THE COOPER UNION FOR
THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

A Trading Strategy Based on Sentiment Analysis and Transformers

By
Danny Hong

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Engineering

January 2025

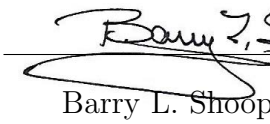
Advisor

Professor Fred L. Fontaine

THE COOPER UNION FOR
THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

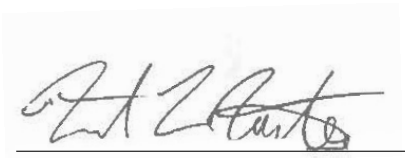
This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.



05/06/2025

Barry L. Shoop, PhD., P.E. Date

Dean, Albert Nerken School of Engineering



05/06/2025

Professor Fred L. Fontaine Date

Candidate's Thesis Advisor

Acknowledgments

I would first like to give thanks to my thesis advisor, Professor Fontaine, who has supported me immensely throughout my time at the Cooper Union. Professor Fontaine was the one who had initially suggested to me what my thesis topic should be. He would regularly check in with me on my progress and give me creative ideas that would help improve my work. Most of all, he has kept my spirits up during difficult moments when I felt like giving up and helped me stay on course to complete the project. Professor Fontaine's commitment to me and other Cooper Union students is truly admirable and cannot be understated enough.

I would also like to thank all the friends and faculty members that I have come across in my time as a student here at the Cooper Union. I have learned very much from the people that I have come across here and will remember the relationships that I have formed with great fondness.

Finally, I would like to thank my family. My parents have supported me every step of the way throughout my educational journey at the Cooper Union. I would not have be able to complete my degrees without their continual encouragement and support. Hopefully, they are proud of my accomplishments and believe that their investment in my academic growth and development has paid off.

Abstract

Most previous work at creating trading strategies for stock data have relied primarily on using historical stock prices and other technical indicators, such as moving averages, to forecast market trends. While these models do certainly provide insights into stock price patterns, they often fail to account for the human-driven aspects of market behavior, such as the influence of news sentiment or public opinion. This work addresses that gap by developing a trading strategy that integrates financial stock data with sentiment analysis using a transformer model. The goal is to demonstrate that incorporating sentiment scores derived from financial news sources alongside traditional stock metrics can improve predictions of buy, sell, or hold decisions and therefore lead to better profit and loss outcomes over time compared to a model that uses stock data alone.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Previous Trading Strategy Approaches

Stock price forecasting is inherently difficult due to the volatile nature of financial markets. Traditional approaches often focus on analyzing past price data to identify patterns and trends that might help predict future movements. As explained in [3], models like ARIMA (AutoRegressive Integrated Moving Average) and LSTM (Long Short-Term Memory networks) process historical stock price data sequentially, meaning they analyze data one time step at a time in the order it comes in. While this approach aligns well with the structure of time-series data, it also has some drawbacks. For example, these models are strongly based on the assumption that past values alone can predict future trends. In actuality, financial markets are influenced by a broader set of factors, including breaking news and public sentiment, making it essential to incorporate and analyze multi-dimensional data sources. Sequential models struggle to handle such diverse data effectively as they are designed mainly to analyze purely sequential inputs. Therefore, these models are inefficient and less adaptable when handling multiple, unaligned data streams, such as combining textual sentiment data with structured market data.

Sequential models also face challenges in capturing long-term dependencies within financial data. Markets usually exhibit both short-term volatility and long-

term trends that are influenced by economic shifts. Short-term volatility involves rapid price changes driven by events such as geopolitical news, while long-term trends consist of gradual movements shaped by factors like monetary policy or technological changes. Although models like LSTM aim to maintain relevant information through memory gates, these states degrade over long sequences, leading to a loss of context from earlier time points [4]. Similarly, ARIMA's reliance on historical averages and error terms also fails to account for cross-modal relationships.

1.2 The Proposed Approach

This project aims to address the shortcomings of traditional stock price forecasting models by introducing a novel approach that combines financial and sentiment data using a multi-head transformer model. Unlike sequential models, transformer models use attention mechanisms to process input data in parallel, allowing them to efficiently capture long-range dependencies and complex interactions between data points [1]. This means a transformer can analyze how events from distant time steps influence present outcomes without losing crucial early information. In addition, transformers are very effective at integrating data like historical prices alongside unstructured data like news sentiment by aligning and processing them simultaneously. They can dynamically weigh the importance of different inputs, such as assigning more relevance to sudden breaking news while minimizing the importance of older data when necessary. As a result, this ability to capture cross-modal relationships is vital in an area as intricate as stock price forecasting as it makes it more adaptable to changes in data.

To achieve this goal, this study implements a multi-head transformer model trained to predict buy, sell, or hold signals based on adjusted close stock prices for stocks. The model is optimized using a custom loss function designed to maximize a profit function, ensuring that the predictions lead to trading actions with the

potential for maximum profitability. An important aspect of this approach is the integration of sentiment analysis with traditional financial data, which is done by analyzing historical financial news headlines for certain stocks using the VADER tool to generate compound sentiment scores that represent the market's mood regarding specific stocks at various times [5]. These scores are then incorporated alongside stock price features such as open, high, low, close, volume, and the S&P 500 index. To evaluate the added value of sentiment data, two versions of the model are trained: one with sentiment scores included as a training input and one without. The results are then compared to determine the impact of sentiment data on the model's ability to predict profitable buy, sell, or hold signals.

