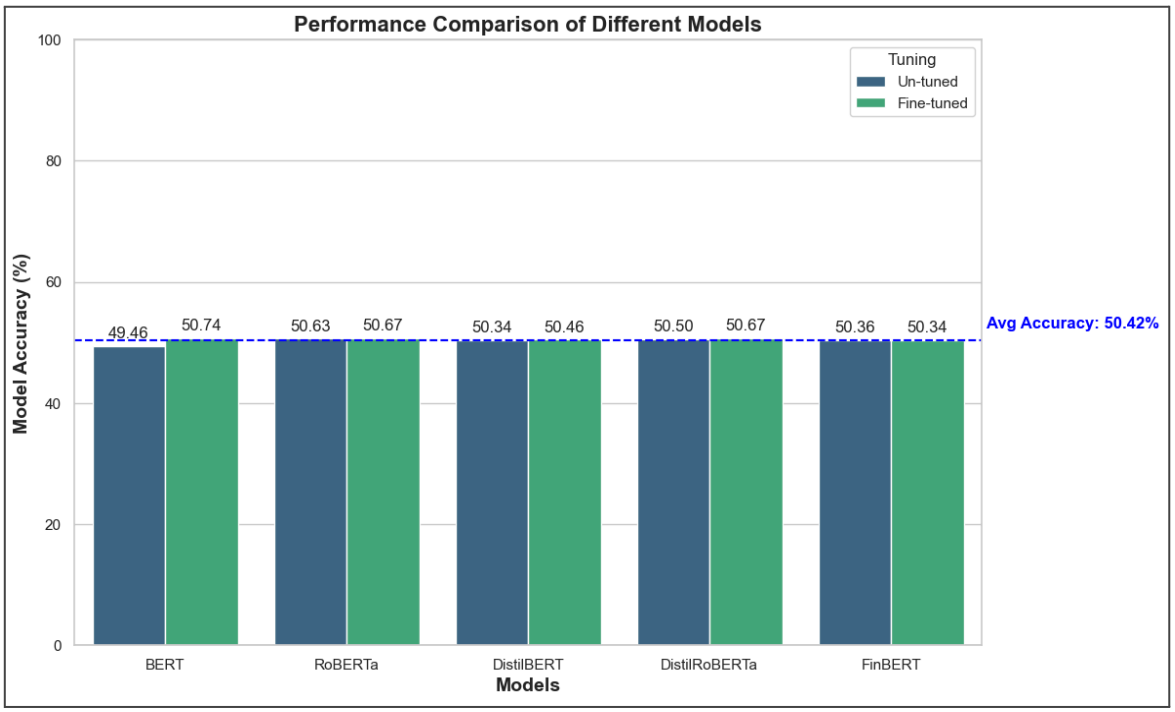


Figure 10. Model Accuracy by average Untuned vs Fine Tuned

Figure 10 presents a comparison of the predictive accuracy of several models (BERT, RoBERTa, DistilBERT, DistilRoBERTa, FinBERT) in determining the direction of price. Every model is assessed in both its un-tuned and fine-tuned stages. The metric demonstrates that fine-tuning typically enhances the accuracy of the models. The BERT model, after being fine-tuned, obtains the maximum accuracy of 50.74%. This indicates that it is the most successful model among the ones studied for predicting return direction. This enhancement emphasises the need of adjusting pre-existing models to suit certain financial scenarios via meticulous calibration. The mean accuracy across all models is 50.42%, suggesting a moderate but steady increase via fine-tuning. To see the complete model accuracy throughout the year, please see Appendix E.

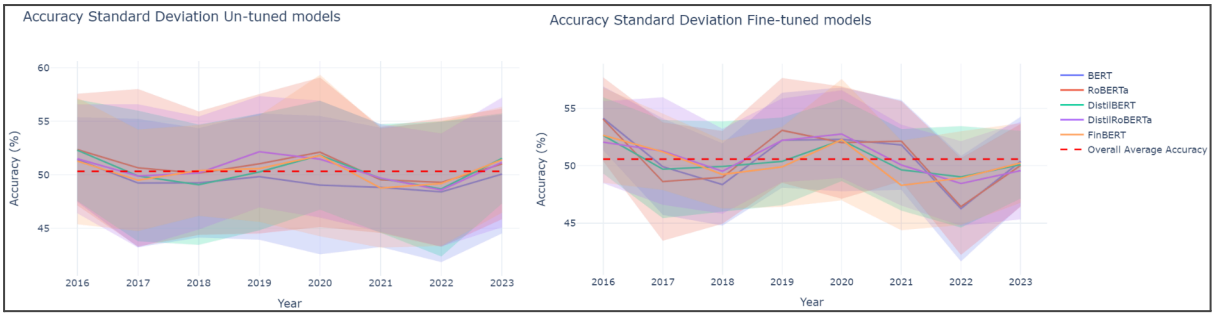


Figure 11. Models' Accuracy standard deviation

Furthermore, the standard deviation of accuracy for both un-tuned and fine-tuned models offers valuable information on the stability of the models' performance over time. The figure depicted in Figure 11 presents the standard deviation of accuracy for each model from 2016 to 2023. By fine-tuning, the standard deviation is decreased, which suggests a higher level of stability and consistency in performance. For instance, the BERT model that has been adjusted to specific requirements demonstrates reduced standard deviation values in comparison to the unadjusted version, indicating its resilience in various market conditions. For the whole table of accuracy standard deviations, please see Appendix F.

Table 7. Sharpe Ratios of all models across different strategies

Strategy		BERT		RoBERTa		Distil BERT		Distil RoBERTa		FinBERT	
		UT	FT	UT	FT	UT	FT	UT	FT	UT	FT
EW	LS	1.11	0.95	-1.52	-1.26	1.47	1.53	-1.24	-1.28	0.73	-0.87
	L	0.20	1.39	-0.08	1.20	1.11	1.06	-0.69	-0.10	1.03	-0.28
	S	-0.67	-0.44	1.97	1.98	-0.14	-1.41	0.49	1.48	0.07	0.97
VW	LS	0.80	1.39	-2.27	-0.55	1.26	1.16	-1.35	-1.45	0.65	-1.15
	L	-0.17	2.06	0.49	0.96	0.69	0.90	-0.59	0.42	0.91	-0.85
	S	-0.74	-0.33	1.57	1.60	-0.09	-0.92	0.72	1.87	-0.23	1.23

Note: *UT = Un-tuned, FT = Fine-tuned, EW = Equal-weighted, VW = Value-weighted, LS = Long-Short, L = Long, S = Short. These terms define the different strategies for each portfolio: Long-Short (LS) involves taking long and short positions simultaneously, Long (L) refers to taking long positions only, and Short (S) refers to taking short positions only.*

The Table 7 displays the Sharpe Ratios for several portfolio strategies, including Long-Short (LS), Long (L), and Short (S), utilising both equal-weighted (EW) and value-weighted (VW) portfolios. The examined models consist of BERT, RoBERTa,

DistilBERT, DistilRoBERTa, and FinBERT, in both their un-tuned and fine-tuned stages. The Sharpe Ratios demonstrate that certain models and strategies provide superior returns when adjusted for risk. The BERT model, especially when fine-tuned, consistently exhibits robust performance across different portfolio strategies. This indicates that BERT is highly suitable for financial applications, efficiently managing the trade-off between return and risk.

BERT's performance, especially in long-short strategies, demonstrates its resilience and dependability in forecasting returns and handling portfolio risk. Although DistilBERT has outstanding performance, BERT is still the favoured model because of its extensive design and superior ability to capture intricate financial patterns.

The findings emphasise the variation in performance across various models and tuning states, with BERT showing significant enhancements in risk-adjusted returns via fine-tuning. This highlights the need of carefully choosing and refining models to optimise financial forecasting models and create lucrative portfolios.

4.2.2 Return Predictions and Portfolio Performance

This section will only analyse the effectiveness of models within long-short strategies, taking into account the better sharpe ratios metric and its significance in minimising risk and attaining higher returns compared to other strategies. Long-short strategies are highly effective in mitigating market volatility and optimising portfolio performance, making them an essential component for study in financial applications.

The findings indicate that fine-tuned models often achieve better performance than their untuned equivalents, as seen by the summary statistics of expected weekly returns in comparison to actual returns. Specialised models like BERT provide superior accuracy and enhanced portfolio performance. Significantly, the projected returns from these models closely align with the actual returns, resulting in more efficient investment strategies. Ensuring that the expected returns correlate closely with the actual returns is essential for building dependable portfolios that may successfully achieve desired financial results.

The performance of the models in portfolio construction is illustrated through the cumulative log return of equal-weighted long-short (EW-LS) and value-weighted long-short (VW-LS) portfolios, shown for both untuned and fine-tuned models.

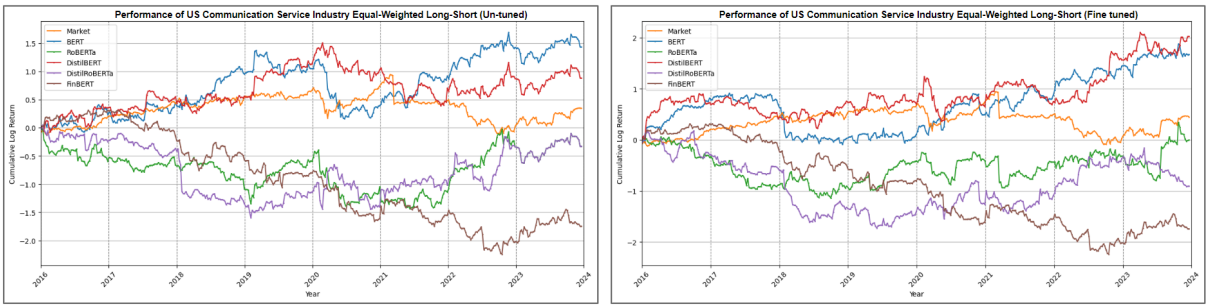


Figure 12. Equal-weighted Long-Short un-tuned (left) and Fine tuned (right) portfolios

First, for the EW-LS strategy on Figure 12, the chart on the left depicts the performance of models that have not been tuned, while the chart on the right illustrates the performance of models that have undergone fine-tuning. The charts show the cumulative log return performance of each model throughout the selected time from 2016 to 2023. The market benchmark is provided for comparison. Generally, the fine-tuned models exhibit enhanced performance compared to the untuned models, as shown by larger cumulative log returns. The BERT (fine-tuned) and DistilBERT (fine-tuned) models consistently demonstrate exceptional performance, surpassing other models and the market standard.

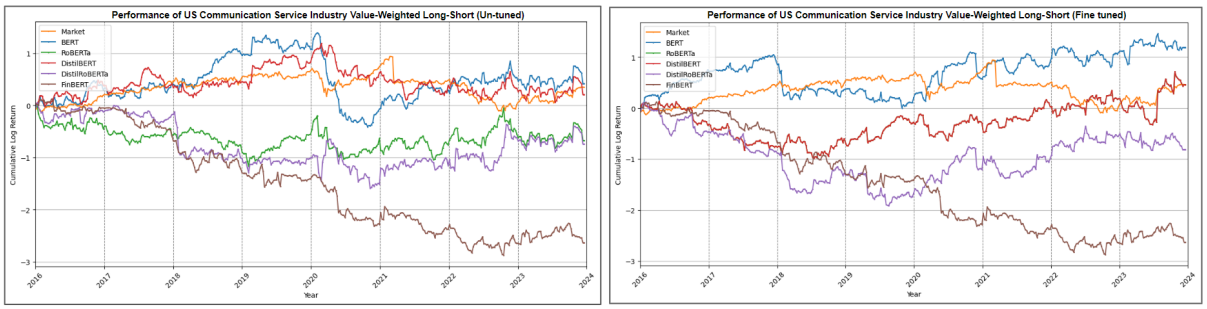


Figure 13. Value-weighted Long-Short un-tuned (left) and Fine tuned (right) portfolios

Second, for the VW-LS strategy on Figure 13, the chart on the left displays the performance of models that have not been adjusted, while the chart on the right displays the performance of models that have been adjusted for the value-weighted portfolios. These figures provide valuable insights into the performance of each model when investments are weighted based on market capitalization. The fine-tuned models once again exhibit enhanced performance, with BERT (fine-tuned) seeing significant improvements in comparison to its untuned equivalent. This emphasises the efficacy of fine-tuning in customising LLMs for certain financial situations and enhancing portfolio results.

4.2.3 Implications for Portfolio Management

The results indicate that making little adjustments greatly improves the ability of LLMs to forecast and perform well in financial applications. BERT and DistilBERT, when optimised, provide the highest level of accuracy in forecasts and demonstrate strong performance in portfolio management, making them well-suited for practical investment strategies in the US Communication Service Industry sector. Through the process of fine-tuning, models are able to enhance their ability to comprehend the subtleties of financial language and market mood, resulting in more precise forecasts. The persistent superiority of finely adjusted models compared to unadjusted models highlights the significance of customisation in financial implementations of LLMs. This customisation allows models to adjust to the individual attributes of financial data, including seasonal patterns, market fluctuations, and sector-specific news occurrences.

The efficacy of the fine-tuned models in the long-short strategies (both equal-weighted and value-weighted) demonstrates their proficiency in risk management. The fine-tuned BERT model's capability to get elevated Sharpe Ratios showcases its aptitude to provide superior risk-adjusted returns. The findings provide support for effective risk management tactics, which include maintaining a balance between long and short holdings and making accurate forecasts.

In volatile markets, it is vital for fine-tuned models to possess flexibility in order to effectively capture market emotion and trends. The cumulative log return findings indicate that the fine-tuned models have superior ability to adapt to changes in the market, constantly surpassing the performance of the market benchmark. This flexibility allows for the modification of strategy in response to market fluctuations.

Lastly, the Fine-tuned models may be used to enhance automated decision-making processes. The reliable and stable performance of the optimised BERT model may be included into automated trading systems to carry out transactions according to model forecasts, therefore simplifying the investing process and minimising human mistakes.

4.2.4 Analysis of Underperforming Models

Unlike BERT and DistilBERT that demonstrated significant improvements by fine-tuning, RoBERTa, DistilRoBERTa, and FinBERT models did not exhibit comparable performance. Multiple variables contribute to the poor performance of

these models. Although RoBERTa and FinBERT exhibit robust performance in a range of natural language processing tasks, they may not adequately capture the intricate intricacies of the financial market compared to BERT and DistilBERT. The designs of RoBERTa and FinBERT may not be well-suited for capturing the unique patterns and feelings often seen in financial news and data pertaining to the Communication Service Industry. The effectiveness of language models relies heavily on the calibre and specificity of the training data.

DistilRoBERTa and FinBERT may have encountered overfitting during the fine-tuning phase, when the models exhibited good performance on the training data but struggled to generalise to unfamiliar data. The observed outcomes may be attributed to the decreased Sharpe Ratios and cumulative log returns. The accuracy of sentiment analysis methods in assessing sentiment from financial texts might vary.

Chapter 5: Conclusion & Future Work

5.1 Conclusion

This study aims to investigate the efficacy of LLMs in forecasting stock returns and generating portfolios in the US Communication Service Industry sector. The extraction of specific sentiment from headline news has a substantial impact on the accuracy of stock return forecasts. Advanced models such as BERT and DistilBERT, which have been fine-tuned, are able to capture subtle subtleties in sentiment more correctly, hence enhancing portfolio performance. The comparative efficacy demonstrates that fine-tuned full-model LLMs, such as BERT, typically achieve better results than their distilled equivalents, but DistilBERT also exhibits significant enhancements. However, models such as RoBERTa, DistilRoBERTa, and FinBERT exhibited inferior performance, underscoring the significance of carefully choosing and optimising the model. Integrating sentiment data with stock return data improves the precision of predictive models for constructing portfolios and assessing risks, as demonstrated by greater Sharpe Ratios and better alignment between expected and actual returns. The findings emphasise the significance of customising models for financial applications, demonstrating the capability of LLMs to enhance portfolio management tactics.

5.2 Implications

This approach has major implications for the management of portfolios and the analysis of financial data. Refined models boost decision-making by offering more accurate forecasts of stock movements, resulting in optimum trading tactics and increased profitability. The exceptional success of the models in long-short portfolios demonstrates their ability to effectively manage risk and generate returns. Specifically, the meticulously optimised BERT model outperforms others by generating greater Sharpe Ratios and delivering superior risk-adjusted returns. Moreover, the models' capacity to identify high-performing stocks in the Communication Service Industry improves the allocation of resources and investment results. Integrating these models into automated trading systems simplifies decision-making, minimises human mistakes, and enhances operational efficiency. The flexibility of well-adjusted LLMs in response to unpredictable markets

provides a distinct advantage, and their use may expand beyond portfolio management to include other domains of financial research, such as credit risk evaluation and economic prediction.

5.3 Limitations

Although the findings show promise, it is important to realise numerous limitations. Although fine-tuned models have shown satisfactory performance overall, specific models like RoBERTa and FinBERT have not attained comparable levels of performance. This underscores the need for more study to enhance and optimise these models, particularly for financial applications. The effectiveness and precision of the training data have a substantial influence on the performance of the model. Overfitting is a persistent worry throughout the fine-tuning process, since it is essential to guarantee that models can properly generalise to new, unknown data.

5.4 Recommendations

Further studies should prioritise the exploration of sophisticated fine-tuning methods, such as using transfer learning with more extensive financial datasets, and integrating a broader array of financial data, including worldwide market data and macroeconomic variables. Utilising ensemble learning methods to merge numerous LLMs may improve the dependability and precision of predictions. Extending the study to more sectors may confirm the applicability of the results and provide insights that are special to each area. Creating real-time trading systems with highly calibrated LLMs may provide useful insights and enhance operational efficiency. By following these recommendations, one may fully unlock the strength of finely calibrated LLMs in financial applications, hence enhancing investment strategies and financial analysis.

References

Bybee, L., Kelly, B.T., Manela, A. and Xiu, D., 2020. The structure of economic news. Technical report, National Bureau of Economic Research.

Chen, Y., Kelly, B. and Xiu, D., 2022. Predicting expected returns using LLMs based on sentiment analysis.

Chen, Y., Kelly, B.T. and Xiu, D., 2022. Expected Returns and Large Language Models.

Conneau, A., Khandelwal, K., Goyal, N., Chaudhary, V., Wenzek, G., Guzmán, F., Grave, E., Ott, M., Zettlemoyer, L. and Stoyanov, V., 2020. Unsupervised cross-lingual representation learning at scale. In D. Jurafsky, J. Chai, N. Schluter, and J.R. Tetreault (Eds.), Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020, 8440–8451. Association for Computational Linguistics.

Devlin, J., Chang, M.W., Lee, K. and Toutanova, K., 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Gu, S., Kelly, B. and Xiu, D., 2020. Empirical asset pricing via machine learning. The Review of Financial Studies, 33, pp.2223–2273.

Jiang, J., Kelly, B. and Xiu, D., 2023. (Re-)Imag(In)ing Price Trends. Journal of Finance, forthcoming.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. and Stoyanov, V., 2019. RoBERTa: A robustly optimized BERT pretraining approach. arXiv preprint arXiv:1907.11692.

Sanh, V., Debut, L., Chaumond, J. and Wolf, T., 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. In Advances in Neural Information Processing Systems (pp.5998-6008).

Weizenbaum, J., 1966. ELIZA—a computer program for the study of natural language communication between man and machine. Communications of the ACM, 9(1), pp.36-45.

Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. and Amodei, D., 2020. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R.

Hadsell, M.F. Balcan, and H. Lin (Eds.), Advances in Neural Information Processing Systems, volume 33, pp.1877–1901. Curran Associates, Inc.

Wu, Z., Liu, Y., Li, X. and Sun, M., 2020. Few-shot learning for financial sentiment analysis with pre-trained language models.

Yang, Y., Lage, I., Chen, W., Yu, M. and others, 2019. FinBERT: A Pretrained Language Model for Financial Communications.

Appendices

Appendix A. Top 25 US Communication Service Stocks by Market Capitalization

No	Ticker	Company Name	Market Capital ($B)
1	GOOGL	ALPHABET INC / GOOGLE	958B
2	DIS	DISNEY WALT CO	367B
3	NFLX	NETFLIX INC	306B
4	T	AT&T INC	289B
5	CMCSA	COMCAST CORP NEW	283B
6	VZ	VERIZON COMMUNICATIONS INC	257B
7	PARA	VIACOM INC	61.1B
8	EA	ELECTRONIC ARTS INC	45.4B
9	TTWO	TAKE TWO INTERACTIVE SOFTWARE INC	27.9B
10	GCI	GANNETT CO INC	21B
11	OMC	OMNICOM GROUP INC	20.9B
12	IPG	INTERPUBLIC GROUP COS INC	15.8B
13	NYT	NEW YORK TIMES CO	9.35B
14	NXST	NEXSTAR BROADCASTING GROUP INC	7.84B
15	WWE	WORLD WRESTLING ENTMT INC	7.12B
16	ZD	J2 GLOBAL COMM INC	6.88B
17	LGF	LIONS GATE ENTERTAINMENT CORP	6.09B
18	USM	UNITED STATES CELLULAR CORP	5.71B
19	TDS	TELEPHONE & DATA SYSTEMS INC	4.48B
20	CCOI	COGENT COMMUNICATIONS GROUP INC	4.28B
21	SBGI	SINCLAIR BROADCAST GROUP INC	4.06B
22	ARLP	ALLIANCE RESOURCE PARTNERS	3.79
23	JW	WILEY JOHN & SONS INC	3.42B
24	SHEN	SHENANDOAH TELECOM CO	3.05B

25	IMAX	IMAX CORP	3.01B

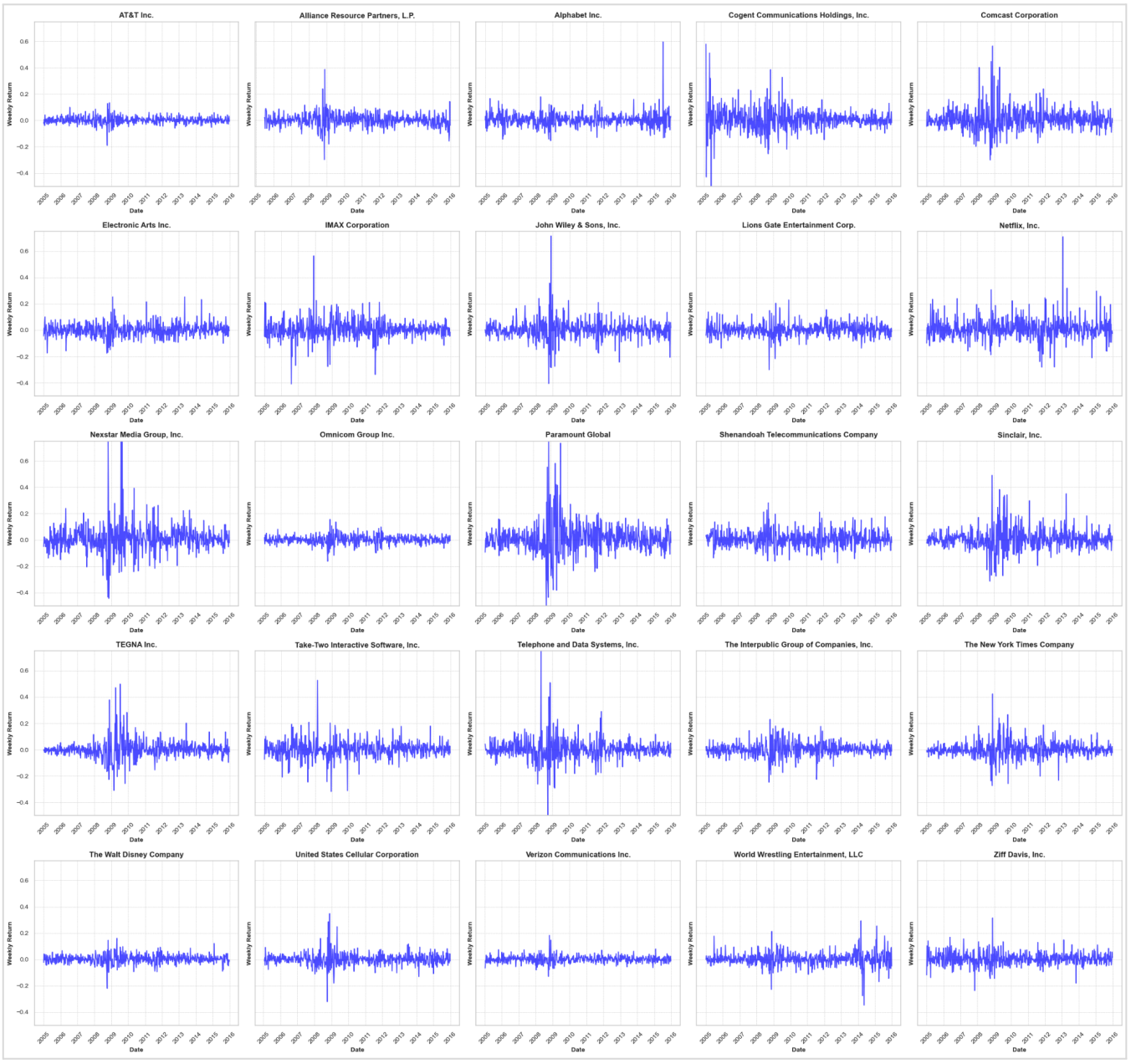
Note: *This table lists the top 25 companies in the US Communication Service sector, ranked by their market capitalization in billions of dollars.*

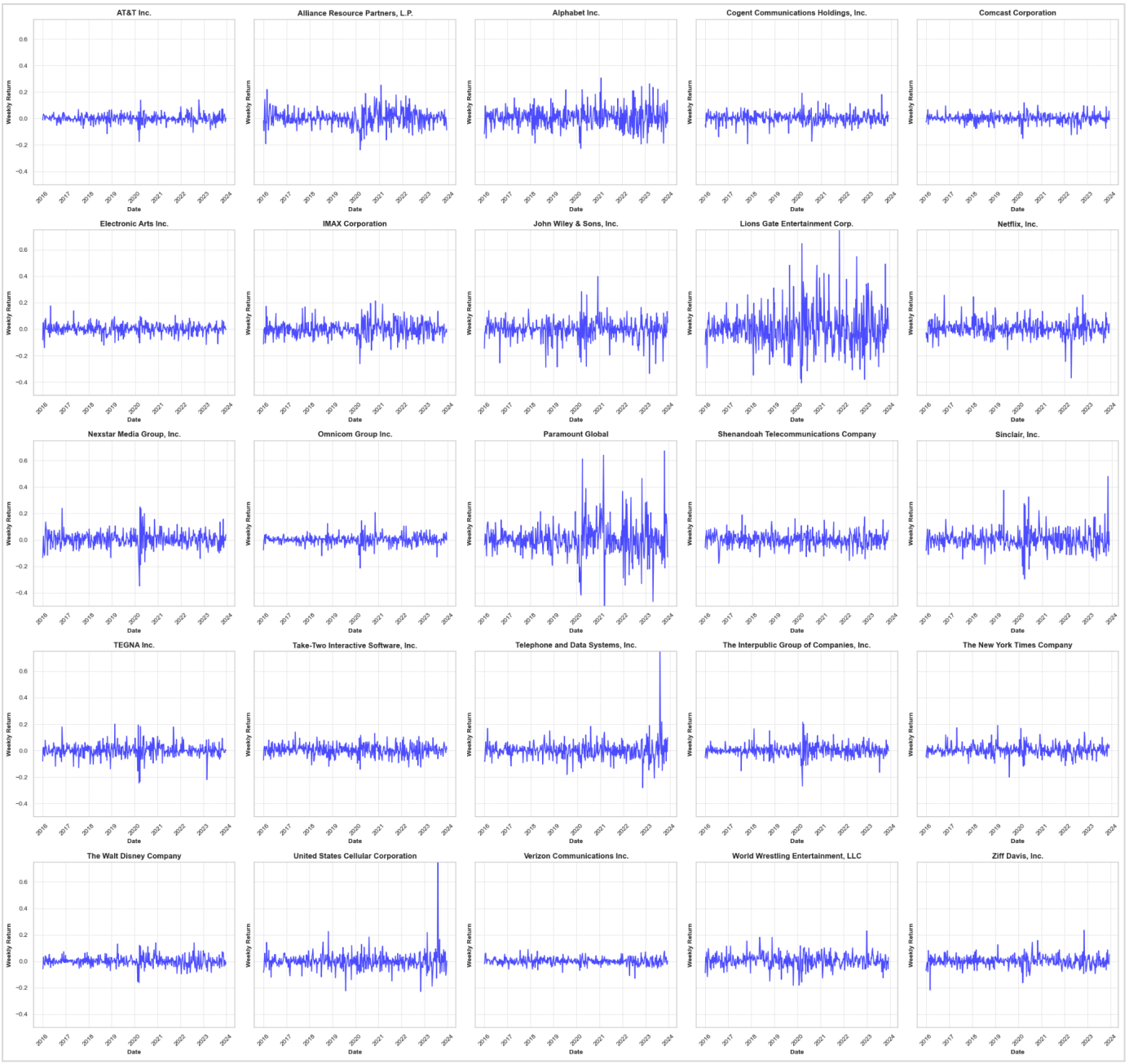
Appendix B. Summary statistics of weekly return all company

No	Ticker	count	mean	std	min	q1	q2	q3	max
1	GOOGL	991	0.006956	0.063279	-0.226603	-0.027436	0.007135	0.036506	0.596943
2	DIS	991	0.002038	0.03559	-0.220041	-0.01685	0.000651	0.019732	0.160438
3	NFLX	991	0.008233	0.072586	-0.368218	-0.029329	0.004971	0.041429	0.70979
4	T	991	0.001357	0.029655	-0.189807	-0.014286	0.000891	0.016607	0.140761
5	CMCSA	991	0.004918	0.066175	-0.300594	-0.026259	0.002003	0.031566	0.564535
6	VZ	991	0.001347	0.027135	-0.128871	-0.014429	0.001186	0.01634	0.183014
7	PARA	991	0.007398	0.121324	-0.747803	-0.049161	0.004355	0.055571	0.779581
8	EA	991	0.001876	0.046061	-0.175372	-0.023307	0.002392	0.02697	0.252592
9	TTWO	991	0.003558	0.056641	-0.317409	-0.025348	0.003366	0.032079	0.526498
10	GCI	991	0.001405	0.06221	-0.309527	-0.023766	0.000359	0.024484	0.498432
11	OMC	991	0.001803	0.034112	-0.213004	-0.015565	0.002005	0.020146	0.206783
12	IPG	991	0.002518	0.049037	-0.268264	-0.02094	0.000836	0.026805	0.229858
13	NYT	991	0.001877	0.054126	-0.27248	-0.022565	-0.000609	0.023862	0.423671
14	NXST	991	0.007267	0.096741	-0.443716	-0.032464	0.004848	0.042007	1.132812
15	WWE	991	0.004104	0.051425	-0.347046	-0.02205	0.003174	0.030026	0.293192
16	ZD	991	0.00278	0.045947	-0.236577	-0.021563	0.001912	0.026021	0.315364
17	LGF	991	0.004318	0.103868	-0.406322	-0.037402	0.001351	0.039848	0.781998
18	USM	991	0.001594	0.057517	-0.320665	-0.022969	0.000728	0.024425	0.888143
19	TDS	991	0.002339	0.076904	-0.524436	-0.030282	-0.001001	0.0336	0.848071
20	CCOI	991	0.004506	0.071181	-0.700179	-0.024925	0.00341	0.035339	0.578946
21	ARLP	991	0.004199	0.078264	-0.313432	-0.037042	0.000927	0.037297	0.490566
22	SBGI	991	0.003083	0.054803	-0.297794	-0.025542	0.001846	0.031268	0.386911
23	JW	991	0.001896	0.075018	-0.405347	-0.03004	0.002101	0.036611	0.715546
24	SHEN	991	0.003824	0.057182	-0.177703	-0.032294	0.003415	0.036791	0.281814
25	IMAX	991	0.002876	0.067145	-0.408602	-0.032889	0.002567	0.035972	0.564103

Note: *This table presents the summary statistics of weekly returns for all companies in the study. The columns include the ticker symbol, count of observations, mean return, standard deviation (Std Dev), minimum (Min), first quartile (Q1), median (Q2), third quartile (Q3), and maximum (Max) weekly returns for each company. This provides a comprehensive view of the return distribution for each stock.*

Appendix C. Weekly Return each company in-sample (top) and out-of-sample (bottom) datasets





Note: *These visualisations depict the weekly returns for each company in the US Communication Service Industry sector. The top row represents the in-sample dataset, while the bottom row shows the out-of-sample dataset. Each chart illustrates the volatility and return trends over the respective periods, providing a comprehensive view of the stock performance for each company.*

Appendix D. Word cloud each company in-sample (top) and out-of-sample (bottom) datasets

Note: *These word clouds illustrate the most frequently occurring tokens in the headline news for each company in the US Communication Service Industry sector. The top row represents the in-sample dataset, while the bottom row shows the out-of-sample dataset. The size of each word indicates its frequency, providing insights into common themes and topics discussed in the headlines for each company.*

Appendix E. Models accuracy throughout the year

	BERT		RoBERTa		Distil BERT		Distil RoBERTa		FinBERT	
	UT	FT	UT	FT	UT	FT	UT	FT	UT	FT
2016	51.4	54.13	52.3	54.05	52.2	52.64	51.4	52.05	51.2	52.65
2017	49.2	49.92	50.6	48.62	49.8	49.7	49.8	51.27	49.4	51.21
2018	49.2	48.36	50.1	48.99	49	49.94	50.1	49.52	50.3	49.24
2019	49.8	52.22	51	53.08	50.2	50.38	52.1	52.21	50.5	49.9
2020	49	52.29	52.1	52.01	51.8	52.23	51.4	52.76	51.7	52.27
2021	48.8	51.81	49.5	52.14	49.6	49.64	49.6	50.06	48.7	48.29
2022	48.3	46.26	49.2	46.44	48.6	49.03	48.5	48.45	49.7	48.92
2023	50	50.38	51	50.03	51.5	50.09	51.1	49.57	51.4	50.24
Avg	49.46	50.74	50.63	50.67	50.34	50.46	50.50	50.67	50.36	50.34

Note: *The table above presents the accuracy of various models (BERT, RoBERTa, DistilBERT, DistilRoBERTa, FinBERT) in their untuned (UT) and fine-tuned (FT) states from 2016 to 2023. This data illustrates how fine-tuning generally improves the accuracy of the models over time.*

Appendix F. Models' accuracy standard deviation throughout the year

	BERT		RoBERTa		Distil BERT		Distil RoBERTa		FinBERT	
	UT	FT	UT	FT	UT	FT	UT	FT	UT	FT
2016	3.95	2.77	5.21	3.65	4.74	3.32	5.09	3.56	5.9	4.13
2017	6.0	4.2	7.36	5.15	6.1	4.27	6.69	4.68	4.75	3.33
2018	5.11	3.58	5.77	4.04	5.62	3.93	5.28	3.7	4.25	2.98
2019	5.9	4.13	6.51	4.56	5.45	3.82	5.2	3.64	4.97	3.48
2020	6.46	4.52	6.98	4.89	5.09	3.56	5.46	3.82	7.54	5.28
2021	5.57	3.9	4.96	3.47	5.07	3.55	5.03	3.52	5.57	3.9
2022	6.55	4.59	6.01	4.21	6.31	4.42	5.27	3.69	5.83	4.08
2023	5.54	3.88	5.15	3.61	4.2	2.94	6.06	4.24	4.95	3.47
Avg	5.64	3.95	5.99	4.20	5.32	3.73	5.51	3.86	5.47	3.83

Note: *The table above details the standard deviation of accuracy for different models (BERT, RoBERTa, DistilBERT, DistilRoBERTa, FinBERT) in their untuned (UT) and fine-tuned (FT) states from 2016 to 2023. This data provides insight into the consistency of model performance, highlighting the reduced variance in accuracy achieved through fine-tuning.*

Dataset Loading

```
#Import important libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import pandas_market_calendars as mcal


# Load the CRSP and IQ Key Development dataset
CRSP = pd.read_csv('CRSP - Communication Services Dataset.csv')
IQKD = pd.read_csv('IQ Key Development - Communication Services Dataset.csv')
```

CRSP Dataset Preprocessing

Organizing the Company Name and Tickers of CRSP Dataset

```
# Columns to be dropped as it's unusable
columns_to_remove_atCRSP = ['PERMNO']
CRSP = CRSP.drop(columns=columns_to_remove_atCRSP)

# Display the modified DataFrame
CRSP.head(3)


# Group by 'PERMCO' and select the first 'TICKER' and 'COMNAM' for each group
first_values = CRSP.groupby('PERMCO').first()

# Map these first values back to the original DataFrame
CRSP['TICKER'] = CRSP['PERMCO'].map(first_values['TICKER'])
CRSP['COMNAM'] = CRSP['PERMCO'].map(first_values['COMNAM'])

# Now CRSP DataFrame should have standardized 'TICKER' and 'COMNAM' for each 'PERMCO'
```

```
#Summarize of the dataset
def summarize_data(df, name):
    print(f"Summary for {name}:")
    print("Total columns:", df.shape[1])
    print("Total rows:", df.shape[0])
    print("Missing values per column:")
    print(df.isnull().sum())
    print("\n")
summarize_data(CRSP, "CRSP Dataset")


# Display all rows
pd.set_option('display.max_rows', None)

# Group by 'PERMCO' and aggregate the 'date' column
result = CRSP.groupby('PERMCO')['date'].agg(['min', 'max', 'count'])

# Convert 'min' column to datetime to compare with January 2005
result['min'] = pd.to_datetime(result['min'])

# Filter out rows where 'min' date is earlier than January 2005
filtered_result = result[result['min'] <= '2005-01-03']

# Show the filtered result
print(filtered_result)

# Add the total count of unique PERMCO
total_unique_permco = filtered_result.shape[0]
print(f"Total count of unique PERMCO: {total_unique_permco}")



# Ensure 'date' is in the correct datetime format
CRSP['date'] = pd.to_datetime(CRSP['date'])

# Convert 'PRC' and 'SHROUT' to numeric to avoid any type issues, dropping NaN values
CRSP.dropna(subset=['PRC', 'SHROUT'], inplace=True)
CRSP['PRC'] = pd.to_numeric(CRSP['PRC'], errors='coerce')
CRSP['SHROUT'] = pd.to_numeric(CRSP['SHROUT'], errors='coerce')

# Group by 'COMNAM' and calculate maximum market cap observed on the fly, and min/max dat
market_cap_result = CRSP.groupby('COMNAM').agg({
    'PRC': lambda x: (x * CRSP.loc[x.index, 'SHROUT']).max(),
    'date': ['min', 'max']
}).reset_index()

# Rename columns to make the DataFrame easier to read
market_cap_result.columns = ['COMNAM', 'max_market_cap', 'min_date', 'max_date']

# Sort the results by market cap in descending order
market_cap_result = market_cap_result.sort_values(by='max_market_cap', ascending=False).r

# Display the result
market_cap_result
```

```
# Count the unique PERMCO values
unique_permco_count = CRSP['PERMCO'].nunique()

# Display the count of unique PERMCOs
print(f"Total count of unique PERMCO: {unique_permco_count}")



# Check for any negative values in the 'PRC' column
negative_prc = CRSP[CRSP['PRC'] < 0]

# Display the entries with negative 'PRC' values, if any
if not negative_prc.empty:
    print("Negative PRC values found:")
    print(negative_prc)
else:
    print("No negative PRC values found.")



from IPython.display import display

# Create a DataFrame showing each PERMCO with its associated COMNAM
comnam_tickers = CRSP.groupby('PERMCO')['COMNAM'].apply(set).reset_index()

# Name the columns as requested
comnam_tickers.columns = ['PERMCO', 'Associated Company Name']

# Display the DataFrame in a scrollable output window in Jupyter Notebook
display(comnam_tickers.style.set_table_attributes('style="display:block; height:200px; o



CRSP.head(5)


#Summarize of the dataset
def summarize_data(df, name):
    print(f"Summary for {name}:")
    print("Total columns:", df.shape[1])
    print("Total rows:", df.shape[0])
    print("Missing values per column:")
    print(df.isnull().sum())
    print("\n")
summarize_data(CRSP, "CRSP Dataset")


# Filter out the rows where 'COMNAM' is 'MILLICOM INTL CELLULAR SA' or 'CHARTER COMMUNIC
CRSP = CRSP[~CRSP['COMNAM'].isin(['MILLICOM INTL CELLULAR SA', 'CHARTER COMMUNICATIONS I
```

```
CRSP['date'] = pd.to_datetime(CRSP['date'])

# Determine the overall date range in the dataset
min_date = CRSP['date'].min()
max_date = CRSP['date'].max()
full_date_range = pd.date_range(start=min_date, end=max_date, freq='D')

# Function to calculate missing dates for a given ticker
def calculate_missing_dates(ticker_data, full_date_range):
    ticker_dates = pd.DatetimeIndex(ticker_data['date'])
    missing_dates = full_date_range.difference(ticker_dates)
    return missing_dates

# Apply the function to each ticker and store the results in a DataFrame
results = {}
for COMNAM in CRSP['COMNAM'].unique():
    ticker_data = CRSP[CRSP['COMNAM'] == COMNAM]
    missing_dates = calculate_missing_dates(ticker_data, full_date_range)
    results[COMNAM] = len(missing_dates)

# Convert results dictionary to DataFrame for better readability and analysis
results_df = pd.DataFrame(list(results.items()), columns=['COMNAM', 'Missing Dates Count

# Display the results
print(results_df)
```

CRSP with DropNA (Drop the missing value, gap, or uncommon value)

```
# Drop rows with any missing values
CRSP.dropna(inplace=True)

# Drop rows where the 'PRC' column contains negative values
CRSP = CRSP[CRSP['PRC'] > 0]

# Ensure 'RET' contains only numerical values and drop rows with non-numerical values
CRSP = CRSP[pd.to_numeric(CRSP['RET'], errors='coerce').notna()]


# Print unique PERMCO values and their count
PERMCO_unique = CRSP['PERMCO'].unique()
print("Unique PERMCO values in CRSP")
PERMCO_unique
print("Count of unique COMNAM values:", len(PERMCO_unique))
```

```
#Summarize of the dataset
def summarize_data(df, name):
    print(f"Summary for {name}:")
    print("Total columns:", df.shape[1])
    print("Total rows:", df.shape[0])
    print("Missing values per column:")
    print(df.isnull().sum())
    print("\n")
summarize_data(CRSP, "CRSP Dataset")


# Group by 'PERMCO' and aggregate the 'date' column to find min and max dates
final_CRSP_processing_check = CRSP.groupby('PERMCO').agg(
    min_date=('date', 'min'),
    max_date=('date', 'max')
).reset_index()

comnam_mapping = CRSP[['PERMCO', 'COMNAM']].drop_duplicates().set_index('PERMCO')
final_CRSP_processing_check['COMNAM'] = final_CRSP_processing_check['PERMCO'].map(comnam

# Show the result
final_CRSP_processing_check
```