1.3 Structural Overview

Chapter 2 introduces the core concepts of transformers and covers their architecture and functionality. It provides a detailed breakdown of the encoder and decoder components, including their individual structures and key differences. Numerical data embeddings and essential transformer operations such as multi-head attention, feed-forward networks, and residual connections are also explained. Chapter 3 explores sentiment analysis with VADER, highlighting its advantages and providing a step-by-step explanation of how VADER computes sentiment scores. Chapter 4 focuses on financial analysis, outlining metrics such as OHLC (Open, High, Low, Close), adjusted close price, volume, and the S&P 500 index. Chapter 5 describes the implementation of the proposed approach, including the model parameters, optimizer updates, and the hybrid loss function that integrates profit-based rewards with categorical cross-entropy. It also provides details on data collection and preprocessing, which include the aggregation of daily sentiment scores and stock price data transformation. Chapter 6 evaluates the results of the proposed method and discusses the findings. Finally, Chapter 7 summarizes the method and results and ends with remarks on future research ideas.

Chapter 2

Transformers

2.1 Overview

Transformers are built on an encoder-decoder architecture, which excels in sequence-to-sequence tasks where an input sequence is mapped to an output sequence. This includes tasks such as translating textual data from one language to another or summarizing long texts in just a few sentences. Shown in Figure 2.1, each block (encoder or decoder) uses self-attention and feed-forward layers to process information. As explained in [1], the encoder transforms the input sequence $X = [x_1, x_2, \ldots, x_n]$ into a sequence of hidden representations $H = [h_1, h_2, \ldots, h_n]$ while the decoder predicts the output sequence $Y = [y_1, y_2, \ldots, y_m]$ using H.

When applying the transformer architecture to numerical data such as stock prices and sentiment scores, several adjustments are needed since transformers were originally designed for natural language processing tasks, where data would typically be represented as sequences of textual tokens. A token is a single unit of meaningful data often used in Natural Language Processing (NLP) tasks to represent a basic component of text [6]. In this case, the input sequence will consist of numerical values rather than discrete tokens, although the mechanism for processing and learning representations remains mostly the same. As a result, some key concepts such as embedding tokens in a transformer model must be

adapted accordingly.