IQ Key Development Dataset Preprocessing

Organize Company Name and Ticker of IQKD Dataset

```
#Summarize of the dataset
def summarize_data(df, name):
    print(f"Summary for {name}:")
    print("Total columns:", df.shape[1])
    print("Total rows:", df.shape[0])
    print("Missing values per column:")
    print(df.isnull().sum())
    print("\n")
summarize_data(IQKD, "IQ Key Development Dataset")


# Columns to be removed
columns_to_remove = ['companyid', 'sptodate']

# Drop the specified columns from the IQKD DataFrame
IQKD= IQKD.drop(columns=columns_to_remove)

# Display the modified DataFrame
IQKD.head(3)
```

```
# Display the unique gvkey and companyname
unique_gvkey_companyname = IQKD[['gvkey', 'companyname']].drop_duplicates().reset_index(

# Print the unique gvkey and companyname
print(unique_gvkey_companyname)



# Remove rows with specified company names
companies_to_remove = ['M&A Rumors and Discussions', 'Millicom International Cellular S.A
IQKD = IQKD[~IQKD['companyname'].isin(companies_to_remove)].reset_index(drop=True)

# Fill NaN gvkey based on similar companyname
# Create a mapping of companyname to gvkey for non-NaN gvkey
company_to_gvkey_map = IQKD.dropna(subset=['gvkey']).set_index('companyname')['gvkey'].to

# Fill NaN gvkey based on the company_to_gvkey_map
IQKD['gvkey'] = IQKD.apply(
    lambda row: company_to_gvkey_map.get(row['companyname'], row['gvkey']),
    axis=1
)

# Remove duplicate rows based on both gvkey and companyname
IQKD = IQKD.drop_duplicates().reset_index(drop=True)



# Group by 'PERMCO' and select the first 'gvkey' and 'companyname' for each group
first_values = IQKD.groupby('gvkey').first()

# Map these first values back to the original DataFrame
IQKD['companyname'] = IQKD['gvkey'].map(first_values['companyname'])

# Now CRSP DataFrame should have standardized 'gvkey' and 'companyname' for each 'gvkey'



#Summarize of the dataset
def summarize_data(df, name):
    print(f"Summary for {name}:")
    print("Total columns:", df.shape[1])
    print("Total rows:", df.shape[0])
    print("Missing values per column:")
    print(df.isnull().sum())
    print("\n")
summarize_data(IQKD, "IQ Key Development Dataset")
```

```
IQKD['announcedate'] = pd.to_datetime(IQKD['announcedate'], errors='coerce')

# Group by the new 'companyname' and analyze the 'announcedate' column
grouped_GVKeycompany = IQKD.groupby('companyname')['announcedate'].agg(['min', 'max', 'c

# Display the results
print("Unified date range and count of entries for each company before merging:")
grouped_GVKeycompany



#Summarize of the dataset
def summarize_data(df, name):
    print(f"Summary for {name}:")
    print("Total columns:", df.shape[1])
    print("Total rows:", df.shape[0])
    print("Missing values per column:")
    print(df.isnull().sum())
    print("\n")
summarize_data(IQKD, "IQ Key Development Dataset")
```

Removing Redundant Headline on the Same Day for every company

```
# Group by 'gvkey' and 'announcedate' and collect headlines
headline_groups = IQKD.groupby(['gvkey', 'announcedate'])['headline'].agg(list)

# Check for redundant headlines within each group
redundant_headlines = headline_groups.apply(lambda headlines: len(headlines) != len(set(h

# Filter to show only the groups where there are redundant headlines
redundant_headlines = redundant_headlines[redundant_headlines]

# Join this with the original DataFrame to get the rows with redundant headlines
redundant_data = IQKD[IQKD.set_index(['gvkey', 'announcedate']).index.isin(redundant_hea

# Display the data
print("Data with redundant headlines:")
redundant_data.head(2)
```

```
# Filter for the specific ticker and date
specific_data = IQKD[(IQKD['gvkey'] == '126136') & (pd.to_datetime(IQKD['announcedate'])

# Display the relevant rows
print("Entries for Charter Communications:")
print(specific_data)

# To explicitly check for redundant headlines
if specific_data['headline'].duplicated().any():
    print("Redundant headlines found.")
else:
    print("No redundant headlines.")



# Count rows per Company Name before removing duplicates
count_before = IQKD.groupby('companyname').size()

# Remove rows where 'Company Name', 'announcedate', and 'headline' are identical, keeping
IQKD_cleaned = IQKD.drop_duplicates(subset=['companyname', 'announcedate', 'headline'], k

# Count rows per Company Name after removing duplicates
count_after = IQKD_cleaned.groupby('companyname').size()

# Combine counts into a DataFrame for side-by-side comparison
comparison_df = pd.DataFrame({
    'Before': count_before,
    'After': count_after
})

# Fill NaN values with 0 if there are any, in case some Company Name disappear completely
comparison_df.fillna(0, inplace=True)

# Print the comparison DataFrame
print("Comparison of row counts per Company Name before and after removing Headlines dupl
print(comparison_df)



# Remove rows where 'gvkey', 'announcedate', and 'headline' are identical except keep the
IQKD = IQKD.drop_duplicates(subset=['gvkey', 'announcedate', 'headline'], keep='first')


#Summarize of the dataset
def summarize_data(df, name):
    print(f"Summary for {name}:")
    print("Total columns:", df.shape[1])
    print("Total rows:", df.shape[0])
    print("Missing values per column:")
    print(df.isnull().sum())
    print("\n")
summarize_data(IQKD, "IQ Key Development Dataset")
```

Dropping the rows that having missing announcedate

```
rows_with_missing_values = IQKD[IQKD.isnull().any(axis=1)]

# Show the first 10 rows that will be dropped
rows_with_missing_values.head(5)


# Drop rows where 'announcedate' is NaT
IQKD = IQKD[pd.notnull(IQKD['announcedate'])]


#Summarize of the dataset
def summarize_data(df, name):
    print(f"Summary for {name}:")
    print("Total columns:", df.shape[1])
    print("Total rows:", df.shape[0])
    print("Missing values per column:")
    print(df.isnull().sum())
    print("\n")
summarize_data(IQKD, "IQ Key Development Dataset")
```

Put [No_Headline] placeholders on the date where no headlines

```
IQKD['announcedate'] = pd.to_datetime(IQKD['announcedate'])

# Sort the DataFrame by 'gvkey' and 'announcedate' to ensure correct date difference cal
IQKD.sort_values(['gvkey', 'announcedate'], inplace=True)

# Calculate the difference between consecutive dates for each gvkey
IQKD['date_diff'] = IQKD.groupby('gvkey')['announcedate'].diff().dt.days - 1

# Filter to show only the rows where there is a gap (date_diff > 0)
date_gaps = IQKD[IQKD['date_diff'] > 0]

# Optional: summarize the maximum gap for each gvkey
max_gaps = date_gaps.groupby('gvkey')['date_diff'].max()

# Print or view the summary of maximum gaps
print("Maximum date gaps for each company:")
print(max_gaps)
```

```
# Sort the DataFrame by 'gvkey' and 'announcedate' to ensure correct date difference cal
IQKD.sort_values(['gvkey', 'announcedate'], inplace=True)

# Calculate the difference between consecutive dates for each gvkey
IQKD['date_diff'] = IQKD.groupby('gvkey')['announcedate'].diff().dt.days

# Filter to show only the rows where there is a gap (date_diff > 0)
date_gaps = IQKD[IQKD['date_diff'] > 0]

# Summarize the average gap for each gvkey
average_gaps = date_gaps.groupby('gvkey')['date_diff'].mean().reset_index()

# Retrieve the first company name for each gvkey
first_company_name = IQKD.drop_duplicates(subset='gvkey')[['gvkey', 'companyname']]

# Merge the average gaps with the first company name
average_gaps = average_gaps.merge(first_company_name, on='gvkey')

# Print or view the summary of average gaps with gvkey and companyname
print("Average date gaps for each gvkey and companyname:")
print(average_gaps)



# Filter to find gaps specifically for Verizon
Verizon_gaps = IQKD[(IQKD['gvkey'] == 2136.0) & (IQKD['date_diff'] > 0)]

# Print the rows with gaps for Google Inc (Alphabet)
print("Rows with gaps for Verizon:")
Verizon_gaps.head(10)



# Ensure 'announcedate' is in datetime format and sort the DataFrame
IQKD['announcedate'] = pd.to_datetime(IQKD['announcedate'])
IQKD.sort_values(['gvkey', 'announcedate'], inplace=True)

# Function to fill missing dates for each gvkey
def fill_missing_dates(group):
    all_dates = pd.date_range(start=group['announcedate'].min(), end=group['announcedate
    all_dates_df = pd.DataFrame(all_dates, columns=['announcedate'])
    merged = all_dates_df.merge(group, on='announcedate', how='left')
    if merged.isna().sum().sum() > 0:  # Check if there are any missing values to fill
        merged['gvkey'] = group['gvkey'].iloc[0]
        merged['companyname'] = group['companyname'].iloc[0]
        merged['headline'].fillna('[No_Headline]', inplace=True)
        merged['eventtype'].fillna('[No_Event]', inplace=True)
    return merged

# Apply the function to each gvkey group
IQKD = IQKD.groupby('gvkey').apply(fill_missing_dates).reset_index(drop=True)

# Print or examine updated DataFrame
IQKD.head(10)
```

```
# Filter the DataFrame to show rows where 'headline' is '[No_Headline]'
no_headline_rows = IQKD[IQKD['headline'] == '[No_Headline]']

# Display these rows.
no_headline_rows.head(10)


IQKD.sort_values(['gvkey', 'announcedate'], inplace=True)

# Filter the DataFrame to show rows where 'headline' is '[No_Headline]'
no_headline_rows = IQKD[IQKD['headline'] == '[No_Headline]']

# Group by 'companyname' and 'announcedate' and count occurrences
no_headline_counts = no_headline_rows.groupby(['companyname', 'announcedate']).size()

# Filter to keep only groups with more than one occurrence
multiple_no_headline = no_headline_counts[no_headline_counts > 1].reset_index()

if multiple_no_headline.empty:
    print("There are no more than 1 [No_Headline] on each company for the same announceda
else:
    # Merge to get detailed rows for these groups
    detailed_no_headline_rows = no_headline_rows.merge(multiple_no_headline, on=['company

    # Display these rows
    print("Rows with more than 1 no headline on the same announcedate for each company:"
    print(detailed_no_headline_rows.head(5))



Verizon_data = IQKD[IQKD['companyname'] == 'Verizon Communications Inc.']

print("Head rows for Verizon:")
Verizon_data.head(10)
```

Add 'Situation' from WRDS Cloud

```
import wrds
db = wrds.Connection(wrds_username='baguspranata')
db.create_pgpass_file()


db.close()
db = wrds.Connection(wrds_username='baguspranata')


table_description = db.describe_table(library='ciq_keydev', table='ciqkeydev')
table_description


unique_keydevids = IQKD['keydevid'].unique()
```

```
# Convert the list of keydevids to a format suitable for SQL queries
keydevid_list = ','.join([f"'{str(k)}'" for k in unique_keydevids])

# Construct the SQL query
sql_query = f"""
SELECT keydevid, headline, situation, announceddate
FROM ciq_keydev.ciqkeydev
WHERE keydevid IN ({keydevid_list})
"""

# Fetch the data
matched_data = db.raw_sql(sql_query)
matched_data.head()


# Convert keydevid in IQKD to integer (remove non-numeric entries first if necessary)
IQKD['keydevid'] = pd.to_numeric(IQKD['keydevid'], errors='coerce').dropna().astype(int)

# Convert keydevid in matched_data to integer
matched_data['keydevid'] = pd.to_numeric(matched_data['keydevid'], errors='coerce').dropı

# Merge the situation data from matched_data into IQKD based on keydevid
IQKD = pd.merge(IQKD, matched_data[['keydevid', 'situation']], on='keydevid', how='left'

# Check and fill missing situations after merge
IQKD['situation'] = IQKD['situation'].fillna('[No Situation]')

# Print the resulting DataFrame to check the results
IQKD.head(3)



# Filter the DataFrame to include only rows where 'situation' is not missing
filtered_IQKD = IQKD[IQKD['situation'].notna()]

# Print the filtered DataFrame
filtered_IQKD.head(2)
```

```
IQKD.sort_values(['gvkey', 'announcedate'], inplace=True)

# Filter the DataFrame to show rows where 'headline' is '[No_Headline]'
no_headline_rows = IQKD[IQKD['headline'] == '[No_Headline]']

# Group by 'companyname' and 'announcedate' and count occurrences
no_headline_counts = no_headline_rows.groupby(['companyname', 'announcedate']).size()

# Filter to keep only groups with more than one occurrence
multiple_no_headline = no_headline_counts[no_headline_counts > 1].reset_index()

if multiple_no_headline.empty:
    print("There are no more than 1 [No_Headline] on each company for the same announceda
else:
    # Merge to get detailed rows for these groups
    detailed_no_headline_rows = no_headline_rows.merge(multiple_no_headline, on=['company

    # Display these rows
    print("Rows with more than 1 no headline on the same announcedate for each company:"
    print(detailed_no_headline_rows.head(10))
```

Distribution of headlines throughout the time on every ticker

```
IQKD['announcedate'] = pd.to_datetime(IQKD['announcedate'])

# Group by 'ticker' and 'announcedate' and count the number of headlines
headline_counts = IQKD.groupby(['companyname', 'announcedate']).size().reset_index(name=

# Get unique tickers
unique_tickers = headline_counts['companyname'].unique()

# Calculate the number of rows needed for subplots (2 plots per row)
num_rows = (len(unique_tickers) + 1) // 2

# Create a figure and array of axes with 2 columns
fig, axes = plt.subplots(nrows=num_rows, ncols=2, figsize=(14, num_rows * 4))
axes = axes.flatten()  # Flatten to 1D array for easier iteration

# Plot each ticker's data in its respective subplot
for idx, companyname in enumerate(unique_tickers):
    ax = axes[idx]
    # Filter data for the current companyname
    ticker_data = headline_counts[headline_counts['companyname'] == companyname]

    # Create the line plot on the specified axes
    sns.lineplot(ax=ax, data=ticker_data, x='announcedate', y='headline_count', marker='

    # Adding plot details
    ax.set_title(f'Number of Headlines Over Time for {companyname}')
    ax.set_xlabel('Date')
    ax.set_ylabel('Number of Headlines')
    ax.legend()
    ax.grid(True)

# If the number of tickers is odd, hide the last subplot (if unused)
if len(unique_tickers) % 2 != 0:
    axes[-1].set_visible(False)

plt.tight_layout()
plt.show()
```

CRSP Weekly Return Computation

```
CRSP.head(2)
```

Checking the frequency of outliers of PRC and RET on each TICKERs

```
# Convert 'RET' and 'PRC' to numeric, coercing errors to NaN
CRSP['RET'] = pd.to_numeric(CRSP['RET'], errors='coerce')
CRSP['PRC'] = pd.to_numeric(CRSP['PRC'], errors='coerce')

# Determine the first company name for each PERMCO
first_comnam = CRSP.sort_values(by='date').groupby('PERMCO')['COMNAM'].first().reset_ind

# Map this first company name back to the main DataFrame using PERMCO
CRSP = CRSP.merge(first_comnam, on='PERMCO', suffixes=('', '_first'))

# Handle NaN values if necessary
# CRSP.dropna(subset=['RET', 'PRC'], inplace=True)  # Option to drop NaNs

# Calculate IQR and bounds within a function to ensure proper grouping
def calculate_outliers_and_proportions(group):
    Q1 = group[['RET', 'PRC']].quantile(0.25)
    Q3 = group[['RET', 'PRC']].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Count outliers
    ret_outliers = ((group['RET'] < lower_bound['RET']) | (group['RET'] > upper_bound['R
    prc_outliers = ((group['PRC'] < lower_bound['PRC']) | (group['PRC'] > upper_bound['P

    # Calculate proportions
    ret_outliers_count = ret_outliers.sum()
    prc_outliers_count = prc_outliers.sum()
    ret_proportion = ret_outliers_count / len(group)
    prc_proportion = prc_outliers_count / len(group)

    return pd.Series({'RET_Outliers': ret_outliers_count, 'PRC_Outliers': prc_outliers_c
                      'RET_Outlier_Proportion': ret_proportion, 'PRC_Outlier_Proportion'

# Apply function to each group based on the first COMNAM
outlier_counts = CRSP.groupby('COMNAM').apply(calculate_outliers_and_proportions)

# Print the result
outlier_counts



CRSP.head(2)
```

Weekly Return Computation Without PRC & RET Outliers Removal

```
import pandas as pd
import numpy as np

# Assuming CRSP is your DataFrame

# Step 1: Convert date to datetime and other data to numeric
CRSP['date'] = pd.to_datetime(CRSP['date'])
CRSP['RET'] = pd.to_numeric(CRSP['RET'], errors='coerce')
CRSP['PRC'] = pd.to_numeric(CRSP['PRC'], errors='coerce')
CRSP['SHROUT'] = pd.to_numeric(CRSP['SHROUT'], errors='coerce')

# Step 2: Filter the data
NonMissing_CRSP = CRSP[(CRSP['PRC'] > 0) & (CRSP['RET'].notna())]

# Step 3: Calculate weekly returns and market cap
# Set index to date for resampling
NonMissing_CRSP.set_index('date', inplace=True)

# Resample to get weekly data
weekly_CRSP = NonMissing_CRSP.groupby(['PERMCO', 'COMNAM']).resample('W-FRI').agg({
    'PRC': 'last',  # Get the last price of the week
    'RET': lambda x: np.exp(np.sum(np.log(1 + x))) - 1,  # Compound weekly return
    'SHROUT': 'last'  # Share outstanding at the end of the week
}).reset_index()

# Calculate market cap
weekly_CRSP['market_cap'] = weekly_CRSP['PRC'] * weekly_CRSP['SHROUT']

# Rename the column to indicate it is the weekly compound return
weekly_CRSP.rename(columns={'RET': 'Weekly Compound Return'}, inplace=True)

# Calculate Return Direction based on price movement compared to the previous week
weekly_CRSP['Past Return Direction'] = np.select(
    [
        weekly_CRSP['PRC'].diff() > 0,
        weekly_CRSP['PRC'].diff() < 0
    ],
    [
        'Up',  # Price went up
        'Down'  # Price went down
    ],
    default='No Change'  # No change in price
)

# Calculate Future Direction by comparing with the next week's price
weekly_CRSP['Future Return Direction'] = np.select(
    [
        weekly_CRSP['PRC'].shift(-1) > weekly_CRSP['PRC'],
        weekly_CRSP['PRC'].shift(-1) < weekly_CRSP['PRC']
    ],
    [
        'Up',  # Price will go up next week
        'Down'  # Price will go down next week
    ],
    default='No Change'  # No change in price next week
```

```
)

# Display the head of the modified DataFrame
print(weekly_CRSP.head())



# Calculate the 'Date From' by subtracting 6 days from 'date' (since 'date' is the end o
weekly_CRSP['Date From'] = weekly_CRSP['date'] - pd.Timedelta(days=6)

# 'Date To' is just the 'date' column, which represents the week ending
weekly_CRSP['Date To'] = weekly_CRSP['date']

# Rearrange columns for better readability, if necessary
weekly_CRSP = weekly_CRSP[['COMNAM','PERMCO','date', 'Date From', 'Date To','market_cap'
```

```
# Provided PERMCO-GVKEY pairs
permco_gvkey_mapping = {
    45483: 160329,
    21645: 9899,
    11204: 24708,
    16779: 122915,
    17322: 126136,
    42769: 147204,
    43613: 3226,
    10303: 16721,
    1908: 5284,
    13136: 30312,
    4922: 11499,
    34913: 28378,
    12746: 30024,
    43145: 147579,
    44625: 149177,
    1367: 4066,
    21866: 13714,
    15708: 65460,
    11358: 9466,
    40213: 9664,
    13758: 60800,
    15436: 64630,
    20782: 4988,
    5230: 10411,
    20997: 6136,
    21280: 7866,
    20587: 3980,
    17205: 125240,
    21826: 14369,
    20288: 2136,
    16665: 122172
}

# Mapping the gvkey to the weekly_CRSP DataFrame
weekly_CRSP['gvkey'] = weekly_CRSP['PERMCO'].map(permco_gvkey_mapping)



# Display the DataFrame
weekly_CRSP.tail(10)


# Count unique TICKER values in the weekly_CRSP DataFrame
unique_PERMCO_count = weekly_CRSP['PERMCO'].nunique()
print(f"Number of unique PERMCOs: {unique_PERMCO_count}")
```

```
# Find the latest date
latest_date = weekly_CRSP['Date To'].max()

# Display the latest date
print("The latest date in the 'Date To' column is:", latest_date)



# Make sure the 'Date To' column is in datetime format
weekly_CRSP['Date To'] = pd.to_datetime(weekly_CRSP['Date To'])

# Set Seaborn's aesthetic parameters to make the plots more visually appealing
sns.set(style="whitegrid")

# Generate a list of years to use as x-axis ticks
years = pd.date_range(start='2005-01-01', end='2024-12-31', freq='YS').year

# Iterate over each group defined by 'COMNAM'
for ticker, group in weekly_CRSP.groupby('COMNAM'):
    fig, ax = plt.subplots(figsize=(12, 2))  # Create a new figure for each ticker
    group = group.sort_values('Date To')  # Ensure data is sorted by date for plotting

    # Use Seaborn's lineplot for better aesthetics; since seaborn is based on matplotlib
    sns.lineplot(data=group, x='Date To', y='Weekly Compound Return', marker='o', linesty

    # Set title and labels with enhanced formatting
    ax.set_title(f'Weekly Compound Return for {ticker}', fontsize=12)
    ax.set_xlabel('Date', fontsize=10)
    ax.set_ylabel('Weekly Compound Return', fontsize=10)
    ax.legend()

    # Set x-ticks to display each year
    ax.set_xticks(pd.to_datetime(years, format='%Y'))
    ax.set_xticklabels(years, rotation=45)

    # Show the plot
    plt.show()
```

```
# Setting Seaborn's aesthetic parameters for better visual effects
sns.set(style="whitegrid", palette="pastel")

# Assuming 'weekly_CRSP' is your DataFrame
all_tickers = weekly_CRSP['COMNAM'].unique()  # Get all unique tickers

# Setup the figure size to fit all tickers horizontally with improved vertical dimension
plt.figure(figsize=(0.5 * len(all_tickers), 5))  # Increase the height for better visibi

# Create a seaborn boxplot across a single axis without subplot division
sns.boxplot(x='COMNAM', y='Weekly Compound Return', data=weekly_CRSP, width=0.6)

# Set the plot title and labels with improved formatting
plt.title('Weekly Returns for All COMNAM', fontsize=15)
plt.xlabel('COMNAM', fontsize=14)
plt.ylabel('Weekly Compound Return', fontsize=14)
plt.xticks(rotation=90)  # Rotate ticker labels for better readability if necessary

plt.grid(True)  # Ensure the grid is visible for better readability
plt.show()
```

Merging & Split Weekly Return CRSP with IQKD and stored it into desired dataframe structure

```
#To avoiding the memory error issue, save the previous IQKD and weekly_CRSP that had beer

# Save the IQKD DataFrame to a CSV file
IQKD.to_csv('cleaned_IQKD.csv', index=False)

# Save the weekly_CRSP DataFrame to a CSV file
weekly_CRSP.to_csv('cleaned_weekly_CRSP.csv', index=False)
```

```
# Reload the CRSP and IQ Key Development dataset
weekly_CRSP = pd.read_csv('cleaned_weekly_CRSP.csv')
IQKD = pd.read_csv('cleaned_IQKD.csv')
```

```
# Convert 'announcedate' to datetime format
IQKD['announcedate'] = pd.to_datetime(IQKD['announcedate'])

# Filter out rows where headline is '[No_Headline]'
filtered_IQKD = IQKD[IQKD['headline'] != '[No_Headline]']

# Extract year and month from 'announcedate'
filtered_IQKD['year'] = filtered_IQKD['announcedate'].dt.year
filtered_IQKD['month'] = filtered_IQKD['announcedate'].dt.month_name()

# Set the style of the plots
sns.set(style="whitegrid")

# Define the color
viridis_blue = '#1f77b4'  # Hex code for a blue color in the Viridis palette

# Plot 1: Total Counts of headlines based on year
plt.figure(figsize=(10, 6))
yearly_counts = filtered_IQKD['year'].value_counts().sort_index()
sns.barplot(x=yearly_counts.index, y=yearly_counts.values, color=viridis_blue)
plt.title('Total Counts of Headlines Based on Year')
plt.xlabel('Year')
plt.ylabel('Total Counts')
plt.xticks(rotation=0)
plt.grid(axis='y', linestyle='--')
plt.show()

# Plot 2: Total Counts of headlines based on month
plt.figure(figsize=(12, 6))
monthly_counts = filtered_IQKD['month'].value_counts().reindex([
    'January', 'February', 'March', 'April', 'May', 'June', 'July',
    'August', 'September', 'October', 'November', 'December'
])
sns.barplot(x=monthly_counts.index, y=monthly_counts.values, color=viridis_blue)
plt.title('Total Counts of Headlines Based on Month')
plt.xlabel('Month')
plt.ylabel('Total Counts')
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--')
plt.show()
```

```
# Ensure that the 'Date From', 'Date To' in weekly_CRSP and 'announcedate' in IQKD are da
weekly_CRSP['Date From'] = pd.to_datetime(weekly_CRSP['Date From'], errors='coerce')
weekly_CRSP['Date To'] = pd.to_datetime(weekly_CRSP['Date To'], errors='coerce')
IQKD['announcedate'] = pd.to_datetime(IQKD['announcedate'], errors='coerce')

# Function to split DataFrame into in-sample and out-sample based on a split date
def split_dataframes(df, date_column, split_date):
    in_sample = df[df[date_column] < split_date]
    out_sample = df[df[date_column] >= split_date]
    return in_sample, out_sample

# Define the split date
split_date = pd.to_datetime('2016-01-01')

# Split weekly_CRSP DataFrame
InSample_weekly_CRSP, OutSample_weekly_CRSP = split_dataframes(weekly_CRSP, 'Date From',

# Split IQKD DataFrame
InSample_IQKD, OutSample_IQKD = split_dataframes(IQKD, 'announcedate', split_date)

# Display the resulting DataFrames
print("InSample_weekly_CRSP:")
print(InSample_weekly_CRSP.head(1))

print("\nOutSample_weekly_CRSP:")
print(OutSample_weekly_CRSP.head(1))

print("\nInSample_IQKD:")
print(InSample_IQKD.head(1))

print("\nOutSample_IQKD:")
print(OutSample_IQKD.head(1))



pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)  # Be cautious with this on large DataFrames



weekly_CRSP.tail(2)


IQKD.tail(10)



# Merge InSample DataFrames
insample_df = pd.merge(InSample_weekly_CRSP, InSample_IQKD, left_on='gvkey', right_on='gv
insample_df = insample_df[(insample_df['announcedate'] >= insample_df['Date From']) & (in
insample_df = insample_df[['companyname', 'Date From', 'Date To', 'date', 'Weekly Compou

# Print or return the resulting DataFrames
print("Merged In-Sample DataFrame:")
insample_df.head(5)
```

```
# Filter the DataFrame to show rows where 'headline' is '[No_Headline]'
no_headline_rows = insample_df[insample_df['headline'] == '[No_Headline]']

# Group by 'companyname' and 'announcedate' and count occurrences
no_headline_counts = no_headline_rows.groupby(['companyname', 'date']).size()

# Filter to keep only groups with more than one occurrence
multiple_no_headline = no_headline_counts[no_headline_counts > 1].reset_index()

# Merge to get detailed rows for these groups
detailed_no_headline_rows = no_headline_rows.merge(multiple_no_headline, on=['companynam

# Display these rows
print("INSAMPLE Rows with more than 1 no headline on the same date for each company:")
detailed_no_headline_rows.head(5)
```

```
# Filter the DataFrame to show rows where 'headline' is '[No_Headline]'
no_headline_rows = insample_df[insample_df['headline'] == '[No_Headline]']

# Group by 'companyname' and 'date' and count occurrences
no_headline_counts = no_headline_rows.groupby(['companyname', 'date']).size()

# Filter to keep only groups with more than one occurrence
multiple_no_headline = no_headline_counts[no_headline_counts > 1].reset_index()

# Merge to get detailed rows for these groups
detailed_no_headline_rows = no_headline_rows.merge(multiple_no_headline, on=['companynam

# Display these rows
print("INSAMPLE Rows with more than 1 no headline on the same date for each company:")
print(detailed_no_headline_rows.head(5))

# Remove duplicates keeping only the first occurrence
insample_df_cleaned = insample_df.drop_duplicates(subset=['companyname', 'date', 'headli

# Save the cleaned DataFrame back into insample_df
insample_df = insample_df_cleaned

# Verify the cleaning process
no_headline_cleaned = insample_df[insample_df['headline'] == '[No_Headline]']
multiple_no_headline_cleaned = no_headline_cleaned.groupby(['companyname', 'date']).size

# Filter to keep only groups with more than one occurrence after cleaning
multiple_no_headline_cleaned = multiple_no_headline_cleaned[multiple_no_headline_cleaned

# Check if there are any duplicates left
if not multiple_no_headline_cleaned.empty:
    print("There are still rows with more than 1 no headline on the same date for each c
    print(multiple_no_headline_cleaned.head(5))
else:
    print("No more duplicate 'No_Headline' rows found for the same date and company afte

# Display cleaned DataFrame
print("Cleaned INSAMPLE DataFrame:")
print(insample_df.head(5))



insample_df.head(5)


# Merge OutSample DataFrames
outsample_df = pd.merge(OutSample_weekly_CRSP, OutSample_IQKD, left_on='gvkey', right_on=
outsample_df = outsample_df[(outsample_df['announcedate'] >= outsample_df['Date From']) &
outsample_df = outsample_df[['companyname', 'Date From', 'Date To', 'date', 'Weekly Compo

print("\nMerged Out-Sample DataFrame:")
outsample_df.head(5)
```

```
# Filter the DataFrame to show rows where 'headline' is '[No_Headline]'
no_headline_rows = outsample_df[outsample_df['headline'] == '[No_Headline]']

# Group by 'companyname' and 'announcedate' and count occurrences
no_headline_counts = no_headline_rows.groupby(['companyname', 'date']).size()

# Filter to keep only groups with more than one occurrence
multiple_no_headline = no_headline_counts[no_headline_counts > 1].reset_index()

# Merge to get detailed rows for these groups
detailed_no_headline_rows = no_headline_rows.merge(multiple_no_headline, on=['companynam

# Display these rows
print("OUTSAMPLE Rows with more than 1 no headline on the same date for each company:")
detailed_no_headline_rows.head(5)
```

```
# Filter the DataFrame to show rows where 'headline' is '[No_Headline]'
no_headline_rows = outsample_df[outsample_df['headline'] == '[No_Headline]']

# Group by 'companyname' and 'date' and count occurrences
no_headline_counts = no_headline_rows.groupby(['companyname', 'date']).size()

# Filter to keep only groups with more than one occurrence
multiple_no_headline = no_headline_counts[no_headline_counts > 1].reset_index()

# Merge to get detailed rows for these groups
detailed_no_headline_rows = no_headline_rows.merge(multiple_no_headline, on=['companynam

# Display these rows
print("OUTSAMPLE Rows with more than 1 no headline on the same date for each company:")
print(detailed_no_headline_rows.head(5))

# Remove duplicates keeping only the first occurrence
outsample_df_cleaned = outsample_df.drop_duplicates(subset=['companyname', 'date', 'head

# Save the cleaned DataFrame back into outsample_df
outsample_df = outsample_df_cleaned

# Verify the cleaning process
no_headline_cleaned = outsample_df[outsample_df['headline'] == '[No_Headline]']
multiple_no_headline_cleaned = no_headline_cleaned.groupby(['companyname', 'date']).size

# Filter to keep only groups with more than one occurrence after cleaning
multiple_no_headline_cleaned = multiple_no_headline_cleaned[multiple_no_headline_cleaned

# Check if there are any duplicates left
if not multiple_no_headline_cleaned.empty:
    print("There are still rows with more than 1 no headline on the same date for each c
    print(multiple_no_headline_cleaned.head(5))
else:
    print("No more duplicate 'No_Headline' rows found for the same date and company afte

# Display cleaned DataFrame
print("Cleaned OUTSAMPLE DataFrame:")
print(outsample_df.head(5))
```

```
# Filter the DataFrame to show rows where 'headline' is '[No_Headline]'
no_headline_rows = outsample_df[outsample_df['headline'] == '[No_Headline]']

# Group by 'companyname' and 'announcedate' and count occurrences
no_headline_counts = no_headline_rows.groupby(['companyname', 'date']).size()

# Filter to keep only groups with more than one occurrence
multiple_no_headline = no_headline_counts[no_headline_counts > 1].reset_index()

# Merge to get detailed rows for these groups
detailed_no_headline_rows = no_headline_rows.merge(multiple_no_headline, on=['companynam

# Display these rows
print("OUTSAMPLE Rows with more than 1 no headline on the same date for each company:")
detailed_no_headline_rows.head(5)



# Remove the column '0' from insample_df if it exists
if '0' in insample_df.columns:
    insample_df = insample_df.drop(columns=['0'])

# Remove the column '0' from outsample_df if it exists
if '0' in outsample_df.columns:
    outsample_df = outsample_df.drop(columns=['0'])



# Count the unique company names in insample_df
unique_companynames_insample = insample_df['companyname'].nunique()

# Count the unique company names in outsample_df
unique_companynames_outsample = outsample_df['companyname'].nunique()

# Print the results
print(f"Unique company names in insample_df: {unique_companynames_insample}")
print(f"Unique company names in outsample_df: {unique_companynames_outsample}")



# Get unique company names from each DataFrame
unique_companynames_insample = insample_df['companyname'].unique()
unique_companynames_outsample = outsample_df['companyname'].unique()

# Combine the unique company names and get the unique values across both DataFrames
all_unique_companynames = pd.unique(pd.concat([pd.Series(unique_companynames_insample),

# Sort the unique company names
sorted_unique_companynames = sorted(all_unique_companynames)

# Print the sorted unique company names
print("Sorted unique company names from both DataFrames:")
for company in sorted_unique_companynames:
    print(company)
```

```
# List of companies to remove
companies_to_remove = ['Gray Television, Inc.', 'Scholastic Corporation', 'RADCOM Ltd.',

# Check if 'companyname' column exists in both DataFrames
if 'companyname' in insample_df.columns and 'companyname' in outsample_df.columns:
    # Filter insample_df
    insample_df = insample_df[~insample_df['companyname'].isin(companies_to_remove)]

    # Filter outsample_df
    outsample_df = outsample_df[~outsample_df['companyname'].isin(companies_to_remove)]



# Count the unique company names in insample_df
unique_companynames_insample = insample_df['companyname'].nunique()

# Count the unique company names in outsample_df
unique_companynames_outsample = outsample_df['companyname'].nunique()

# Print the results
print(f"Unique company names in insample_df: {unique_companynames_insample}")
print(f"Unique company names in outsample_df: {unique_companynames_outsample}")



# Get unique company names from each DataFrame
unique_companynames_insample = insample_df['companyname'].unique()
unique_companynames_outsample = outsample_df['companyname'].unique()

# Combine the unique company names and get the unique values across both DataFrames
all_unique_companynames = pd.unique(pd.concat([pd.Series(unique_companynames_insample),

# Sort the unique company names
sorted_unique_companynames = sorted(all_unique_companynames)

# Print the sorted unique company names
print("Sorted unique company names from both DataFrames:")
for company in sorted_unique_companynames:
    print(company)



# Save the Insample DataFrame to a CSV file
insample_df.to_csv('insample_df.csv', index=False)

# Save the Outsample DataFrame to a CSV file
outsample_df.to_csv('outsample_df.csv', index=False)
```

Tokenize and Word Cloud

```
!pip install nltk wordcloud matplotlib
```

```
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import string
from wordcloud import WordCloud
from collections import Counter

# Download stopwords from NLTK
import nltk
nltk.download('punkt')
nltk.download('stopwords')
```

```
# Reload the insample and outsample datasets
insample_df = pd.read_csv('insample_df.csv')
outsample_df = pd.read_csv('outsample_df.csv')
```

Visualization

```
insample_df.head(10)
```

```
# Function to remove duplicate Date From and Date To for each company
def remove_duplicates(df):
    return df.drop_duplicates(subset=['companyname', 'Date From', 'Date To'])

# Remove duplicates
insample_df = remove_duplicates(insample_df)

def plot_weekly_return(df, title):
    # Convert 'Date From' and 'Date To' to datetime format
    df['Date From'] = pd.to_datetime(df['Date From'])
    df['Date To'] = pd.to_datetime(df['Date To'])

    unique_companies = sorted(df['companyname'].unique())  # Sort companies alphabetical

    # Calculate ideal plot dimensions based on number of companies
    num_cols = 5
    num_rows = (len(unique_companies) + num_cols - 1) // num_cols

    fig, axes = plt.subplots(num_rows, num_cols, figsize=(25, 25), sharey=True)  # Adjus
    fig.suptitle(title, fontsize=20, fontweight='bold', color='navy')

    for i, company in enumerate(unique_companies):
        row, col = divmod(i, num_cols)
        ax = axes[row, col]

        company_data = df[df['companyname'] == company].sort_values(by='Date From')

        # Use line plot for better visualization of trends
        ax.plot(company_data['Date From'], company_data['Weekly Compound Return'],
                linewidth=1.5, alpha=0.7, color='blue')

        ax.set_title(company, fontsize=12, fontweight='bold')  # Smaller title
        ax.set_xlabel('Date', fontsize=10, fontweight='bold')    # Smaller label
        ax.set_ylabel('Weekly Return', fontsize=10, fontweight='bold')
        ax.grid(True, linestyle='--', linewidth=0.5)

        # Set vertical scale from -0.5 to 0.75
        ax.set_ylim(-0.5, 0.75)

        # Dynamically adjust x-axis ticks based on date range
        date_range = company_data['Date From'].max() - company_data['Date From'].min()
        if date_range > pd.Timedelta(days=365*2):
            ax.xaxis.set_major_locator(mdates.YearLocator())
            ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
        elif date_range > pd.Timedelta(days=365):
            ax.xaxis.set_major_locator(mdates.MonthLocator(bymonth=[1, 4, 7, 10]))
            ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
        else:
            ax.xaxis.set_major_locator(mdates.MonthLocator())
            ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))

        ax.tick_params(axis='x', rotation=45, labelsize=10)
        ax.tick_params(axis='y', labelsize=10)

    # Remove any unused subplots
```

```
    for i in range(len(unique_companies), num_rows * num_cols):
        fig.delaxes(axes.flatten()[i])

    plt.tight_layout(rect=[0, 0, 1, 0.95])
    plt.show()

plot_weekly_return(insample_df, 'In-sample Weekly Compound Return')
```

```
# Function to remove duplicate Date From and Date To for each company
def remove_duplicates(df):
    return df.drop_duplicates(subset=['companyname', 'Date From', 'Date To'])

# Remove duplicates
outsample_df = remove_duplicates(outsample_df)

def plot_weekly_return(df, title):
    # Convert 'Date From' and 'Date To' to datetime format
    df['Date From'] = pd.to_datetime(df['Date From'])
    df['Date To'] = pd.to_datetime(df['Date To'])

    unique_companies = sorted(df['companyname'].unique())  # Sort companies alphabetical

    # Calculate ideal plot dimensions based on number of companies
    num_cols = 5
    num_rows = (len(unique_companies) + num_cols - 1) // num_cols

    fig, axes = plt.subplots(num_rows, num_cols, figsize=(25, 25), sharey=True)  # Adjus
```

```
import matplotlib.dates as mdates
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Function to remove duplicate Date From and Date To for each company
def remove_duplicates(df):
    return df.drop_duplicates(subset=['companyname', 'Date From', 'Date To'])

# Remove duplicates from insample_df and outsample_df
insample_df = remove_duplicates(insample_df)
outsample_df = remove_duplicates(outsample_df)
```

Load Library

```
import pandas as pd
import torch
import numpy as np
from transformers import BertTokenizer, BertModel,RobertaTokenizer, RobertaModel,DistilBertModel, DistilBertTokenizer, DistilBertForSequ
```

```
# Load the data
insample_df = pd.read_csv('insample_df.csv')
outsample_df = pd.read_csv('outsample_df.csv')
```

```
insample_df.head(2)
```

Preprocess

```
#Step 2: Pre-process Text Data

# Encode the 'Up' and 'Down' labels
label_mapping = {'Up': 1, 'Down': 0}

# Filter out 'No_Change' labels and encode target variable
insample_df = insample_df[insample_df['Future Return Direction'].isin(['Up', 'Down'])]
outsample_df = outsample_df[outsample_df['Future Return Direction'].isin(['Up', 'Down'])]

insample_df['Future Return Direction'] = insample_df['Future Return Direction'].map(label_mapping)
outsample_df['Future Return Direction'] = outsample_df['Future Return Direction'].map(label_mapping)

# Drop rows with NaN values in the target variable
insample_df = insample_df.dropna(subset=['Future Return Direction'])
outsample_df = outsample_df.dropna(subset=['Future Return Direction'])
```

BERT

```
# Load the pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Function to generate embeddings in batches
def get_average_embedding_batch(text_list, batch_size=16):
    embeddings = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs)
        batch_embeddings = outputs.last_hidden_state.mean(dim=1).cpu().numpy()
        embeddings.extend(batch_embeddings)
    return embeddings

# Apply the function to generate embeddings for headlines
insample_df['embedding'] = get_average_embedding_batch(insample_df['headline'].tolist())
outsample_df['embedding'] = get_average_embedding_batch(outsample_df['headline'].tolist())
```

```
# Save the dataframes to a pickle file
with open('embedding-BERT-AllCompany-NEW.pkl', 'wb') as f:
    pd.to_pickle((insample_df, outsample_df), f)

print("Dataframes with embeddings have been saved to 'embedding-BERT-AllCompany-NEW.pkl'")
```