2.2 Encoder and Decoder Architecture

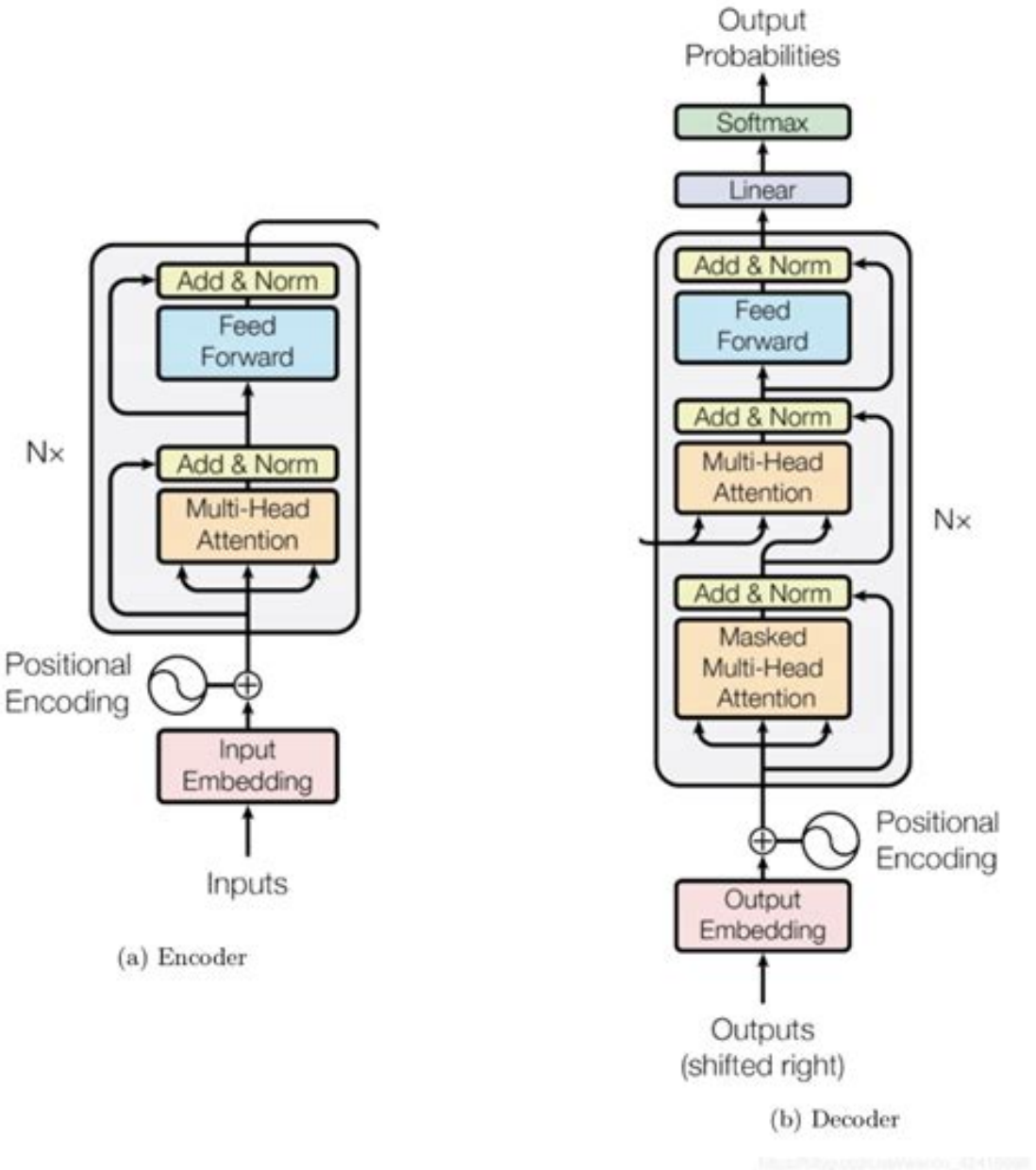


Figure 2.1: Encoder-Decoder Structure in Transformers (Figure 1 from [1])

2.2.1 Encoder

The encoder in a transformer model consists of a stack of identical layers. Each layer is made up of two main components: multi-head self-attention and feed-

forward neural networks [1]. The multi-head self-attention mechanism allows the model to attend to different parts of the input sequence simultaneously, enabling it to capture complex relationships and dependencies. Each attention head focuses on a different subset of relationships or features, which enhances the model's ability to capture diverse aspects of the input. The self-attention mechanism ensures that each element of the sequence, such as a word or token, can interact with all other elements in the sequence. For example, in a sentence, a word at the beginning can interact with words at the end of the sentence. This interaction helps the model to understand the relationships between words regardless of their distance in the sequence.

Following the self-attention layer, the encoder includes a feed-forward neural network. This is typically a two-layer fully connected network applied to each position independently. After the self-attention mechanism has processed the dependencies between elements in the sequence, the feed-forward neural network refines the representation at each position.

The encoder's primary task is to map the input sequence, such as a sentence in the case of text, into a continuous representation or context vector that captures meaningful relationships. The input embeddings are passed through these stacked layers, and the final output of the encoder produces an increasingly nuanced understanding of the input sequence, with progressively more complex features learned at deeper layers.

2.2.2 Decoder

The decoder shares a similar multi-layered structure with the encoder but introduces additional mechanisms to handle sequence generation. The decoder is responsible for producing an output sequence based on the representations learned by the encoder. Like the encoder, the decoder also contains stacked layers of multi-head attention and feed-forward networks, but the attention mechanisms differ slightly due to the autoregressive nature of the generation process.

The decoder first applies a masked multi-head self-attention mechanism. This is different from the self-attention of the encoder in that the attention is masked to prevent each position from accessing future positions in the sequence [1]. Masking ensures that each token in the output sequence only has access to previously generated tokens (or input tokens in certain models), making the decoder autoregressive in nature. By masking future tokens, the model is restricted to seeing only past tokens while generating the next token, which maintains the temporal structure needed for sequential prediction.