RoBERTa

```
# Load the pre-trained RoBERTa model and tokenizer
tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
model = RobertaModel.from_pretrained('roberta-base')

# Function to generate embeddings in batches
def get_average_embedding_batch(text_list, batch_size=16):
    embeddings = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs)
        batch_embeddings = outputs.last_hidden_state.mean(dim=1).cpu().numpy()
        embeddings.extend(batch_embeddings)
    return embeddings

# Apply the function to generate embeddings for headlines
insample_df['embedding'] = get_average_embedding_batch(insample_df['headline'].tolist())
outsample_df['embedding'] = get_average_embedding_batch(outsample_df['headline'].tolist())


# Save the dataframes to a pickle file
with open('embedding-RoBERTa-AllCompany-NEW.pkl', 'wb') as f:
    pd.to_pickle((insample_df, outsample_df), f)

print("Dataframes with embeddings have been saved to 'embedding-RoBERTa-AllCompany-NEW.pkl'")
```

DistilBERT

```
# Load the pre-trained DistilBERT model and tokenizer
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
model = DistilBertModel.from_pretrained('distilbert-base-uncased')

# Function to generate embeddings in batches
def get_average_embedding_batch(text_list, batch_size=16):
    embeddings = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs)
        batch_embeddings = outputs.last_hidden_state.mean(dim=1).cpu().numpy()
        embeddings.extend(batch_embeddings)
    return embeddings

# Apply the function to generate embeddings for headlines
insample_df['embedding'] = get_average_embedding_batch(insample_df['headline'].tolist())
outsample_df['embedding'] = get_average_embedding_batch(outsample_df['headline'].tolist())


# Save the dataframes to a pickle file
with open('embedding-DistilBERT-AllCompany-NEW.pkl', 'wb') as f:
    pd.to_pickle((insample_df, outsample_df), f)

print("Dataframes with embeddings have been saved to 'embedding-DistilBERT-AllCompany-NEW.pkl'")
```

DistilRoBERTa

```
# Load the pre-trained DistilRoBERTa model and tokenizer

# Save the dataframes to a pickle file
with open('embedding-DistilRoBERTa-AllCompany-NEW.pkl', 'wb') as f:
    pd.to_pickle((insample_df, outsample_df), f)

print("Dataframes with embeddings have been saved to 'embedding-DistilRoBERTa-AllCompany-NEW.pkl'")
    for i in range(0, len(text_list), batch_size):
```

FinBERT

```
            outputs = model(**inputs)

# Load the pre-trained FinBERT model and tokenizer
tokenizer = AutoTokenizer.from_pretrained('yiyanghkust/finbert-tone', use_fast=False)
model = AutoModel.from_pretrained('yiyanghkust/finbert-tone')

# Function to generate embeddings in batches
def get_average_embedding_batch(text_list, batch_size=16):
    embeddings = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs, return_dict=True)
        # Extract the output embeddings (CLS token representation from the last hidden state)
        last_hidden_states = outputs.last_hidden_state
        # Average pooling of the last hidden state across tokens
        avg_pooled_embeddings = torch.mean(last_hidden_states, dim=1).cpu().numpy()
        embeddings.extend(avg_pooled_embeddings)
    return embeddings

# Assuming insample_df and outsample_df are defined earlier with 'headline' column

# Apply the function to generate embeddings for headlines
insample_df['embedding'] = get_average_embedding_batch(insample_df['headline'].tolist())
outsample_df['embedding'] = get_average_embedding_batch(outsample_df['headline'].tolist())


# Save the dataframes to a pickle file
with open('embedding-FinBERT-AllCompany-NEW.pkl', 'wb') as f:
    pd.to_pickle((insample_df, outsample_df), f)

print("Dataframes with embeddings have been saved to 'embedding-FinBERT-AllCompany-NEW.pkl'")
```

Import Library

```
import pandas as pd
from transformers import BertTokenizer, BertModel,RobertaTokenizer, RobertaModel, Distil
import torch
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, con
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import timedelta
from sklearn.model_selection import TimeSeriesSplit
import matplotlib.dates as mdates
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
import plotly.graph_objs as go
import plotly.express as px
```

BERT

Load Embedding

```
# Load the pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Load embedded Dataframe
with open('embedding-BERT-AllCompany-NEW.pkl', 'rb') as f:
    insample_df, outsample_df = pd.read_pickle(f)

# Display the first 2 rows of outsample_df to check
filtered_df = outsample_df[outsample_df['headline'] != '[No_Headline]']
filtered_df.head(3)
```

Accuracy per-companies

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Define rolling window parameters
window_size = 365 * 10  # 10 years in days
prediction_period = 365  # 1 year in days

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)
```

```
    # Ensure the length of predictions and test_df matches
    if len(y_pred) != len(test_df):
        raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

    # Store the predictions and true values
    test_df['predicted'] = y_pred

    # Update the company-specific DataFrame with predictions
    company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    print(f'Company: {company}')
    print(f'Training window: {train_df["Date From"].min()} to {train_df["Date From"]
    print(f'Test window: {test_df["Date From"].min()} to {test_df["Date From"].max()
    print(f'Predicted years: {test_df["Date From"].dt.year.unique()}')

# Evaluate the Model for the company
for year in range(2016, 2024):
    year_df = company_df[company_df['Date From'].dt.year == year]

    if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
        print(f"Missing predictions for year {year} for company {company} due to ins
        continue

    y_test = year_df['Future Return Direction'].values
    y_pred = year_df['predicted'].values

    # Remove NaN values in predictions
    valid_indices = ~np.isnan(y_pred)
    y_test = y_test[valid_indices]
    y_pred = y_pred[valid_indices]

    if len(y_pred) == 0 or len(y_test) == 0:
        print(f"Insufficient valid predictions for year {year} for company {company}
        continue

    # Calculate evaluation metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary', pos_label=1)
    recall = recall_score(y_test, y_pred, average='binary', pos_label=1)
    f1 = f1_score(y_test, y_pred, average='binary', pos_label=1)

    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)
    years.append(year)

    print(f'Company: {company}, Year: {year}')
    print(f'Accuracy: {accuracy:.2f}')
    print(f'Precision: {precision:.2f}')
    print(f'Recall: {recall:.2f}')
    print(f'F1 Score: {f1:.2f}')
    print()

# Store the results for the company in a DataFrame
```

```
results_df = pd.DataFrame({
    'Year': years,
    'Accuracy': accuracies,
    'Precision': precisions,
    'Recall': recalls,
    'F1 Score': f1_scores
})

# Save the DataFrame in the dictionary
company_results_dfs[company] = results_df

print(f"Evaluation results for {company}:")
print(results_df)

# Plot accuracy over time for the company
plt.figure(figsize=(10, 6))
plt.plot(results_df['Year'], results_df['Accuracy'], marker='o', label='Accuracy')
plt.plot(results_df['Year'], results_df['Precision'], marker='o', label='Precision')
plt.plot(results_df['Year'], results_df['Recall'], marker='o', label='Recall')
plt.plot(results_df['Year'], results_df['F1 Score'], marker='o', label='F1 Score')

plt.title(f'Performance Metrics Over Time for {company}')
plt.xlabel('Year')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()

# Plot confusion matrices for each year
n_plots = len(years)
n_cols = 4
n_rows = (n_plots // n_cols) + (n_plots % n_cols > 0)

fig, axes = plt.subplots(n_rows, n_cols, figsize=(20, 4 * n_rows))
axes = axes.flatten()

for i, year in enumerate(years):
    year_df = company_df[company_df['Date From'].dt.year == year]
    y_test = year_df['Future Return Direction'].values
    y_pred = year_df['predicted'].values

    # Remove NaN values in predictions
    valid_indices = ~np.isnan(y_pred)
    y_test = y_test[valid_indices]
    y_pred = y_pred[valid_indices]

    if len(y_pred) > 0 and len(y_test) > 0:
        cm = confusion_matrix(y_test, y_pred)
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=logistic_m
        disp.plot(cmap='Blues', ax=axes[i])
        axes[i].set_title(f'{company} {year}')

for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])
```

```
    plt.tight_layout()
    plt.show()

# Access the stored DataFrames
for company, results_df in company_results_dfs.items():
    print(f"\nResults for {company}:")
    print(results_df)



# Concatenate all the company results into one DataFrame
all_results_df = pd.concat(company_results_dfs.values(), keys=company_results_dfs.keys()

# Save the combined DataFrame to a CSV file
all_results_df.to_csv('[EVAL] BERT_all_company_results.csv', index=False)
```

Accuracy all years

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Define rolling window parameters
window_size = 365 * 10  # 10 years in days
prediction_period = 365  # 1 year in days

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)
```

```
        # Ensure the length of predictions and test_df matches
        if len(y_pred) != len(test_df):
            raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

        # Store the predictions and true values
        test_df['predicted'] = y_pred

        # Update the company-specific DataFrame with predictions
        company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    # Evaluate the Model for the company
    for year in range(2016, 2024):
        year_df = company_df[company_df['Date From'].dt.year == year]

        if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
            continue

        y_test = year_df['Future Return Direction'].values
        y_pred = year_df['predicted'].values

        # Remove NaN values in predictions
        valid_indices = ~np.isnan(y_pred)
        y_test = y_test[valid_indices]
        y_pred = y_pred[valid_indices]

        if len(y_pred) == 0 or len(y_test) == 0:
            continue

        # Calculate evaluation metrics
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='binary', pos_label=1)
        recall = recall_score(y_test, y_pred, average='binary', pos_label=1)
        f1 = f1_score(y_test, y_pred, average='binary', pos_label=1)

        accuracies.append(accuracy)
        precisions.append(precision)
        recalls.append(recall)
        f1_scores.append(f1)
        years.append(year)

    # Store the results for the company in a DataFrame
    results_df = pd.DataFrame({
        'Year': years,
        'Accuracy': accuracies,
        'Precision': precisions,
        'Recall': recalls,
        'F1 Score': f1_scores
    })

    # Save the DataFrame in the dictionary
    company_results_dfs[company] = results_df

# Combine results of all companies into a single DataFrame
combined_results = pd.concat(company_results_dfs.values(), ignore_index=True)
```

```
# Calculate average accuracy per year across all companies
average_metrics_per_year = combined_results.groupby('Year').mean().reset_index()

# Print average metrics for each year
print("Average Metrics Per Year:")
print(average_metrics_per_year)

# Plot average accuracy per year
plt.figure(figsize=(10, 6))
sns.set(style='whitegrid')

plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Accuracy'], marker=
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Precision'], marker=
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Recall'], marker='o
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['F1 Score'], marker=

plt.title('Average Performance Metrics Over Time Across All Companies')
plt.xlabel('Year')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()
```

Standard Deviation all years

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)

        # Ensure the length of predictions and test_df matches
        if len(y_pred) != len(test_df):
            raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

        # Store the predictions and true values
        test_df['predicted'] = y_pred
```

```
        # Update the company-specific DataFrame with predictions
        company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    # Evaluate the Model for the company
    for year in range(2016, 2024):
        year_df = company_df[company_df['Date From'].dt.year == year]

        if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
            continue

        y_test = year_df['Future Return Direction'].values
        y_pred = year_df['predicted'].values

        # Remove NaN values in predictions
        valid_indices = ~np.isnan(y_pred)
        y_test = y_test[valid_indices]
        y_pred = y_pred[valid_indices]

        if len(y_pred) == 0 or len(y_test) == 0:
            continue

        # Calculate evaluation metrics
        accuracy = accuracy_score(y_test, y_pred)

        accuracies.append(accuracy)
        years.append(year)

    # Store the results for the company in a DataFrame
    results_df = pd.DataFrame({
        'Year': years,
        'Accuracy': accuracies
    })

    # Save the DataFrame in the dictionary
    company_results_dfs[company] = results_df

# Combine results of all companies into a single DataFrame
combined_results = pd.concat(company_results_dfs.values(), ignore_index=True)

# Calculate average accuracy per year across all companies
average_metrics_per_year = combined_results.groupby('Year').agg(
    Accuracy_mean=('Accuracy', 'mean'),
    Accuracy_std=('Accuracy', 'std')
).reset_index()

# Print average metrics for each year
print("Average Metrics Per Year:")
print(average_metrics_per_year)

# Get a color sequence from Plotly's default colors
colors = px.colors.qualitative.Plotly

# Function to make the color more transparent
def get_transparent_color(color, alpha=0.2):
    # Convert hex to RGB and then to RGBA
```

```
    hex_color = color.lstrip('#')
    rgb_color = tuple(int(hex_color[i:i+2], 16) for i in (0, 2, 4))
    return f'rgba({rgb_color[0]}, {rgb_color[1]}, {rgb_color[2]}, {alpha})'

# Create a figure
fig = go.Figure()

# Add the mean line
fig.add_trace(go.Scatter(
    x=average_metrics_per_year['Year'],
    y=average_metrics_per_year['Accuracy_mean'],
    mode='lines',
    name='Accuracy',
    line=dict(color=colors[0], width=2)
))

# Add the standard deviation shaded area
fig.add_trace(go.Scatter(
    x=pd.concat([average_metrics_per_year['Year'], average_metrics_per_year['Year'][::-1
    y=pd.concat([average_metrics_per_year['Accuracy_mean'] + average_metrics_per_year['A
                 (average_metrics_per_year['Accuracy_mean'] - average_metrics_per_year['
    fill='toself',
    fillcolor=get_transparent_color(colors[0], alpha=0.2),  # Use the same color with tra
    line=dict(color='rgba(255,255,255,0)'),
    hoverinfo="skip",
    showlegend=False,
    name='Accuracy std dev'
))

# Customize layout
fig.update_layout(
    title='Average Rolling Window Accuracy Over Time Across All Companies',
    xaxis_title='Year',
    yaxis_title='Accuracy',
    template='plotly_white',
    showlegend=True
)

# Show the plot
fig.show()
```

Portofolio

```
def prepare_data(insample_df, outsample_df):
    df = pd.concat([insample_df, outsample_df])
    df['Date From'] = pd.to_datetime(df['Date From'])
    df = df.sort_values(by='Date From')
    available_years = df['Date From'].dt.year.unique()
    print("Years available in the data:", available_years)
    return df

def rolling_window_analysis(df):
    companies = df['companyname'].unique()
    predictions_df = pd.DataFrame()

    for company in companies:
        company_df = df[df['companyname'] == company].copy()
        company_df = company_df.sort_values(by='Date From')

        # Define the rolling window parameters
        start_year = company_df['Date From'].dt.year.min()
        end_year = company_df['Date From'].dt.year.max()
        window_size = 10
        validation_size = 1

        for start in range(start_year, end_year - window_size - validation_size + 1):
            train_start = start
            train_end = start + window_size
            val_start = train_end
            val_end = train_end + validation_size

            train_df = company_df[(company_df['Date From'].dt.year >= train_start) &
                                  (company_df['Date From'].dt.year < train_end)]
            test_df = company_df[(company_df['Date From'].dt.year >= val_start) &
                                 (company_df['Date From'].dt.year < val_end)]

            if len(test_df) == 0 or len(train_df) == 0:
                continue

            X_train = np.vstack(train_df['embedding'].values)
            y_train = train_df['Future Return Direction'].values
            X_test = np.vstack(test_df['embedding'].values)
            y_test = test_df['Future Return Direction'].values

            # Train logistic regression model
            logistic_model = LogisticRegression(max_iter=1000)
            logistic_model.fit(X_train, y_train)

            # Get prediction probabilities
            y_prob = logistic_model.predict_proba(X_test)[:, 1]  # Probability of the pos

            if len(y_prob) != len(test_df):
                raise ValueError(f"Mismatch between test data length ({len(test_df)}) and

            test_df['predicted_prob'] = y_prob
            predictions_df = pd.concat([predictions_df, test_df[['Date From', 'companynam

    df = df.merge(predictions_df, on=['Date From', 'companyname', 'Weekly Compound Return
```

```
    return df

def construct_portfolio(df, time_period='Week'):
    df['Date From'] = pd.to_datetime(df['Date From'])
    if time_period == 'Week':
        df['Period'] = df['Date From'].dt.to_period('W').dt.to_timestamp()
    else:
        raise ValueError("Invalid time_period. Use 'Week'.")

    portfolio_returns = []

    for period, group in df.groupby('Period'):
        if period.year < 2016:
            continue

        # Sort group by predicted_prob descending
        group_sorted = group.sort_values(by='predicted_prob', ascending=False)

        # Select top and bottom companies
        num_top_companies = 5
        num_bottom_companies = 5
        top_companies = group_sorted.head(num_top_companies)
        bottom_companies = group_sorted.tail(num_bottom_companies)

        # Equal-weighted returns
        long_return_eq = np.mean(np.log1p(top_companies['Weekly Compound Return']))
        short_return_eq = np.mean(np.log1p(bottom_companies['Weekly Compound Return']))
        long_short_return_eq = long_return_eq - short_return_eq

        # Value-weighted returns
        long_return_val = np.sum(np.log1p(top_companies['Weekly Compound Return']) * top_
        short_return_val = np.sum(np.log1p(bottom_companies['Weekly Compound Return']) *
        long_short_return_val = long_return_val - short_return_val

        portfolio_returns.append({
            'Period': period,
            'Long Return (Eq)': long_return_eq,
            'Short Return (Eq)': short_return_eq,
            'Long-Short Return (Eq)': long_short_return_eq,
            'Long Return (Val)': long_return_val,
            'Short Return (Val)': short_return_val,
            'Long-Short Return (Val)': long_short_return_val
        })

    portfolio_df = pd.DataFrame(portfolio_returns)
    portfolio_df['EW L'] = portfolio_df['Long Return (Eq)'].cumsum()
    portfolio_df['EW S'] = portfolio_df['Short Return (Eq)'].cumsum()
    portfolio_df['EW LS'] = portfolio_df['Long-Short Return (Eq)'].cumsum()
    portfolio_df['VW L'] = portfolio_df['Long Return (Val)'].cumsum()
    portfolio_df['VW S'] = portfolio_df['Short Return (Val)'].cumsum()
    portfolio_df['VW LS'] = portfolio_df['Long-Short Return (Val)'].cumsum()

    actual_returns = df[df['Date From'].dt.year >= 2016].groupby('Period')['Weekly Compou
    actual_cumulative_returns = np.log1p(actual_returns).cumsum()
    portfolio_df = portfolio_df.merge(actual_cumulative_returns.rename('Market'), on='Pe
```

```
    metrics = {}
    for portfolio in ['EW L', 'EW S', 'EW LS', 'VW L', 'VW S', 'VW LS']:
        returns = portfolio_df[portfolio]

        if returns.isnull().all() or returns.eq(0).all():
            sharpe_ratio = np.nan
            max_drawdown = np.nan
            volatility = np.nan
        else:
            sharpe_ratio = returns.mean() / returns.std() * np.sqrt(52) if returns.std()
            cumulative_returns = returns.cumsum()
            max_drawdown = (cumulative_returns.cummax() - cumulative_returns).max()
            volatility = returns.std() * np.sqrt(52)

        metrics[portfolio] = {
            'Sharpe Ratio': sharpe_ratio
        }

        print(f"Metrics for {portfolio}:")
        print(f"Sharpe Ratio: {sharpe_ratio}")
        print()

    portfolio_df.to_csv('BERT_portfolio_returns.csv', index=False)
    print("Portfolio returns saved to 'BERT_portfolio_returns.csv'")
    return portfolio_df

def plot_portfolio_returns(portfolio_df, title_suffix=''):
    plt.figure(figsize=(12, 6))

    plt.plot(portfolio_df['Period'], portfolio_df['EW L'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['EW S'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['EW LS'], marker='o', markersize=1, lab
    plt.plot(portfolio_df['Period'], portfolio_df['VW L'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['VW S'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['VW LS'], marker='o', markersize=1, lab
    plt.plot(portfolio_df['Period'], portfolio_df['Market'], marker='o', markersize=1, la

    plt.title(f'Cumulative {title_suffix} Portfolio Returns Over Time')
    plt.xlabel('Period')
    plt.ylabel('Cumulative Log Return')
    plt.legend()
    plt.grid(True)

    plt.gca().xaxis.set_major_locator(mdates.YearLocator())
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    plt.xticks(rotation=45)
    plt.show()

# Example usage for Weekly
df = prepare_data(insample_df, outsample_df)
df = rolling_window_analysis(df)

# Weekly Portfolio
portfolio_df_week = construct_portfolio(df, time_period='Week')
```

```
portfolio_df_week = portfolio_df_week[portfolio_df_week['Period'].dt.year >= 2016]
plot_portfolio_returns(portfolio_df_week, title_suffix='Weekly')
```

RoBERTa

Load Embedding

```
# Load the pre-trained RoBERTa model and tokenizer
tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
model = RobertaModel.from_pretrained('roberta-base')

# Load embedded Dataframe
with open('embedding-RoBERTa-AllCompany-NEW.pkl', 'rb') as f:
    insample_df, outsample_df = pd.read_pickle(f)

# Display the first 2 rows of outsample_df to check
filtered_df = outsample_df[outsample_df['headline'] != '[No_Headline]']
filtered_df.head(3)
```

Accuracy per-companies

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Define rolling window parameters
window_size = 365 * 10  # 10 years in days
prediction_period = 365  # 1 year in days

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)
```

```
    # Ensure the length of predictions and test_df matches
    if len(y_pred) != len(test_df):
        raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

    # Store the predictions and true values
    test_df['predicted'] = y_pred

    # Update the company-specific DataFrame with predictions
    company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    print(f'Company: {company}')
    print(f'Training window: {train_df["Date From"].min()} to {train_df["Date From"]
    print(f'Test window: {test_df["Date From"].min()} to {test_df["Date From"].max()
    print(f'Predicted years: {test_df["Date From"].dt.year.unique()}')

# Evaluate the Model for the company
for year in range(2016, 2024):
    year_df = company_df[company_df['Date From'].dt.year == year]

    if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
        print(f"Missing predictions for year {year} for company {company} due to ins
        continue

    y_test = year_df['Future Return Direction'].values
    y_pred = year_df['predicted'].values

    # Remove NaN values in predictions
    valid_indices = ~np.isnan(y_pred)
    y_test = y_test[valid_indices]
    y_pred = y_pred[valid_indices]

    if len(y_pred) == 0 or len(y_test) == 0:
        print(f"Insufficient valid predictions for year {year} for company {company}
        continue

    # Calculate evaluation metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary', pos_label=1)
    recall = recall_score(y_test, y_pred, average='binary', pos_label=1)
    f1 = f1_score(y_test, y_pred, average='binary', pos_label=1)

    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)
    years.append(year)

    print(f'Company: {company}, Year: {year}')
    print(f'Accuracy: {accuracy:.2f}')
    print(f'Precision: {precision:.2f}')
    print(f'Recall: {recall:.2f}')
    print(f'F1 Score: {f1:.2f}')
    print()

# Store the results for the company in a DataFrame
```

```
results_df = pd.DataFrame({
    'Year': years,
    'Accuracy': accuracies,
    'Precision': precisions,
    'Recall': recalls,
    'F1 Score': f1_scores
})

# Save the DataFrame in the dictionary
company_results_dfs[company] = results_df

print(f"Evaluation results for {company}:")
print(results_df)

# Plot accuracy over time for the company
plt.figure(figsize=(10, 6))
plt.plot(results_df['Year'], results_df['Accuracy'], marker='o', label='Accuracy')
plt.plot(results_df['Year'], results_df['Precision'], marker='o', label='Precision')
plt.plot(results_df['Year'], results_df['Recall'], marker='o', label='Recall')
plt.plot(results_df['Year'], results_df['F1 Score'], marker='o', label='F1 Score')

plt.title(f'Performance Metrics Over Time for {company}')
plt.xlabel('Year')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()

# Plot confusion matrices for each year
n_plots = len(years)
n_cols = 4
n_rows = (n_plots // n_cols) + (n_plots % n_cols > 0)

fig, axes = plt.subplots(n_rows, n_cols, figsize=(20, 4 * n_rows))
axes = axes.flatten()

for i, year in enumerate(years):
    year_df = company_df[company_df['Date From'].dt.year == year]
    y_test = year_df['Future Return Direction'].values
    y_pred = year_df['predicted'].values

    # Remove NaN values in predictions
    valid_indices = ~np.isnan(y_pred)
    y_test = y_test[valid_indices]
    y_pred = y_pred[valid_indices]

    if len(y_pred) > 0 and len(y_test) > 0:
        cm = confusion_matrix(y_test, y_pred)
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=logistic_m
        disp.plot(cmap='Blues', ax=axes[i])
        axes[i].set_title(f'{company} {year}')

for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])
```

```
    plt.tight_layout()
    plt.show()

# Access the stored DataFrames
for company, results_df in company_results_dfs.items():
    print(f"\nResults for {company}:")
    print(results_df)



# Concatenate all the company results into one DataFrame
all_results_df = pd.concat(company_results_dfs.values(), keys=company_results_dfs.keys()

# Save the combined DataFrame to a CSV file
all_results_df.to_csv('[EVAL] RoBERTa_all_company_results.csv', index=False)
```

Accuracy all years

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Define rolling window parameters
window_size = 365 * 10  # 10 years in days
prediction_period = 365  # 1 year in days

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)
```

```
        # Ensure the length of predictions and test_df matches
        if len(y_pred) != len(test_df):
            raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

        # Store the predictions and true values
        test_df['predicted'] = y_pred

        # Update the company-specific DataFrame with predictions
        company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    # Evaluate the Model for the company
    for year in range(2016, 2024):
        year_df = company_df[company_df['Date From'].dt.year == year]

        if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
            continue

        y_test = year_df['Future Return Direction'].values
        y_pred = year_df['predicted'].values

        # Remove NaN values in predictions
        valid_indices = ~np.isnan(y_pred)
        y_test = y_test[valid_indices]
        y_pred = y_pred[valid_indices]

        if len(y_pred) == 0 or len(y_test) == 0:
            continue

        # Calculate evaluation metrics
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='binary', pos_label=1)
        recall = recall_score(y_test, y_pred, average='binary', pos_label=1)
        f1 = f1_score(y_test, y_pred, average='binary', pos_label=1)

        accuracies.append(accuracy)
        precisions.append(precision)
        recalls.append(recall)
        f1_scores.append(f1)
        years.append(year)

    # Store the results for the company in a DataFrame
    results_df = pd.DataFrame({
        'Year': years,
        'Accuracy': accuracies,
        'Precision': precisions,
        'Recall': recalls,
        'F1 Score': f1_scores
    })

    # Save the DataFrame in the dictionary
    company_results_dfs[company] = results_df

# Combine results of all companies into a single DataFrame
combined_results = pd.concat(company_results_dfs.values(), ignore_index=True)
```

```
# Calculate average accuracy per year across all companies
average_metrics_per_year = combined_results.groupby('Year').mean().reset_index()

# Print average metrics for each year
print("Average Metrics Per Year:")
print(average_metrics_per_year)

# Plot average accuracy per year
plt.figure(figsize=(10, 6))
sns.set(style='whitegrid')

plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Accuracy'], marker=
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Precision'], marker=
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Recall'], marker='o
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['F1 Score'], marker=

plt.title('Average Performance Metrics Over Time Across All Companies')
plt.xlabel('Year')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()
```

Standard Deviation all years

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)

        # Ensure the length of predictions and test_df matches
        if len(y_pred) != len(test_df):
            raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

        # Store the predictions and true values
        test_df['predicted'] = y_pred
```

```
        # Update the company-specific DataFrame with predictions
        company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    # Evaluate the Model for the company
    for year in range(2016, 2024):
        year_df = company_df[company_df['Date From'].dt.year == year]

        if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
            continue

        y_test = year_df['Future Return Direction'].values
        y_pred = year_df['predicted'].values

        # Remove NaN values in predictions
        valid_indices = ~np.isnan(y_pred)
        y_test = y_test[valid_indices]
        y_pred = y_pred[valid_indices]

        if len(y_pred) == 0 or len(y_test) == 0:
            continue

        # Calculate evaluation metrics
        accuracy = accuracy_score(y_test, y_pred)

        accuracies.append(accuracy)
        years.append(year)

    # Store the results for the company in a DataFrame
    results_df = pd.DataFrame({
        'Year': years,
        'Accuracy': accuracies
    })

    # Save the DataFrame in the dictionary
    company_results_dfs[company] = results_df

# Combine results of all companies into a single DataFrame
combined_results = pd.concat(company_results_dfs.values(), ignore_index=True)

# Calculate average accuracy per year across all companies
average_metrics_per_year = combined_results.groupby('Year').agg(
    Accuracy_mean=('Accuracy', 'mean'),
    Accuracy_std=('Accuracy', 'std')
).reset_index()

# Print average metrics for each year
print("Average Metrics Per Year:")
print(average_metrics_per_year)

# Get a color sequence from Plotly's default colors
colors = px.colors.qualitative.Plotly

# Function to make the color more transparent
def get_transparent_color(color, alpha=0.2):
    # Convert hex to RGB and then to RGBA
```

```
    hex_color = color.lstrip('#')
    rgb_color = tuple(int(hex_color[i:i+2], 16) for i in (0, 2, 4))
    return f'rgba({rgb_color[0]}, {rgb_color[1]}, {rgb_color[2]}, {alpha})'

# Create a figure
fig = go.Figure()

# Add the mean line
fig.add_trace(go.Scatter(
    x=average_metrics_per_year['Year'],
    y=average_metrics_per_year['Accuracy_mean'],
    mode='lines',
    name='Accuracy',
    line=dict(color=colors[0], width=2)
))

# Add the standard deviation shaded area
fig.add_trace(go.Scatter(
    x=pd.concat([average_metrics_per_year['Year'], average_metrics_per_year['Year'][::-1
    y=pd.concat([average_metrics_per_year['Accuracy_mean'] + average_metrics_per_year['A
                 (average_metrics_per_year['Accuracy_mean'] - average_metrics_per_year['
    fill='toself',
    fillcolor=get_transparent_color(colors[0], alpha=0.2),  # Use the same color with tra
    line=dict(color='rgba(255,255,255,0)'),
    hoverinfo="skip",
    showlegend=False,
    name='Accuracy std dev'
))

# Customize layout
fig.update_layout(
    title='Average Rolling Window Accuracy Over Time Across All Companies',
    xaxis_title='Year',
    yaxis_title='Accuracy',
    template='plotly_white',
    showlegend=True
)

# Show the plot
fig.show()
```

Portofolio Revision

```
def prepare_data(insample_df, outsample_df):
    df = pd.concat([insample_df, outsample_df])
    df['Date From'] = pd.to_datetime(df['Date From'])
    df = df.sort_values(by='Date From')
    available_years = df['Date From'].dt.year.unique()
    print("Years available in the data:", available_years)
    return df

def rolling_window_analysis(df):
    companies = df['companyname'].unique()
    predictions_df = pd.DataFrame()

    for company in companies:
        company_df = df[df['companyname'] == company].copy()
        company_df = company_df.sort_values(by='Date From')

        # Define the rolling window parameters
        start_year = company_df['Date From'].dt.year.min()
        end_year = company_df['Date From'].dt.year.max()
        window_size = 10
        validation_size = 1

        for start in range(start_year, end_year - window_size - validation_size + 1):
            train_start = start
            train_end = start + window_size
            val_start = train_end
            val_end = train_end + validation_size

            train_df = company_df[(company_df['Date From'].dt.year >= train_start) &
                                  (company_df['Date From'].dt.year < train_end)]
            test_df = company_df[(company_df['Date From'].dt.year >= val_start) &
                                 (company_df['Date From'].dt.year < val_end)]

            if len(test_df) == 0 or len(train_df) == 0:
                continue

            X_train = np.vstack(train_df['embedding'].values)
            y_train = train_df['Future Return Direction'].values
            X_test = np.vstack(test_df['embedding'].values)
            y_test = test_df['Future Return Direction'].values

            # Train logistic regression model
            logistic_model = LogisticRegression(max_iter=1000)
            logistic_model.fit(X_train, y_train)

            # Get prediction probabilities
            y_prob = logistic_model.predict_proba(X_test)[:, 1]  # Probability of the pos

            if len(y_prob) != len(test_df):
                raise ValueError(f"Mismatch between test data length ({len(test_df)}) and

            test_df['predicted_prob'] = y_prob
            predictions_df = pd.concat([predictions_df, test_df[['Date From', 'companynam

    df = df.merge(predictions_df, on=['Date From', 'companyname', 'Weekly Compound Return
```

```
    return df

def construct_portfolio(df, time_period='Week'):
    df['Date From'] = pd.to_datetime(df['Date From'])
    if time_period == 'Week':
        df['Period'] = df['Date From'].dt.to_period('W').dt.to_timestamp()
    else:
        raise ValueError("Invalid time_period. Use 'Week'.")

    portfolio_returns = []

    for period, group in df.groupby('Period'):
        if period.year < 2016:
            continue

        # Sort group by predicted_prob descending
        group_sorted = group.sort_values(by='predicted_prob', ascending=False)

        # Select top and bottom companies
        num_top_companies = 5
        num_bottom_companies = 5
        top_companies = group_sorted.head(num_top_companies)
        bottom_companies = group_sorted.tail(num_bottom_companies)

        # Equal-weighted returns
        long_return_eq = np.mean(np.log1p(top_companies['Weekly Compound Return']))
        short_return_eq = np.mean(np.log1p(bottom_companies['Weekly Compound Return']))
        long_short_return_eq = long_return_eq - short_return_eq

        # Value-weighted returns
        long_return_val = np.sum(np.log1p(top_companies['Weekly Compound Return']) * top_
        short_return_val = np.sum(np.log1p(bottom_companies['Weekly Compound Return']) *
        long_short_return_val = long_return_val - short_return_val

        portfolio_returns.append({
            'Period': period,
            'Long Return (Eq)': long_return_eq,
            'Short Return (Eq)': short_return_eq,
            'Long-Short Return (Eq)': long_short_return_eq,
            'Long Return (Val)': long_return_val,
            'Short Return (Val)': short_return_val,
            'Long-Short Return (Val)': long_short_return_val
        })

    portfolio_df = pd.DataFrame(portfolio_returns)
    portfolio_df['EW L'] = portfolio_df['Long Return (Eq)'].cumsum()
    portfolio_df['EW S'] = portfolio_df['Short Return (Eq)'].cumsum()
    portfolio_df['EW LS'] = portfolio_df['Long-Short Return (Eq)'].cumsum()
    portfolio_df['VW L'] = portfolio_df['Long Return (Val)'].cumsum()
    portfolio_df['VW S'] = portfolio_df['Short Return (Val)'].cumsum()
    portfolio_df['VW LS'] = portfolio_df['Long-Short Return (Val)'].cumsum()

    actual_returns = df[df['Date From'].dt.year >= 2016].groupby('Period')['Weekly Compou
    actual_cumulative_returns = np.log1p(actual_returns).cumsum()
    portfolio_df = portfolio_df.merge(actual_cumulative_returns.rename('Market'), on='Pe
```

```
    metrics = {}
    for portfolio in ['EW L', 'EW S', 'EW LS', 'VW L', 'VW S', 'VW LS']:
        returns = portfolio_df[portfolio]

        if returns.isnull().all() or returns.eq(0).all():
            sharpe_ratio = np.nan
            max_drawdown = np.nan
            volatility = np.nan
        else:
            sharpe_ratio = returns.mean() / returns.std() * np.sqrt(52) if returns.std()
            cumulative_returns = returns.cumsum()
            max_drawdown = (cumulative_returns.cummax() - cumulative_returns).max()
            volatility = returns.std() * np.sqrt(52)

        metrics[portfolio] = {
            'Sharpe Ratio': sharpe_ratio
        }

        print(f"Metrics for {portfolio}:")
        print(f"Sharpe Ratio: {sharpe_ratio}")
        print()

    portfolio_df.to_csv('RoBERTa_portfolio_returns.csv', index=False)
    print("Portfolio returns saved to 'RoBERTa_portfolio_returns.csv'")
    return portfolio_df

def plot_portfolio_returns(portfolio_df, title_suffix=''):
    plt.figure(figsize=(12, 6))

    plt.plot(portfolio_df['Period'], portfolio_df['EW L'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['EW S'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['EW LS'], marker='o', markersize=1, lab
    plt.plot(portfolio_df['Period'], portfolio_df['VW L'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['VW S'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['VW LS'], marker='o', markersize=1, lab
    plt.plot(portfolio_df['Period'], portfolio_df['Market'], marker='o', markersize=1, la

    plt.title(f'Cumulative {title_suffix} Portfolio Returns Over Time')
    plt.xlabel('Period')
    plt.ylabel('Cumulative Log Return')
    plt.legend()
    plt.grid(True)

    plt.gca().xaxis.set_major_locator(mdates.YearLocator())
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    plt.xticks(rotation=45)
    plt.show()

# Example usage for Weekly
df = prepare_data(insample_df, outsample_df)
df = rolling_window_analysis(df)

# Weekly Portfolio
portfolio_df_week = construct_portfolio(df, time_period='Week')
```

```
portfolio_df_week = portfolio_df_week[portfolio_df_week['Period'].dt.year >= 2016]
plot_portfolio_returns(portfolio_df_week, title_suffix='Weekly')
```

Distil BERT

Load Embedding

```
# Load the pre-trained DistilBERT model and tokenizer
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
model = DistilBertModel.from_pretrained('distilbert-base-uncased')

# Load embedded Dataframe
with open('embedding-DistilBERT-AllCompany-NEW.pkl', 'rb') as f:
    insample_df, outsample_df = pd.read_pickle(f)

# Display the first 2 rows of outsample_df to check
filtered_df = outsample_df[outsample_df['headline'] != '[No_Headline]']
filtered_df.head(3)
```

Accuracy per-companies

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Define rolling window parameters
window_size = 365 * 10  # 10 years in days
prediction_period = 365  # 1 year in days

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)
```

```
    # Ensure the length of predictions and test_df matches
    if len(y_pred) != len(test_df):
        raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

    # Store the predictions and true values
    test_df['predicted'] = y_pred

    # Update the company-specific DataFrame with predictions
    company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    print(f'Company: {company}')
    print(f'Training window: {train_df["Date From"].min()} to {train_df["Date From"]
    print(f'Test window: {test_df["Date From"].min()} to {test_df["Date From"].max()
    print(f'Predicted years: {test_df["Date From"].dt.year.unique()}')

# Evaluate the Model for the company
for year in range(2016, 2024):
    year_df = company_df[company_df['Date From'].dt.year == year]

    if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
        print(f"Missing predictions for year {year} for company {company} due to ins
        continue

    y_test = year_df['Future Return Direction'].values
    y_pred = year_df['predicted'].values

    # Remove NaN values in predictions
    valid_indices = ~np.isnan(y_pred)
    y_test = y_test[valid_indices]
    y_pred = y_pred[valid_indices]

    if len(y_pred) == 0 or len(y_test) == 0:
        print(f"Insufficient valid predictions for year {year} for company {company}
        continue

    # Calculate evaluation metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary', pos_label=1)
    recall = recall_score(y_test, y_pred, average='binary', pos_label=1)
    f1 = f1_score(y_test, y_pred, average='binary', pos_label=1)

    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)
    years.append(year)

    print(f'Company: {company}, Year: {year}')
    print(f'Accuracy: {accuracy:.2f}')
    print(f'Precision: {precision:.2f}')
    print(f'Recall: {recall:.2f}')
    print(f'F1 Score: {f1:.2f}')
    print()

# Store the results for the company in a DataFrame
```

```
results_df = pd.DataFrame({
    'Year': years,
    'Accuracy': accuracies,
    'Precision': precisions,
    'Recall': recalls,
    'F1 Score': f1_scores
})

# Save the DataFrame in the dictionary
company_results_dfs[company] = results_df

print(f"Evaluation results for {company}:")
print(results_df)

# Plot accuracy over time for the company
plt.figure(figsize=(10, 6))
plt.plot(results_df['Year'], results_df['Accuracy'], marker='o', label='Accuracy')
plt.plot(results_df['Year'], results_df['Precision'], marker='o', label='Precision')
plt.plot(results_df['Year'], results_df['Recall'], marker='o', label='Recall')
plt.plot(results_df['Year'], results_df['F1 Score'], marker='o', label='F1 Score')

plt.title(f'Performance Metrics Over Time for {company}')
plt.xlabel('Year')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()

# Plot confusion matrices for each year
n_plots = len(years)
n_cols = 4
n_rows = (n_plots // n_cols) + (n_plots % n_cols > 0)

fig, axes = plt.subplots(n_rows, n_cols, figsize=(20, 4 * n_rows))
axes = axes.flatten()

for i, year in enumerate(years):
    year_df = company_df[company_df['Date From'].dt.year == year]
    y_test = year_df['Future Return Direction'].values
    y_pred = year_df['predicted'].values

    # Remove NaN values in predictions
    valid_indices = ~np.isnan(y_pred)
    y_test = y_test[valid_indices]
    y_pred = y_pred[valid_indices]

    if len(y_pred) > 0 and len(y_test) > 0:
        cm = confusion_matrix(y_test, y_pred)
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=logistic_m
        disp.plot(cmap='Blues', ax=axes[i])
        axes[i].set_title(f'{company} {year}')

for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])
```

```
    plt.tight_layout()
    plt.show()

# Access the stored DataFrames
for company, results_df in company_results_dfs.items():
    print(f"\nResults for {company}:")
    print(results_df)



# Concatenate all the company results into one DataFrame
all_results_df = pd.concat(company_results_dfs.values(), keys=company_results_dfs.keys()

# Save the combined DataFrame to a CSV file
all_results_df.to_csv('[EVAL] DistilBERT_all_company_results.csv', index=False)
```

Accuracy all years

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Define rolling window parameters
window_size = 365 * 10  # 10 years in days
prediction_period = 365  # 1 year in days

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)
```

```
        # Ensure the length of predictions and test_df matches
        if len(y_pred) != len(test_df):
            raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

        # Store the predictions and true values
        test_df['predicted'] = y_pred

        # Update the company-specific DataFrame with predictions
        company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    # Evaluate the Model for the company
    for year in range(2016, 2024):
        year_df = company_df[company_df['Date From'].dt.year == year]

        if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
            continue

        y_test = year_df['Future Return Direction'].values
        y_pred = year_df['predicted'].values

        # Remove NaN values in predictions
        valid_indices = ~np.isnan(y_pred)
        y_test = y_test[valid_indices]
        y_pred = y_pred[valid_indices]

        if len(y_pred) == 0 or len(y_test) == 0:
            continue

        # Calculate evaluation metrics
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='binary', pos_label=1)
        recall = recall_score(y_test, y_pred, average='binary', pos_label=1)
        f1 = f1_score(y_test, y_pred, average='binary', pos_label=1)

        accuracies.append(accuracy)
        precisions.append(precision)
        recalls.append(recall)
        f1_scores.append(f1)
        years.append(year)

    # Store the results for the company in a DataFrame
    results_df = pd.DataFrame({
        'Year': years,
        'Accuracy': accuracies,
        'Precision': precisions,
        'Recall': recalls,
        'F1 Score': f1_scores
    })

    # Save the DataFrame in the dictionary
    company_results_dfs[company] = results_df

# Combine results of all companies into a single DataFrame
combined_results = pd.concat(company_results_dfs.values(), ignore_index=True)
```

```
# Calculate average accuracy per year across all companies
average_metrics_per_year = combined_results.groupby('Year').mean().reset_index()

# Print average metrics for each year
print("Average Metrics Per Year:")
print(average_metrics_per_year)

# Plot average accuracy per year
plt.figure(figsize=(10, 6))
sns.set(style='whitegrid')

plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Accuracy'], marker=
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Precision'], marker=
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['Recall'], marker='o
plt.plot(average_metrics_per_year['Year'], average_metrics_per_year['F1 Score'], marker=

plt.title('Average Performance Metrics Over Time Across All Companies')
plt.xlabel('Year')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()
```