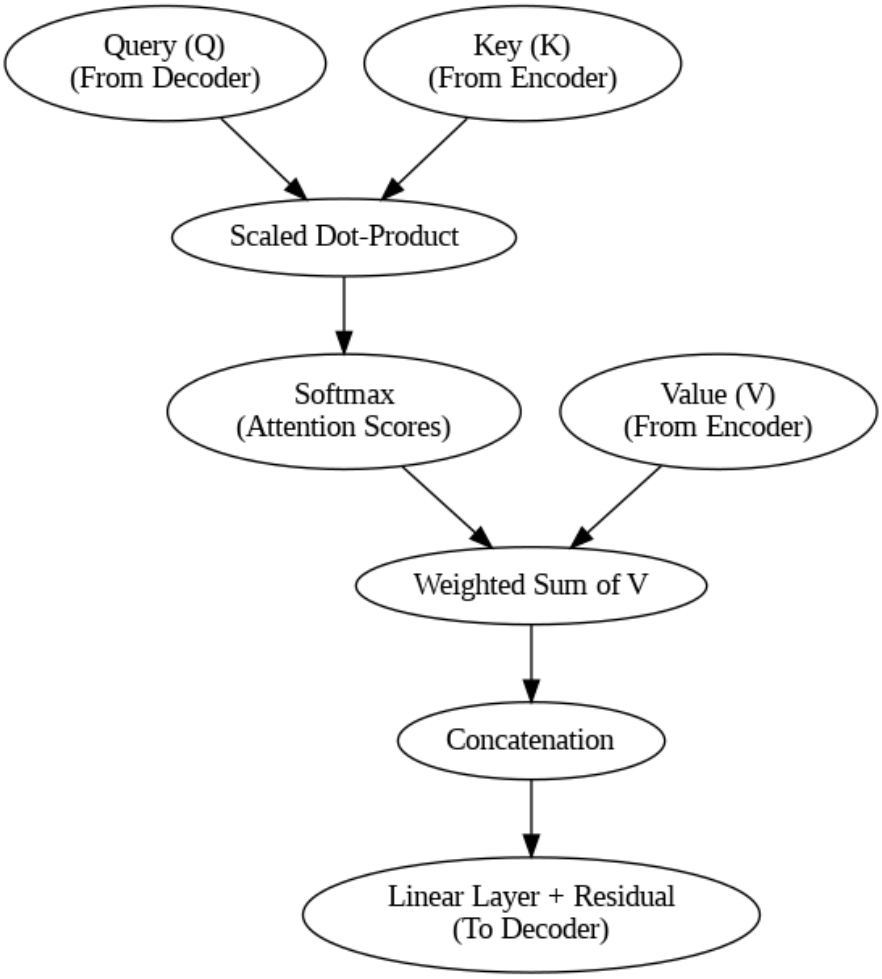


Figure 2.2: Cross Attention Mechanism in Transformers

After the masked self-attention block, the decoder incorporates an encoder-decoder attention layer, which allows the decoder to focus on relevant parts of the encoder's output. This is known as cross-attention, and it enables the decoder to integrate information from the encoder's learned representations as it gener-

ates the output sequence. It operates using three inputs: the Query (Q) vector from the decoder and the Key (K) and Value (V) vectors from the encoder. The decoder's Query is compared to the encoder's Key using a scaled dot-product, and the result is passed through a softmax function to compute attention scores. These scores weigh the corresponding Value vectors, producing a context-aware representation. Multiple attention heads perform this process in parallel, capturing different aspects of the input. The outputs are then concatenated, linearly transformed, and passed through a residual connection, allowing the decoder to generate coherent and contextually relevant outputs.

Like the encoder, the decoder also contains a feed-forward neural network that processes each position independently. This further refines the representation at each position to ensure that the decoder is able to generate a clear and meaningful sequence. The decoder takes the context provided by the encoder and generates an output sequence one token at a time. The self-attention mechanism makes sure that the sequence is generated autoregressively, while the cross-attention mechanism allows the decoder to focus on relevant information from the encoder. Lastly, similar to the encoder, the output is continually being refined by each stacked layer of the decoder, allowing the model to create increasingly complex sequences.

2.2.3 Key Differences Between Encoder and Decoder

The encoder and decoder share many similarities in their structural components, but they differ in their specific mechanisms and their roles within the model. The first difference lies in the attention mechanisms. In the encoder, the self-attention is applied directly to the input sequence, which allows for each token to access all the other tokens in the sequence. In contrast, the decoder uses masked self-attention, which prevents each token from accessing future tokens in the sequence. This ensures that the decoder generates tokens in an autoregressive fashion, generating one token at a time based on previous tokens.

Additionally, the decoder introduces an encoder-decoder attention mechanism,

which allows the decoder to attend to the encoder's output sequence. This cross-attention mechanism helps the decoder retrieve valuable information from the encoder, ensuring that the generated output sequence is grounded in the input sequence. The encoder lacks this cross-attention component, as its role is simply to encode the input sequence into rich representations for further processing by the decoder.

Finally, the purpose of the encoder and decoder is different. The encoder's goal is to encode the input sequence into a continuous representation that captures the semantic relationships and contextual dependencies within the sequence. On the other hand, the decoder uses the encoded information to generate an output sequence by attending to both previously generated tokens and the encoder's output.

2.2.4 Layer Repetition

Both the encoder and the decoder consist of stacked layers, and this repetition of layers allows the model to handle increasingly complex data. The stacking of identical layers helps the model refine and transform the input sequence or output sequence through multiple stages, progressively learning more abstract features. Each additional layer refines the data representations through repeated applications of the attention mechanisms and feed-forward networks, making the model more capable of handling complex, high-dimensional data.

By stacking layers, the model has the capacity to learn increasingly abstract and complex patterns, both in terms of the relationships between tokens in the input sequence in the encoder and the generation of tokens in the output sequence in the decoder. The ability to stack multiple layers and refine the representations at each level is important to the functionality of the transformer architecture, enabling it to perform well on tasks that require understanding long-range dependencies and complex relationships.

2.3 Numerical Data Embeddings

2.3.1 Input Embeddings for Numerical Data

For prediction tasks such as stock price prediction, the input consists of numerical features such as open prices, close prices, high/low prices, and traded volume. These features are continuous by nature and must be embedded into a high-dimensional space before being fed into the transformer.

Given an input sequence of features $x_1, x_2, \ldots, x_n$ at time steps $t_1, t_2, \ldots, t_T$, where x_i is a vector representing the numerical features at a specific time step, the input embeddings are computed as:

$$E(x_i) = W_{\text{embed}} x_i + b_{\text{embed}} \tag{2.1}$$

where W_{embed} is a learnable weight matrix projecting input features to the model's dimension and b_{embed} is a bias term [1].

Each input embedding incorporates temporal position information using positional encodings to capture the sequential nature of stock data. These encodings ensure that time-sensitive relationships like trends and seasonality are preserved. By using learned linear transformations, each numerical feature contributes proportionally to the embedding vector. This approach handles diverse feature ranges and maintains compatibility with self-attention mechanisms.