Standard Deviation all years

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)

        # Ensure the length of predictions and test_df matches
        if len(y_pred) != len(test_df):
            raise ValueError(f"Mismatch between test data length ({len(test_df)}) and pr

        # Store the predictions and true values
        test_df['predicted'] = y_pred
```

```
        # Update the company-specific DataFrame with predictions
        company_df.loc[test_df.index, 'predicted'] = test_df['predicted']

    # Evaluate the Model for the company
    for year in range(2016, 2024):
        year_df = company_df[company_df['Date From'].dt.year == year]

        if 'predicted' not in year_df.columns or year_df['predicted'].isnull().all():
            continue

        y_test = year_df['Future Return Direction'].values
        y_pred = year_df['predicted'].values

        # Remove NaN values in predictions
        valid_indices = ~np.isnan(y_pred)
        y_test = y_test[valid_indices]
        y_pred = y_pred[valid_indices]

        if len(y_pred) == 0 or len(y_test) == 0:
            continue

        # Calculate evaluation metrics
        accuracy = accuracy_score(y_test, y_pred)

        accuracies.append(accuracy)
        years.append(year)

    # Store the results for the company in a DataFrame
    results_df = pd.DataFrame({
        'Year': years,
        'Accuracy': accuracies
    })

    # Save the DataFrame in the dictionary
    company_results_dfs[company] = results_df

# Combine results of all companies into a single DataFrame
combined_results = pd.concat(company_results_dfs.values(), ignore_index=True)

# Calculate average accuracy per year across all companies
average_metrics_per_year = combined_results.groupby('Year').agg(
    Accuracy_mean=('Accuracy', 'mean'),
    Accuracy_std=('Accuracy', 'std')
).reset_index()

# Print average metrics for each year
print("Average Metrics Per Year:")
print(average_metrics_per_year)

# Get a color sequence from Plotly's default colors
colors = px.colors.qualitative.Plotly

# Function to make the color more transparent
def get_transparent_color(color, alpha=0.2):
    # Convert hex to RGB and then to RGBA
```

```
    hex_color = color.lstrip('#')
    rgb_color = tuple(int(hex_color[i:i+2], 16) for i in (0, 2, 4))
    return f'rgba({rgb_color[0]}, {rgb_color[1]}, {rgb_color[2]}, {alpha})'

# Create a figure
fig = go.Figure()

# Add the mean line
fig.add_trace(go.Scatter(
    x=average_metrics_per_year['Year'],
    y=average_metrics_per_year['Accuracy_mean'],
    mode='lines',
    name='Accuracy',
    line=dict(color=colors[0], width=2)
))

# Add the standard deviation shaded area
fig.add_trace(go.Scatter(
    x=pd.concat([average_metrics_per_year['Year'], average_metrics_per_year['Year'][::-1
    y=pd.concat([average_metrics_per_year['Accuracy_mean'] + average_metrics_per_year['A
                 (average_metrics_per_year['Accuracy_mean'] - average_metrics_per_year['
    fill='toself',
    fillcolor=get_transparent_color(colors[0], alpha=0.2),  # Use the same color with tra
    line=dict(color='rgba(255,255,255,0)'),
    hoverinfo="skip",
    showlegend=False,
    name='Accuracy std dev'
))

# Customize layout
fig.update_layout(
    title='Average Rolling Window Accuracy Over Time Across All Companies',
    xaxis_title='Year',
    yaxis_title='Accuracy',
    template='plotly_white',
    showlegend=True
)

# Show the plot
fig.show()
```

Portfolio

```
def prepare_data(insample_df, outsample_df):
    df = pd.concat([insample_df, outsample_df])
    df['Date From'] = pd.to_datetime(df['Date From'])
    df = df.sort_values(by='Date From')
    available_years = df['Date From'].dt.year.unique()
    print("Years available in the data:", available_years)
    return df

def rolling_window_analysis(df):
    companies = df['companyname'].unique()
    predictions_df = pd.DataFrame()

    for company in companies:
        company_df = df[df['companyname'] == company].copy()
        company_df = company_df.sort_values(by='Date From')

        # Define the rolling window parameters
        start_year = company_df['Date From'].dt.year.min()
        end_year = company_df['Date From'].dt.year.max()
        window_size = 10
        validation_size = 1

        for start in range(start_year, end_year - window_size - validation_size + 1):
            train_start = start
            train_end = start + window_size
            val_start = train_end
            val_end = train_end + validation_size

            train_df = company_df[(company_df['Date From'].dt.year >= train_start) &
                                  (company_df['Date From'].dt.year < train_end)]
            test_df = company_df[(company_df['Date From'].dt.year >= val_start) &
                                 (company_df['Date From'].dt.year < val_end)]

            if len(test_df) == 0 or len(train_df) == 0:
                continue

            X_train = np.vstack(train_df['embedding'].values)
            y_train = train_df['Future Return Direction'].values
            X_test = np.vstack(test_df['embedding'].values)
            y_test = test_df['Future Return Direction'].values

            # Train logistic regression model
            logistic_model = LogisticRegression(max_iter=1000)
            logistic_model.fit(X_train, y_train)

            # Get prediction probabilities
            y_prob = logistic_model.predict_proba(X_test)[:, 1]  # Probability of the pos

            if len(y_prob) != len(test_df):
                raise ValueError(f"Mismatch between test data length ({len(test_df)}) and

            test_df['predicted_prob'] = y_prob
            predictions_df = pd.concat([predictions_df, test_df[['Date From', 'companynam

    df = df.merge(predictions_df, on=['Date From', 'companyname', 'Weekly Compound Return
```

```
    return df

def construct_portfolio(df, time_period='Week'):
    df['Date From'] = pd.to_datetime(df['Date From'])
    if time_period == 'Week':
        df['Period'] = df['Date From'].dt.to_period('W').dt.to_timestamp()
    else:
        raise ValueError("Invalid time_period. Use 'Week'.")

    portfolio_returns = []

    for period, group in df.groupby('Period'):
        if period.year < 2016:
            continue

        # Sort group by predicted_prob descending
        group_sorted = group.sort_values(by='predicted_prob', ascending=False)

        # Select top and bottom companies
        num_top_companies = 5
        num_bottom_companies = 5
        top_companies = group_sorted.head(num_top_companies)
        bottom_companies = group_sorted.tail(num_bottom_companies)

        # Equal-weighted returns
        long_return_eq = np.mean(np.log1p(top_companies['Weekly Compound Return']))
        short_return_eq = np.mean(np.log1p(bottom_companies['Weekly Compound Return']))
        long_short_return_eq = long_return_eq - short_return_eq

        # Value-weighted returns
        long_return_val = np.sum(np.log1p(top_companies['Weekly Compound Return']) * top_
        short_return_val = np.sum(np.log1p(bottom_companies['Weekly Compound Return']) *
        long_short_return_val = long_return_val - short_return_val

        portfolio_returns.append({
            'Period': period,
            'Long Return (Eq)': long_return_eq,
            'Short Return (Eq)': short_return_eq,
            'Long-Short Return (Eq)': long_short_return_eq,
            'Long Return (Val)': long_return_val,
            'Short Return (Val)': short_return_val,
            'Long-Short Return (Val)': long_short_return_val
        })

    portfolio_df = pd.DataFrame(portfolio_returns)
    portfolio_df['EW L'] = portfolio_df['Long Return (Eq)'].cumsum()
    portfolio_df['EW S'] = portfolio_df['Short Return (Eq)'].cumsum()
    portfolio_df['EW LS'] = portfolio_df['Long-Short Return (Eq)'].cumsum()
    portfolio_df['VW L'] = portfolio_df['Long Return (Val)'].cumsum()
    portfolio_df['VW S'] = portfolio_df['Short Return (Val)'].cumsum()
    portfolio_df['VW LS'] = portfolio_df['Long-Short Return (Val)'].cumsum()

    actual_returns = df[df['Date From'].dt.year >= 2016].groupby('Period')['Weekly Compou
    actual_cumulative_returns = np.log1p(actual_returns).cumsum()
    portfolio_df = portfolio_df.merge(actual_cumulative_returns.rename('Market'), on='Pe
```

```
    metrics = {}
    for portfolio in ['EW L', 'EW S', 'EW LS', 'VW L', 'VW S', 'VW LS']:
        returns = portfolio_df[portfolio]

        if returns.isnull().all() or returns.eq(0).all():
            sharpe_ratio = np.nan
            max_drawdown = np.nan
            volatility = np.nan
        else:
            sharpe_ratio = returns.mean() / returns.std() * np.sqrt(52) if returns.std()
            cumulative_returns = returns.cumsum()
            max_drawdown = (cumulative_returns.cummax() - cumulative_returns).max()
            volatility = returns.std() * np.sqrt(52)

        metrics[portfolio] = {
            'Sharpe Ratio': sharpe_ratio
        }

        print(f"Metrics for {portfolio}:")
        print(f"Sharpe Ratio: {sharpe_ratio}")
        print()

    portfolio_df.to_csv('DistilBERT_portfolio_returns.csv', index=False)
    print("Portfolio returns saved to 'DistilBERT_portfolio_returns.csv'")
    return portfolio_df

def plot_portfolio_returns(portfolio_df, title_suffix=''):
    plt.figure(figsize=(12, 6))

    plt.plot(portfolio_df['Period'], portfolio_df['EW L'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['EW S'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['EW LS'], marker='o', markersize=1, lab
    plt.plot(portfolio_df['Period'], portfolio_df['VW L'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['VW S'], marker='o', markersize=1, labe
    plt.plot(portfolio_df['Period'], portfolio_df['VW LS'], marker='o', markersize=1, lab
    plt.plot(portfolio_df['Period'], portfolio_df['Market'], marker='o', markersize=1, la

    plt.title(f'Cumulative {title_suffix} Portfolio Returns Over Time')
    plt.xlabel('Period')
    plt.ylabel('Cumulative Log Return')
    plt.legend()
    plt.grid(True)

    plt.gca().xaxis.set_major_locator(mdates.YearLocator())
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    plt.xticks(rotation=45)
    plt.show()

# Example usage for Weekly
df = prepare_data(insample_df, outsample_df)
df = rolling_window_analysis(df)

# Weekly Portfolio
portfolio_df_week = construct_portfolio(df, time_period='Week')
```

```
portfolio_df_week = portfolio_df_week[portfolio_df_week['Period'].dt.year >= 2016]
plot_portfolio_returns(portfolio_df_week, title_suffix='Weekly')
```

Distil RoBERTa

Load Embedding

```
# Load the pre-trained DistilRoBERTa model and tokenizer
tokenizer = RobertaTokenizer.from_pretrained('distilroberta-base')
model = RobertaModel.from_pretrained('distilroberta-base')

# Load embedded Dataframe
with open('embedding-DistilRoBERTa-AllCompany-NEW.pkl', 'rb') as f:
    insample_df, outsample_df = pd.read_pickle(f)

# Display the first 2 rows of outsample_df to check
filtered_df = outsample_df[outsample_df['headline'] != '[No_Headline]']
filtered_df.head(3)
```

Accuracy per-companies

```
# Combine insample and outsample data for rolling window
df = pd.concat([insample_df, outsample_df])

# Convert 'Date From' to datetime
df['Date From'] = pd.to_datetime(df['Date From'])

# Sort by date
df = df.sort_values(by='Date From')

# Check available years
available_years = df['Date From'].dt.year.unique()
print("Years available in the data:", available_years)

# Define rolling window parameters
window_size = 365 * 10  # 10 years in days
prediction_period = 365  # 1 year in days

# Get unique companies
companies = df['companyname'].unique()

# Dictionary to store DataFrames for each company
company_results_dfs = {}

# Rolling window analysis for each company
for company in companies:
    company_df = df[df['companyname'] == company].copy()

    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []
    years = []

    # Setup time series cross-validation
    tscv = TimeSeriesSplit(n_splits=5)  # You can adjust the number of splits as needed

    for train_index, test_index in tscv.split(company_df):
        train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

        if len(test_df) == 0 or len(train_df) == 0:
            continue

        X_train = np.vstack(train_df['embedding'].values)
        y_train = train_df['Future Return Direction'].values

        X_test = np.vstack(test_df['embedding'].values)
        y_test = test_df['Future Return Direction'].values

        # Train logistic regression model
        logistic_model = LogisticRegression(max_iter=1000)
        logistic_model.fit(X_train, y_train)

        # Make predictions
        y_pred = logistic_model.predict(X_test)
```

```
# Ensure the length of predictions and test_df matches
if len(preds) != len(test_df):
```

BERT Fine Tune

```
from transformers import BertTokenizer, BertForSequenceClassification
import torch
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import TimeSeriesSplit
import matplotlib.pyplot as plt
import matplotlib.dates as mdates


#Step 1: Load Data

# Load the data
insample_df = pd.read_csv('insample_df.csv')
outsample_df = pd.read_csv('outsample_df.csv')


# Step 4: Sentiment Analysis using Sequence Classifier

# Load the pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')

# Function to predict sentiment
def predict_sentiment(text_list, batch_size=16):
    sentiments = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs)
        batch_sentiments = torch.softmax(outputs.logits, dim=1)[:, 1].cpu().numpy()  # Probability of positive sentiment
        sentiments.extend(batch_sentiments)
    return sentiments

# Apply the function to predict sentiment for headlines
insample_df['sentiment'] = predict_sentiment(insample_df['headline'].tolist())
outsample_df['sentiment'] = predict_sentiment(outsample_df['headline'].tolist())

# Ensure sentiments are correctly formed and the same length as the input data
print(f"insample_df sentiments shape: {len(insample_df['sentiment'])}")
print(f"outsample_df sentiments shape: {len(outsample_df['sentiment'])}")
print(f"insample_df shape: {insample_df.shape}")
print(f"outsample_df shape: {outsample_df.shape}")

def prepare_data(insample_df, outsample_df):
    df = pd.concat([insample_df, outsample_df])
    df['Date From'] = pd.to_datetime(df['Date From'])
    df = df.sort_values(by='Date From')
    available_years = df['Date From'].dt.year.unique()
    print("Years available in the data:", available_years)
    return df

def rolling_window_analysis(df):
    companies = df['companyname'].unique()
    predictions_df = pd.DataFrame()

    for company in companies:
        company_df = df[df['companyname'] == company].copy()
        tscv = TimeSeriesSplit(n_splits=5)

        for train_index, test_index in tscv.split(company_df):
            train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

            if len(test_df) == 0 or len(train_df) == 0:
                continue

            X_test = test_df['sentiment'].values
            y_test = test_df['Future Return Direction'].values

            # Directly use sentiment probabilities for prediction
            y_prob = X_test  # Already the probability of positive sentiment

            if len(y_prob) != len(test_df):
                raise ValueError(f"Mismatch between test data length ({len(test_df)}) and predictions ({len(y_prob)})")

            test_df['predicted_prob'] = y_prob
            predictions_df = pd.concat([predictions_df, test_df[['Date From', 'companyname', 'predicted_prob', 'Weekly Compound Returns']]
```

```
        predictions_df = pd.concat([predictions_df, test_df[['Date From', 'companyname', 'predicted_prob', 'Weekly Compound Return']]])

    df = df.merge(predictions_df, on=['Date From', 'companyname', 'Weekly Compound Return'], how='left', suffixes=('', '_pred'))
    return df

def construct_portfolio(df, time_period='Week'):
    df['Date From'] = pd.to_datetime(df['Date From'])
    if time_period == 'Week':
        df['Period'] = df['Date From'].dt.to_period('W').dt.to_timestamp()
    else:
        raise ValueError("Invalid time_period. Use 'Week'.")

    portfolio_returns = []

    for period, group in df.groupby('Period'):
        if period.year < 2016:
            continue

        # Sort group by predicted_prob descending
        group_sorted = group.sort_values(by='predicted_prob', ascending=False)

        # Select top and bottom companies
        num_top_companies = 5
        num_bottom_companies = 5
        top_companies = group_sorted.head(num_top_companies)
        bottom_companies = group_sorted.tail(num_bottom_companies)

        # Equal-weighted returns
        long_return_eq = np.mean(np.log1p(top_companies['Weekly Compound Return']))
        short_return_eq = np.mean(np.log1p(bottom_companies['Weekly Compound Return']))
        long_short_return_eq = long_return_eq - short_return_eq

        # Value-weighted returns
        long_return_val = np.sum(np.log1p(top_companies['Weekly Compound Return']) * top_companies['market_cap']) / np.sum(top_companies
        short_return_val = np.sum(np.log1p(bottom_companies['Weekly Compound Return']) * bottom_companies['market_cap']) / np.sum(bottom_
        long_short_return_val = long_return_val - short_return_val

        portfolio_returns.append({
            'Period': period,
            'Long Return (Eq)': long_return_eq,
            'Short Return (Eq)': short_return_eq,
            'Long-Short Return (Eq)': long_short_return_eq,
            'Long Return (Val)': long_return_val,
            'Short Return (Val)': short_return_val,
            'Long-Short Return (Val)': long_short_return_val
        })

    portfolio_df = pd.DataFrame(portfolio_returns)
    portfolio_df['EW L'] = portfolio_df['Long Return (Eq)'].cumsum()
    portfolio_df['EW S'] = portfolio_df['Short Return (Eq)'].cumsum()
    portfolio_df['EW LS'] = portfolio_df['Long-Short Return (Eq)'].cumsum()
    portfolio_df['VW L'] = portfolio_df['Long Return (Val)'].cumsum()
    portfolio_df['VW S'] = portfolio_df['Short Return (Val)'].cumsum()
    portfolio_df['VW LS'] = portfolio_df['Long-Short Return (Val)'].cumsum()

    actual_returns = df[df['Date From'].dt.year >= 2016].groupby('Period')['Weekly Compound Return'].mean()
    actual_cumulative_returns = np.log1p(actual_returns).cumsum()
    portfolio_df = portfolio_df.merge(actual_cumulative_returns.rename('Market'), on='Period', how='left')

    metrics = {}
    for portfolio in ['EW L', 'EW S', 'EW LS', 'VW L', 'VW S', 'VW LS']:
        returns = portfolio_df[portfolio]

        if returns.isnull().all() or returns.eq(0).all():
            sharpe_ratio = np.nan
            max_drawdown = np.nan
            volatility = np.nan
        else:
            sharpe_ratio = returns.mean() / returns.std() * np.sqrt(52) if returns.std() != 0 else np.nan
            cumulative_returns = returns.cumsum()
            max_drawdown = (cumulative_returns.cummax() - cumulative_returns).max()
            volatility = returns.std() * np.sqrt(52)

        metrics[portfolio] = {
            'Sharpe Ratio': sharpe_ratio
        }

        print(f"Metrics for {portfolio}:")
        print(f"Sharpe Ratio: {sharpe_ratio}")
        print()

    portfolio_df.to_csv('BERTa_Finetune_portfolio_returns.csv', index=False)
    print("Portfolio returns saved to 'BERTa_Finetune_portfolio_returns.csv'")
```

```
    # Plotting portfolio returns
    plot_portfolio_returns(portfolio_df, title_suffix='Weekly')

    return portfolio_df

def plot_portfolio_returns(portfolio_df, title_suffix=''):
    plt.figure(figsize=(12, 6))

    plt.plot(portfolio_df['Period'], portfolio_df['EW L'], marker='o', markersize=1, label='EW L')
    plt.plot(portfolio_df['Period'], portfolio_df['EW S'], marker='o', markersize=1, label='EW S')
    plt.plot(portfolio_df['Period'], portfolio_df['EW LS'], marker='o', markersize=1, label='EW LS')
    plt.plot(portfolio_df['Period'], portfolio_df['VW L'], marker='o', markersize=1, label='VW L')
    plt.plot(portfolio_df['Period'], portfolio_df['VW S'], marker='o', markersize=1, label='VW S')
    plt.plot(portfolio_df['Period'], portfolio_df['VW LS'], marker='o', markersize=1, label='VW LS')
    plt.plot(portfolio_df['Period'], portfolio_df['Market'], marker='o', markersize=1, label='Market')

    plt.title(f'Cumulative {title_suffix} Portfolio Returns Over Time')
    plt.xlabel('Period')
    plt.ylabel('Cumulative Log Return')
    plt.legend()
    plt.grid(True)

    plt.gca().xaxis.set_major_locator(mdates.YearLocator())
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    plt.xticks(rotation=45)
    plt.show()

# Example usage for Weekly
df = prepare_data(insample_df, outsample_df)
df = rolling_window_analysis(df)

# Weekly Portfolio
portfolio_df_week = construct_portfolio(df, time_period='Week')
portfolio_df_week = portfolio_df_week[portfolio_df_week['Period'].dt.year >= 2016]
plot_portfolio_returns(portfolio_df_week, title_suffix='Weekly')
```

RoBERTa Finetune

```
from transformers import RobertaTokenizer, RobertaModel, RobertaForSequenceClassification

# Load the pre-trained BERT model and tokenizer
tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
model = RobertaForSequenceClassification.from_pretrained('roberta-base')