2.3.2 Positional Encoding

Since the transformer architecture does not inherently encode positional information, positional encodings are added to the input embeddings to provide information about feature order. For a feature at position pos and dimension i, the positional encoding is computed as:

$$\text{PE}_{\text{pos},2i} = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right), \quad \text{PE}_{\text{pos},2i+1} = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{2.2}$$

where, *pos* is the position of the feature in the sequence, i is index of the embedding dimension, and d_{model} is the dimension of the input embedding that corresponds to the number of predicted numerical features [1]. These encodings typically utilize sine and cosine functions of different frequencies so that each position can be uniquely encoded, allowing the model to distinguish between different positions in the sequence without sacrificing its parallel-processing abilities. Once input embeddings are computed, they are passed through the transformer layers, which consist of self-attention and feed-forward sublayers. These layers model both short-term patterns (e.g., intraday volatility) and long-term dependencies (e.g., multi-day trends), making them well-suited for stock prediction tasks.

2.3.3 Output Embeddings for Numerical Data Predictions

The objective in data prediction tasks is to generate numerical outputs, such as the next day's adjusted close price or trading signals (buy, sell, hold) in the case of stock predictions. Unlike token-based tasks, the output embeddings for stock data are continuous and correspond to the feature dimensions of the predicted values. The output embeddings are computed as:

$$y_t = W_{\text{out}} h_t + b_{\text{out}} \tag{2.3}$$

where y_t is the vector of predicted stock features for time step t, W_{out} is the learnable weight matrix mapping hidden representations to the output space, and b_{out} is a bias term.

2.4 Transformer Layers and Functions

2.4.1 Multi-Head Attention

Following the positional encodings done on the input embeddings are the multi-head attention layers. Multi-head attention allows the model to focus on different

parts of the input in parallel across multiple heads. Each attention head operates independently, learning distinct relationships within the data by projecting the input into different subspaces. For example, while one head might focus on understanding syntactic relationships, like subject-verb pairs, another head might capture semantic relationships, such as entity connections. These multiple heads allow the model to gather a richer set of contextual information from each part of the sequence, which are then concatenated and passed through a linear transformation, thereby improving the model's representational power.

For each transformer head, the self-attention mechanism works by first calculating three key vectors: Queries, Keys, and Values [1]. These vectors enable the attention mechanism to analyze relationships in numerical data sequences, which would then allow the model to weigh the importance of past events (e.g., prices, volumes, or market sentiment) when making predictions. The Query (Q) vectors represents the model's request for relevant information about the current state. For instance, it may signify what a stock's current price might need to determine the next time step's price or trend. The Key (K) vectors describe what each historical data point in the sequence has to offer. For example, a spike in trading volume or a major price drop at a previous time step would be encoded into the key vector, capturing the characteristics other time steps may find relevant. Finally, the Value (V) vectors hold the raw information of these time steps, such as the adjusted close stock price, in which the attention mechanism will aggregate once a match with the query is established. These vectors are calculated:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V \tag{2.4}$$

where Q, K, and V represent the query, key, and value matrices respectively derived from the input embeddings X with W_Q, W_K, and W_V being the corresponding weight matrices for Q, K, and V. The attention mechanism then calculates a score matrix:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.5}$$

where d_k is the dimension of the key vectors. The softmax operation, described below, generates a probability distribution that can then be applied to the value vectors, producing an output weighted by the relevance of other tokens.

Once all the attention scores are calculated for each head, the values are concatenated and passed through a final linear layer:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O \tag{2.6}$$

where head_i is the computed attention score for that head and W_O is a final learnable weight matrix that maps the combined output back to the model's expected dimension, maintaining consistency across layers.

2.4.2 Softmax Function

The softmax function converts raw scores into probabilities, ensuring that they add up to one [7]. For a vector $x = [x_1, x_2, \ldots, x_n]$, the softmax of x_i is given by:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.7}$$

For example, if $x = [2, 1, 0]$, the softmax produces $[0.7, 0.2, 0.1]$. The softmax function transforms the input scores into probabilities by emphasizing the largest values while suppressing smaller ones. This is achieved by exponentiating the input scores and then normalizing them. Larger scores are assigned higher probabilities, while smaller scores are reduced and set closer to zero. This dynamic weighing helps the model focus on the most relevant elements of the input based on their relative importance within the given context.

In the context of multi-head attention, the softmax function plays a crucial role in converting the raw attention scores into a normalized probability distribution. After calculating the dot product between the query (Q) and key (K) vectors and scaling the result by the square root of the key dimension ($\sqrt{d_k}$), the softmax function is applied to the resulting score matrix. This operation produces a set

of attention weights, each representing the relevance of a particular token in the input sequence with respect to the current query. These attention weights are then used to weigh the corresponding value (V) vectors, generating the final output of the attention mechanism.

Specifically, softmax ensures that the attention scores are non-negative and sum to one, converting the scores into a probability distribution that indicates the relative importance of each token. The higher values in the score matrix are amplified, leading to higher probabilities for tokens that are considered more relevant, while the smaller values are suppressed, reducing the influence of irrelevant tokens. This capability of softmax allows the model to focus its attention on the most informative parts of the sequence, enhancing its ability to capture long-range dependencies and contextual relationships.

The softmax operation, along with the scaling factor $\frac{1}{\sqrt{d_k}}$, is important for stabilizing the training process of the transformer model. The dot product between the query and key vectors can grow quite large as the dimensionality of the key vectors increases, which in turn can cause the softmax function to saturate and produce very small gradients. This issue, commonly referred to as the vanishing gradient problem, would inhibit the optimization process and affect the model's learning efficiency. The division by the square root of the key vector dimension prevents this by scaling down the dot products, ensuring that the softmax function remains in a non-saturating range and promotes stable gradient flow during training.

In the multi-head attention mechanism, the scaled softmax operation is applied independently to each attention head. Each head computes its own set of attention scores and, after applying the softmax function, attends to different parts of the sequence, enabling the model to focus on various aspects of the data in parallel. The outputs from all heads are then concatenated and projected through a linear transformation, helping the model capture different attention patterns and construct a more nuanced contextual representation of the sequence. This parallel

processing capability makes multi-head attention particularly useful for learning complex dependencies and relationships within the data, further improving the efficiency of the transformer model.

Additionally, the use of softmax allows the model to learn which parts of the input sequence are most relevant at each step, facilitating a dynamic allocation of attention based on the task-specific context. By learning to assign greater attention to tokens with greater relevance, the model is able to emphasize important features and capture complex relationships that might be overlooked. As a result, this attention-based mechanism is key to the model's ability to handle complex input sequences in various natural language processing and machine learning tasks.

2.4.3 Masked Multi-Head Attention

Masked multi-head attention is a specialized variation of the multi-head attention mechanism used in the decoder architecture of transformers. It processes data after positional encodings are applied to the output embeddings, and its main difference from standard multi-head attention is that it enforces causality by masking certain portions of the input sequence during attention computation [1]. In multi-head attention, every data point within a sequence can attend to all other data points. This mechanism is bidirectional, which allows the model to extract global dependencies across the entire sequence. In contrast, masked multi-head attention restricts data points from accessing future positions in the sequence. For tasks like stock price forecasting, this means that each prediction generation step depends solely on preceding data. The restriction aligns the model's predictions with real-world situations where future information is not accessible. For instance, in time-series forecasting, a masked multi-head attention layer ensures that predictions for time t do not consider data from $t + 1$ onwards. In addition, masked multi-head attention introduces some additional steps to apply the mask and handle invalid positions. Although this leads to an increase in computational complexity, this is essential for preserving constraints specific to the task at hand,

especially in predictive systems.

The masking mechanism works similarly to standard multi-head attention but introduces a masking matrix, M, which sets specific positions in the attention scores to $-\infty$ before applying the softmax function. This adjustment helps assign zero probability to future positions in the sequence. The attention score calculation is modified as:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{2.8}$$

where M is a binary matrix in which unmasked positions are set to 0 and masked positions are assigned $-\infty$. Including M ensures that each token only has access to itself and previous tokens in the sequence during training, aligning with the requirements of causal sequence modeling.

2.4.4 Feed-Forward Network Layer

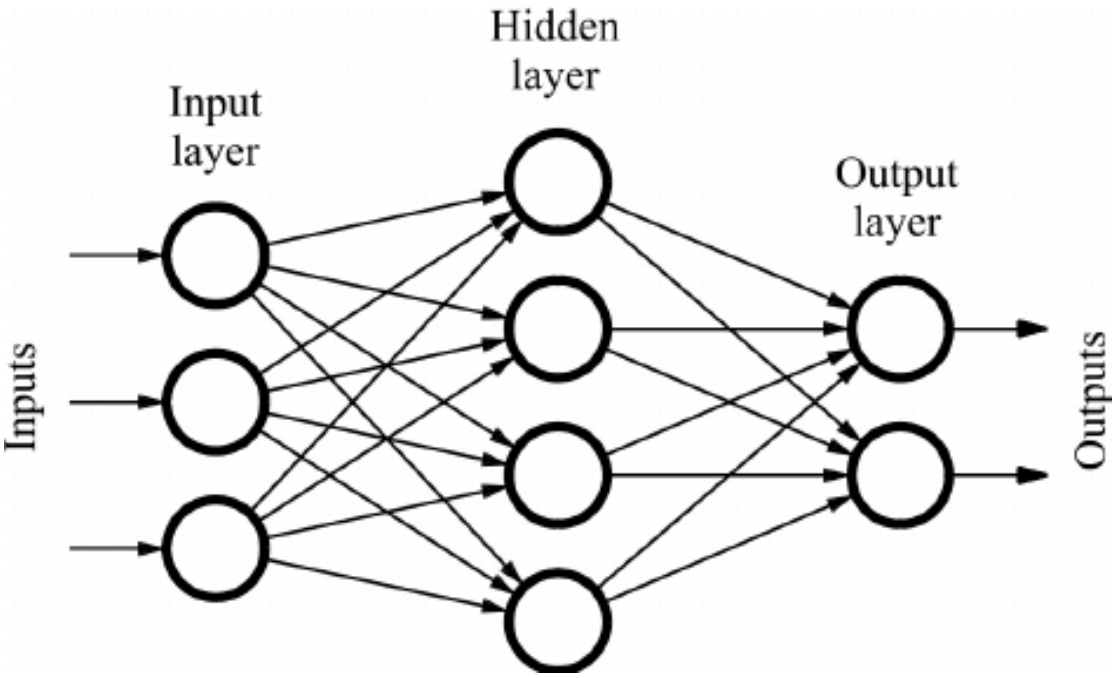


Figure 2.3: Feed-Forward Network Layer Structure [2]

The feed-forward network layers in both the transformer encoder and decoder architectures play a crucial role in learning nonlinear relationships among the fea-

tures of input data. These layers allow the model to learn complex transformations that cannot be captured by simple linear operations, thereby enabling the network to adapt to a wide variety of patterns in the data.