# Function to predict sentiment
def predict_sentiment(text_list, batch_size=16):
    sentiments = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs)
        batch_sentiments = torch.softmax(outputs.logits, dim=1)[:, 1].cpu().numpy()  # Probability of positive sentiment
        sentiments.extend(batch_sentiments)
    return sentiments

# Apply the function to predict sentiment for headlines
insample_df['sentiment'] = predict_sentiment(insample_df['headline'].tolist())
outsample_df['sentiment'] = predict_sentiment(outsample_df['headline'].tolist())

# Ensure sentiments are correctly formed and the same length as the input data
print(f"insample_df sentiments shape: {len(insample_df['sentiment'])}")
print(f"outsample_df sentiments shape: {len(outsample_df['sentiment'])}")
print(f"insample_df shape: {insample_df.shape}")
print(f"outsample_df shape: {outsample_df.shape}")

def prepare_data(insample_df, outsample_df):
    df = pd.concat([insample_df, outsample_df])
    df['Date From'] = pd.to_datetime(df['Date From'])
    df = df.sort_values(by='Date From')
    available_years = df['Date From'].dt.year.unique()
    print("Years available in the data:", available_years)
    return df

def rolling_window_analysis(df):
    companies = df['companyname'].unique()
    predictions_df = pd.DataFrame()
```

```
    for company in companies:
        company_df = df[df['companyname'] == company].copy()
        tscv = TimeSeriesSplit(n_splits=5)

        for train_index, test_index in tscv.split(company_df):
            train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

            if len(test_df) == 0 or len(train_df) == 0:
                continue

            X_test = test_df['sentiment'].values
            y_test = test_df['Future Return Direction'].values

            # Directly use sentiment probabilities for prediction
            y_prob = X_test  # Already the probability of positive sentiment

            if len(y_prob) != len(test_df):
                raise ValueError(f"Mismatch between test data length ({len(test_df)}) and predictions ({len(y_prob)})")

            test_df['predicted_prob'] = y_prob
            predictions_df = pd.concat([predictions_df, test_df[['Date From', 'companyname', 'predicted_prob', 'Weekly Compound Return']]

    df = df.merge(predictions_df, on=['Date From', 'companyname', 'Weekly Compound Return'], how='left', suffixes=('', '_pred'))
    return df

def construct_portfolio(df, time_period='Week'):
    df['Date From'] = pd.to_datetime(df['Date From'])
    if time_period == 'Week':
        df['Period'] = df['Date From'].dt.to_period('W').dt.to_timestamp()
    else:
        raise ValueError("Invalid time_period. Use 'Week'.")

    portfolio_returns = []

    for period, group in df.groupby('Period'):
        if period.year < 2016:
            continue

        # Sort group by predicted_prob descending
        group_sorted = group.sort_values(by='predicted_prob', ascending=False)

        # Select top and bottom companies
        num_top_companies = 5
        num_bottom_companies = 5
        top_companies = group_sorted.head(num_top_companies)
        bottom_companies = group_sorted.tail(num_bottom_companies)

        # Equal-weighted returns
        long_return_eq = np.mean(np.log1p(top_companies['Weekly Compound Return']))
        short_return_eq = np.mean(np.log1p(bottom_companies['Weekly Compound Return']))
        long_short_return_eq = long_return_eq - short_return_eq

        # Value-weighted returns
        long_return_val = np.sum(np.log1p(top_companies['Weekly Compound Return']) * top_companies['market_cap']) / np.sum(top_companies
        short_return_val = np.sum(np.log1p(bottom_companies['Weekly Compound Return']) * bottom_companies['market_cap']) / np.sum(bottom_
        long_short_return_val = long_return_val - short_return_val

        portfolio_returns.append({
            'Period': period,
            'Long Return (Eq)': long_return_eq,
            'Short Return (Eq)': short_return_eq,
            'Long-Short Return (Eq)': long_short_return_eq,
            'Long Return (Val)': long_return_val,
            'Short Return (Val)': short_return_val,
            'Long-Short Return (Val)': long_short_return_val
        })

    portfolio_df = pd.DataFrame(portfolio_returns)
    portfolio_df['EW L'] = portfolio_df['Long Return (Eq)'].cumsum()
    portfolio_df['EW S'] = portfolio_df['Short Return (Eq)'].cumsum()
    portfolio_df['EW LS'] = portfolio_df['Long-Short Return (Eq)'].cumsum()
    portfolio_df['VW L'] = portfolio_df['Long Return (Val)'].cumsum()
    portfolio_df['VW S'] = portfolio_df['Short Return (Val)'].cumsum()
    portfolio_df['VW LS'] = portfolio_df['Long-Short Return (Val)'].cumsum()

    actual_returns = df[df['Date From'].dt.year >= 2016].groupby('Period')['Weekly Compound Return'].mean()
    actual_cumulative_returns = np.log1p(actual_returns).cumsum()
    portfolio_df = portfolio_df.merge(actual_cumulative_returns.rename('Market'), on='Period', how='left')

    metrics = {}
    for portfolio in ['EW L', 'EW S', 'EW LS', 'VW L', 'VW S', 'VW LS']:
        returns = portfolio_df[portfolio]
```

```
        if returns.isnull().all() or returns.eq(0).all():
            sharpe_ratio = np.nan
            max_drawdown = np.nan
            volatility = np.nan
        else:
            sharpe_ratio = returns.mean() / returns.std() * np.sqrt(52) if returns.std() != 0 else np.nan
            cumulative_returns = returns.cumsum()
            max_drawdown = (cumulative_returns.cummax() - cumulative_returns).max()
            volatility = returns.std() * np.sqrt(52)

        metrics[portfolio] = {
            'Sharpe Ratio': sharpe_ratio
        }

        print(f"Metrics for {portfolio}:")
        print(f"Sharpe Ratio: {sharpe_ratio}")
        print()

    portfolio_df.to_csv('RoBERTa_Finetune_portfolio_returns.csv', index=False)
    print("Portfolio returns saved to 'RoBERTa_Finetune_portfolio_returns.csv'")

    # Plotting portfolio returns
    plot_portfolio_returns(portfolio_df, title_suffix='Weekly')

    return portfolio_df

def plot_portfolio_returns(portfolio_df, title_suffix=''):
    plt.figure(figsize=(12, 6))

    plt.plot(portfolio_df['Period'], portfolio_df['EW L'], marker='o', markersize=1, label='EW L')
    plt.plot(portfolio_df['Period'], portfolio_df['EW S'], marker='o', markersize=1, label='EW S')
    plt.plot(portfolio_df['Period'], portfolio_df['EW LS'], marker='o', markersize=1, label='EW LS')
    plt.plot(portfolio_df['Period'], portfolio_df['VW L'], marker='o', markersize=1, label='VW L')
    plt.plot(portfolio_df['Period'], portfolio_df['VW S'], marker='o', markersize=1, label='VW S')
    plt.plot(portfolio_df['Period'], portfolio_df['VW LS'], marker='o', markersize=1, label='VW LS')
    plt.plot(portfolio_df['Period'], portfolio_df['Market'], marker='o', markersize=1, label='Market')

    plt.title(f'Cumulative {title_suffix} Portfolio Returns Over Time')
    plt.xlabel('Period')
    plt.ylabel('Cumulative Log Return')
    plt.legend()
    plt.grid(True)

    plt.gca().xaxis.set_major_locator(mdates.YearLocator())
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    plt.xticks(rotation=45)
    plt.show()

# Example usage for Weekly
df = prepare_data(insample_df, outsample_df)
df = rolling_window_analysis(df)

# Weekly Portfolio
portfolio_df_week = construct_portfolio(df, time_period='Week')
portfolio_df_week = portfolio_df_week[portfolio_df_week['Period'].dt.year >= 2016]
plot_portfolio_returns(portfolio_df_week, title_suffix='Weekly')
```

Distil BERT Finetune

```
from transformers import DistilBertTokenizer, DistilBertForSequenceClassification

# Load the pre-trained BERT model and tokenizer
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased')

# Function to predict sentiment
def predict_sentiment(text_list, batch_size=16):
    sentiments = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs)
        batch_sentiments = torch.softmax(outputs.logits, dim=1)[:, 1].cpu().numpy()  # Probability of positive sentiment
        sentiments.extend(batch_sentiments)
    return sentiments

# Apply the function to predict sentiment for headlines
insample_df['sentiment'] = predict_sentiment(insample_df['headline'].tolist())
outsample_df['sentiment'] = predict_sentiment(outsample_df['headline'].tolist())

# Ensure sentiments are correctly formed and the same length as the input data
print(f"insample_df sentiments shape: {len(insample_df['sentiment'])}")
print(f"outsample_df sentiments shape: {len(outsample_df['sentiment'])}")
print(f"insample_df shape: {insample_df.shape}")
print(f"outsample_df shape: {outsample_df.shape}")

def prepare_data(insample_df, outsample_df):
    df = pd.concat([insample_df, outsample_df])
    df['Date From'] = pd.to_datetime(df['Date From'])
    df = df.sort_values(by='Date From')
    available_years = df['Date From'].dt.year.unique()
    print("Years available in the data:", available_years)
    return df

def rolling_window_analysis(df):
    companies = df['companyname'].unique()
    predictions_df = pd.DataFrame()

    for company in companies:
        company_df = df[df['companyname'] == company].copy()
        tscv = TimeSeriesSplit(n_splits=5)

        for train_index, test_index in tscv.split(company_df):
            train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

            if len(test_df) == 0 or len(train_df) == 0:
                continue

            X_test = test_df['sentiment'].values
            y_test = test_df['Future Return Direction'].values

            # Directly use sentiment probabilities for prediction
            y_prob = X_test  # Already the probability of positive sentiment

            if len(y_prob) != len(test_df):
                raise ValueError(f"Mismatch between test data length ({len(test_df)}) and predictions ({len(y_prob)})")

            test_df['predicted_prob'] = y_prob
            predictions_df = pd.concat([predictions_df, test_df[['Date From', 'companyname', 'predicted_prob', 'Weekly Compound Return'

    df = df.merge(predictions_df, on=['Date From', 'companyname', 'Weekly Compound Return'], how='left', suffixes=('', '_pred'))
    return df

def construct_portfolio(df, time_period='Week'):
    df['Date From'] = pd.to_datetime(df['Date From'])
    if time_period == 'Week':
        df['Period'] = df['Date From'].dt.to_period('W').dt.to_timestamp()
    else:
        raise ValueError("Invalid time_period. Use 'Week'.")

    portfolio_returns = []

    for period, group in df.groupby('Period'):
        if period.year < 2016:
            continue

        # Sort group by predicted_prob descending
        group_sorted = group.sort_values(by='predicted_prob', ascending=False)

        # Select top and bottom companies
```

```
        num_top_companies = 5
        num_bottom_companies = 5
        top_companies = group_sorted.head(num_top_companies)
        bottom_companies = group_sorted.tail(num_bottom_companies)

        # Equal-weighted returns
        long_return_eq = np.mean(np.log1p(top_companies['Weekly Compound Return']))
        short_return_eq = np.mean(np.log1p(bottom_companies['Weekly Compound Return']))
        long_short_return_eq = long_return_eq - short_return_eq

        # Value-weighted returns
        long_return_val = np.sum(np.log1p(top_companies['Weekly Compound Return']) * top_companies['market_cap']) / np.sum(top_companie
        short_return_val = np.sum(np.log1p(bottom_companies['Weekly Compound Return']) * bottom_companies['market_cap']) / np.sum(botto
        long_short_return_val = long_return_val - short_return_val

        portfolio_returns.append({
            'Period': period,
            'Long Return (Eq)': long_return_eq,
            'Short Return (Eq)': short_return_eq,
            'Long-Short Return (Eq)': long_short_return_eq,
            'Long Return (Val)': long_return_val,
            'Short Return (Val)': short_return_val,
            'Long-Short Return (Val)': long_short_return_val
        })

    portfolio_df = pd.DataFrame(portfolio_returns)
    portfolio_df['EW L'] = portfolio_df['Long Return (Eq)'].cumsum()
    portfolio_df['EW S'] = portfolio_df['Short Return (Eq)'].cumsum()
    portfolio_df['EW LS'] = portfolio_df['Long-Short Return (Eq)'].cumsum()
    portfolio_df['VW L'] = portfolio_df['Long Return (Val)'].cumsum()
    portfolio_df['VW S'] = portfolio_df['Short Return (Val)'].cumsum()
    portfolio_df['VW LS'] = portfolio_df['Long-Short Return (Val)'].cumsum()

    actual_returns = df[df['Date From'].dt.year >= 2016].groupby('Period')['Weekly Compound Return'].mean()
    actual_cumulative_returns = np.log1p(actual_returns).cumsum()
    portfolio_df = portfolio_df.merge(actual_cumulative_returns.rename('Market'), on='Period', how='left')

    metrics = {}
    for portfolio in ['EW L', 'EW S', 'EW LS', 'VW L', 'VW S', 'VW LS']:
        returns = portfolio_df[portfolio]

        if returns.isnull().all() or returns.eq(0).all():
            sharpe_ratio = np.nan
            max_drawdown = np.nan
            volatility = np.nan
        else:
            sharpe_ratio = returns.mean() / returns.std() * np.sqrt(52) if returns.std() != 0 else np.nan
            cumulative_returns = returns.cumsum()
            max_drawdown = (cumulative_returns.cummax() - cumulative_returns).max()
            volatility = returns.std() * np.sqrt(52)

        metrics[portfolio] = {
            'Sharpe Ratio': sharpe_ratio
        }

        print(f"Metrics for {portfolio}:")
        print(f"Sharpe Ratio: {sharpe_ratio}")
        print()

    portfolio_df.to_csv('DistilBERT_Finetune_portfolio_returns.csv', index=False)
    print("Portfolio returns saved to 'DistilBERT_Finetune_portfolio_returns.csv'")

    # Plotting portfolio returns
    plot_portfolio_returns(portfolio_df, title_suffix='Weekly')

    return portfolio_df

def plot_portfolio_returns(portfolio_df, title_suffix=''):
    plt.figure(figsize=(12, 6))

    plt.plot(portfolio_df['Period'], portfolio_df['EW L'], marker='o', markersize=1, label='EW L')
    plt.plot(portfolio_df['Period'], portfolio_df['EW S'], marker='o', markersize=1, label='EW S')
    plt.plot(portfolio_df['Period'], portfolio_df['EW LS'], marker='o', markersize=1, label='EW LS')
    plt.plot(portfolio_df['Period'], portfolio_df['VW L'], marker='o', markersize=1, label='VW L')
    plt.plot(portfolio_df['Period'], portfolio_df['VW S'], marker='o', markersize=1, label='VW S')
    plt.plot(portfolio_df['Period'], portfolio_df['VW LS'], marker='o', markersize=1, label='VW LS')
    plt.plot(portfolio_df['Period'], portfolio_df['Market'], marker='o', markersize=1, label='Market')

    plt.title(f'Cumulative {title_suffix} Portfolio Returns Over Time')
    plt.xlabel('Period')
    plt.ylabel('Cumulative Log Return')
    plt.legend()
```

```
    plt.grid(True)

    plt.gca().xaxis.set_major_locator(mdates.YearLocator())
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    plt.xticks(rotation=45)
    plt.show()

# Example usage for Weekly
df = prepare_data(insample_df, outsample_df)
df = rolling_window_analysis(df)

# Weekly Portfolio
portfolio_df_week = construct_portfolio(df, time_period='Week')
portfolio_df_week = portfolio_df_week[portfolio_df_week['Period'].dt.year >= 2016]
plot_portfolio_returns(portfolio_df_week, title_suffix='Weekly')
```

Distil RoBERTa Finetune

```
from transformers import RobertaTokenizer, RobertaForSequenceClassification

# Load the pre-trained DistilRoBERTa model and tokenizer
tokenizer = RobertaTokenizer.from_pretrained('distilroberta-base')
model = RobertaForSequenceClassification.from_pretrained('distilroberta-base')

# Function to predict sentiment
def predict_sentiment(text_list, batch_size=16):
    sentiments = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs)
        batch_sentiments = torch.softmax(outputs.logits, dim=1)[:, 1].cpu().numpy()  # Probability of positive sentiment
        sentiments.extend(batch_sentiments)
    return sentiments

# Apply the function to predict sentiment for headlines
insample_df['sentiment'] = predict_sentiment(insample_df['headline'].tolist())
outsample_df['sentiment'] = predict_sentiment(outsample_df['headline'].tolist())

# Ensure sentiments are correctly formed and the same length as the input data
print(f"insample_df sentiments shape: {len(insample_df['sentiment'])}")
print(f"outsample_df sentiments shape: {len(outsample_df['sentiment'])}")
print(f"insample_df shape: {insample_df.shape}")
print(f"outsample_df shape: {outsample_df.shape}")

def prepare_data(insample_df, outsample_df):
    df = pd.concat([insample_df, outsample_df])
    df['Date From'] = pd.to_datetime(df['Date From'])
    df = df.sort_values(by='Date From')
    available_years = df['Date From'].dt.year.unique()
    print("Years available in the data:", available_years)
    return df

def rolling_window_analysis(df):
    companies = df['companyname'].unique()
    predictions_df = pd.DataFrame()

    for company in companies:
        company_df = df[df['companyname'] == company].copy()
        tscv = TimeSeriesSplit(n_splits=5)

        for train_index, test_index in tscv.split(company_df):
            train_df, test_df = company_df.iloc[train_index], company_df.iloc[test_index]

            if len(test_df) == 0 or len(train_df) == 0:
                continue

            X_test = test_df['sentiment'].values
            y_test = test_df['Future Return Direction'].values

            # Directly use sentiment probabilities for prediction
            y_prob = X_test  # Already the probability of positive sentiment

            if len(y_prob) != len(test_df):
                raise ValueError(f"Mismatch between test data length ({len(test_df)}) and predictions ({len(y_prob)})")

            test_df['predicted_prob'] = y_prob
            predictions_df = pd.concat([predictions_df, test_df[['Date From', 'companyname', 'predicted_prob', 'Weekly Compound Returr

    df = df.merge(predictions_df, on=['Date From', 'companyname', 'Weekly Compound Return'], how='left', suffixes=('', '_pred'))
    return df

def construct_portfolio(df, time_period='Week'):
    df['Date From'] = pd.to_datetime(df['Date From'])
    if time_period == 'Week':
        df['Period'] = df['Date From'].dt.to_period('W').dt.to_timestamp()
    else:
        raise ValueError("Invalid time_period. Use 'Week'.")

    portfolio_returns = []

    for period, group in df.groupby('Period'):
        if period.year < 2016:
            continue

        # Sort group by predicted_prob descending
        group_sorted = group.sort_values(by='predicted_prob', ascending=False)

        # Select top and bottom companies
```

```
        num_top_companies = 5
        num_bottom_companies = 5
        top_companies = group_sorted.head(num_top_companies)
        bottom_companies = group_sorted.tail(num_bottom_companies)

        # Equal-weighted returns
        long_return_eq = np.mean(np.log1p(top_companies['Weekly Compound Return']))
        short_return_eq = np.mean(np.log1p(bottom_companies['Weekly Compound Return']))
        long_short_return_eq = long_return_eq - short_return_eq

        # Value-weighted returns
        long_return_val = np.sum(np.log1p(top_companies['Weekly Compound Return']) * top_companies['market_cap']) / np.sum(top_compani
        short_return_val = np.sum(np.log1p(bottom_companies['Weekly Compound Return']) * bottom_companies['market_cap']) / np.sum(bott
        long_short_return_val = long_return_val - short_return_val

        portfolio_returns.append({
            'Period': period,
            'Long Return (Eq)': long_return_eq,
            'Short Return (Eq)': short_return_eq,
            'Long-Short Return (Eq)': long_short_return_eq,
            'Long Return (Val)': long_return_val,
            'Short Return (Val)': short_return_val,
            'Long-Short Return (Val)': long_short_return_val
        })

    portfolio_df = pd.DataFrame(portfolio_returns)
    portfolio_df['EW L'] = portfolio_df['Long Return (Eq)'].cumsum()
    portfolio_df['EW S'] = portfolio_df['Short Return (Eq)'].cumsum()
    portfolio_df['EW LS'] = portfolio_df['Long-Short Return (Eq)'].cumsum()
    portfolio_df['VW L'] = portfolio_df['Long Return (Val)'].cumsum()
    portfolio_df['VW S'] = portfolio_df['Short Return (Val)'].cumsum()
    portfolio_df['VW LS'] = portfolio_df['Long-Short Return (Val)'].cumsum()

    actual_returns = df[df['Date From'].dt.year >= 2016].groupby('Period')['Weekly Compound Return'].mean()
    actual_cumulative_returns = np.log1p(actual_returns).cumsum()
    portfolio_df = portfolio_df.merge(actual_cumulative_returns.rename('Market'), on='Period', how='left')
```

FinBERT Finetune

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Load the pre-trained FinBERT model and tokenizer
tokenizer = AutoTokenizer.from_pretrained('yiyanghkust/finbert-tone', use_fast=False)
model = AutoModelForSequenceClassification.from_pretrained('yiyanghkust/finbert-tone')

# Function to predict sentiment
def predict_sentiment(text_list, batch_size=16):
    sentiments = []
    for i in range(0, len(text_list), batch_size):
        batch = text_list[i:i + batch_size]
        inputs = tokenizer(batch, return_tensors='pt', padding=True, truncation=True)
        with torch.no_grad():
            outputs = model(**inputs)
        batch_sentiments = torch.softmax(outputs.logits, dim=1)[:, 1].cpu().numpy()  # Probability of positive sentiment
        sentiments.extend(batch_sentiments)
    return sentiments
```