In the context of numerical data, the feed-forward network (FFN) operates on each data point independently. As shown in Figure 2.4, the same transformation is applied to each individual data point in a sequence, regardless of its position. Consequently, while the attention mechanism within the transformer model captures the sequence-level dependencies by considering relationships between all pairs of elements in the sequence, the FFN focuses on independently transforming each data point. This architecture is important because it ensures that the FFN can learn to model local relationships and individual features for each data point while still allowing the attention mechanism to model global relationships between them.

The FFN consists of two linear transformations with a ReLU activation function applied between them, introducing the necessary nonlinearity. This allows the model to learn more complex functions that go beyond the scope of simple linear mappings, increasing the expressive power of the model. The activation function also makes sure that the FFN introduces sparsity and helps to limit issues such as the vanishing gradient problem, which is commonly seen when learning deep neural networks.

Mathematically, the feed-forward network for a sequence of n numerical data points can be expressed as a two-layer operation with a nonlinearity:

$$\mathrm{FFN}(x_i) = W_2 \cdot \max(0, W_1 x_i + b_1) + b_2 \tag{2.9}$$

where x_i is the input vector corresponding to the i-th data point, W_1 and W_2 are learnable weight matrices associated with the two transformations within the network, and b_1 and b_2 are bias vectors used to shift the outputs at each stage [1].

The first operation, $W_1 x_i + b_1$, performs a linear projection of the input x_i into a higher-dimensional space. This transformation allows the network to explore more complex representations of the input data. The ReLU activation, denoted

by $\max(0, \cdot)$, then introduces the nonlinearity required for the model to capture more intricate relationships within the data. ReLU ensures that the network can maintain a balance between learning complex mappings and maintaining computational efficiency by providing an element-wise activation function. The second operation, W_2, performs a final projection to map the intermediate representation back to the original or desired output dimensionality.

The strength of the feed-forward network lies in the composition of these two linear transformations with nonlinearity. While the attention mechanism manages long-range dependencies between elements in the input sequence, the feed-forward network captures rich pointwise transformations, allowing the overall model to learn highly nonlinear mappings that are necessary for tasks requiring complex pattern recognition, such as machine translation, time series prediction, and others. The parallel operation on each data point ensures that the FFN remains computationally efficient while still capable of learning significant relationships.

Moreover, because the FFN is applied uniformly across all positions in the sequence, the architecture remains scalable for sequences of varying lengths. This uniform application, combined with the flexibility of nonlinear transformations, allows the feed-forward layers to capture diverse behaviors and contribute to the network's robust performance in multiple domains. Thus, the combination of attention and feed-forward networks forms the backbone of the transformer model, allowing it to handle a wide array of tasks that require understanding both individual data point features and their global dependencies within a sequence.

2.4.5 ReLU: Rectified Linear Unit

ReLU, which stands for Rectified Linear Unit, is one of the simplest and most widely used activation functions in deep learning. It is mathematically defined as:

$$\text{ReLU}(x) = \max(0, x), \tag{2.10}$$

where x is the input value. The function outputs x itself if $x > 0$ and 0 otherwise.

ReLU serves a critical role in introducing nonlinearity into the model, enabling the network to approximate complex mappings between inputs and outputs [8].

A key property of ReLU is its non-saturating behavior. Unlike other activation functions such as sigmoid or hyperbolic tangent (tanh), ReLU does not limit its output for large positive values, allowing gradients to remain large and stable during training. This characteristic greatly accelerates convergence in optimization algorithms such as stochastic gradient descent.

Another important feature of ReLU is its computational efficiency. Since the function simply thresholds at zero, it is highly efficient to compute, requiring only a comparison and a single multiplication step for implementation. This simplicity is useful when scaling models to handle large datasets or deploying them in real-time systems.

ReLU also introduces sparsity into the network. For negative input values, the output is zero, causing some neurons to be "inactive" during forward and backward passes. Sparse activation has been shown to improve network generalization, as it reduces overfitting by preventing co-adaptation of neurons. This sparsity can also reduce computation time, as fewer neurons need to be updated during training.

Despite these strengths, ReLU is not without limitations. One issue, referred to as the "dying ReLU" problem, occurs when certain neurons output zero for all inputs, effectively stopping learning for those neurons. This can occur due to unfavorable weight initialization or learning rate choices. Variants of ReLU, such as Leaky ReLU and Parametric ReLU, address this issue by allowing small, non-zero gradients for negative inputs, ensuring that neurons remain active during training.

Within the feed-forward network, ReLU serves as a crucial intermediary step between the two linear transformations. After the initial transformation $W_1 x_i + b_1$, the ReLU activation function selectively amplifies positive signals while suppressing negative ones, introducing nonlinearity into the feature space. This selective activation helps the network to capture intricate, non-linear interactions among

features.

By applying ReLU, the FFN helps distinguish overlapping patterns in the data, thereby mapping inputs to a higher-dimensional feature space where separability becomes easier. The second transformation, W_2, then projects these features into the desired dimensionality for further processing into subsequent layers. Therefore, the FFN enables the encoder to capture more abstract and complex patterns, enriching the overall representation of the numerical sequence.

2.4.6 Residual Connections and Layer Normalization

Residual connections and layer normalization are foundational components of the transformer architecture that ensure effective and stable training. Residual connections allow the gradients to flow more smoothly during backpropagation, thereby reducing the vanishing gradient problem. For an input x and a transformation $\mathrm{F}(x)$, the residual connection combines them as:

$$\text{Output} = \mathrm{F}(x) + x \tag{2.11}$$

Layer normalization, on the other hand, normalizes the activations of each layer to ensure stable training [9]. For an input $x = [x_1, x_2, \ldots, x_d]$, the normalized output is computed as:

$$\text{Norm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta \tag{2.12}$$

where μ and σ are the mean and standard deviation of x and are equal to

$$\mu = \frac{1}{d}\sum_{j=1}^{d} x_{ij} \text{ and } \sigma = \sqrt{\frac{1}{d-1}\sum_{j=1}^{d}(x_{ij} - \mu)^2} \tag{2.13}$$

respectively, while γ and β are learnable scale and shift parameters. Layer normalization reduces internal covariate shifts and prevents exploding or vanishing activations, further improving training stability and enhancing generalization.

Chapter 3

VADER For Sentiment Analysis

3.1 Overview of Sentiment Analysis

Sentiment analysis is a text analysis technique that identifies and extracts information from language to determine the emotional tone of a piece of text. It is often used to gauge public opinion and has a wide range of applications, from tracking brand sentiment on social media to analyzing customer reviews [10]. In general, sentiment analysis algorithms classify text as positive, negative, or neutral. Early methods relied on rule-based approaches, in which predefined lists of positive and negative words were used to assess the sentiment. More advanced models use machine learning to learn patterns in language data and make predictions based on labeled training data. The most sophisticated sentiment analysis models today utilize deep learning, often incorporating transformer-based architectures like BERT and GPT. These models have shown impressive results in capturing nuanced sentiment, including sarcasm, which is harder for simpler algorithms to identify.

A common challenge in sentiment analysis involves handling the complexity and ambiguity of human language. For instance, words like "unbelievable" can be either positive ("an unbelievable accomplishment") or negative ("an unbelievable tragedy") based on the context. Sentiment analysis tools must also handle

metaphors, idioms, and slang, which often deviate from literal meanings. In addition, in environments such as social media, users frequently use abbreviations, emojis, and even different spelling variations to convey emotions, making it essential for sentiment analysis models to adapt to these unique language patterns.

3.2 VADER

3.2.1 Overview

One popular sentiment analysis tool is the VADER (Valence-Aware Dictionary and Sentiment Reasoner) model. VADER is a lexicon and rule-based model that was specifically designed for social media sentiment analysis, although it performs well across a variety of text types [11]. Unlike traditional sentiment analysis models that require training data, VADER relies on a pre-built lexicon of words rated by sentiment intensity. It uses a combination of lexicon scores, contextual adjustments for negation and intensity, and sensitivity to punctuation and capitalization to generate sentiment scores that accurately reflect the sentiments expressed in the text. With its efficient calculation process, it is a highly effective tool for sentiment analysis on short informal texts, such as social media posts and product reviews.

3.2.2 Advantages of VADER

VADER's strength lies in its ability to handle the informal language usually found on social media. It includes heuristics for recognizing common intensifiers like capitalization (e.g., "AWESOME!") and punctuation (e.g., many exclamation points or question marks) [11]. Additionally, VADER is able to understand the impact of degree modifiers, which either amplify or reduce the sentiment. For example, words like "very" or "extremely" increase the intensity of the sentiment whereas words like "somewhat" or "partially" decrease it it. The model also recognizes certain emojis and emoticons, which can be important indicators of

sentiment in casual text. These features make VADER very useful for analyzing texts such as comments and reviews where sentiment is often expressed through informal and conversational language.

VADER also remains widely used because of its simplicity and effectiveness in short, opinionated texts. For instance, data such as financial news headers tend to be short, direct, and express strong opinions or sentiments. The concise nature of these headlines makes them ideal candidates for VADER analysis as the tool excels in handling brief and emotional language. In the case of financial news, the headlines often contain certain buzzwords (e.g., "soars", "plummets", "record highs", or "unexpected drops"), making it important to quickly identify sentiment from a small set of words.

Unlike deep learning models that require large datasets and substantial computational resources, VADER also provides accurate sentiment scores without the need for extensive training. In the case of financial headlines, where timely results are necessary, VADER's lexicon-based approach is highly efficient as it uses a dictionary of words that had been manually labeled with sentiment scores instead of relying on training data from scratch. This provides an immediate and interpretable understanding of the headline's sentiment.

Furthermore, the domain-specific nature of financial news means that certain words are likely to be consistent in multiple news articles, which is well suited to the VADER prebuilt lexicon. For example, words like "plummets" and "soars" will always carry a significant negative or positive sentiment, respectively. VADER's ability to quickly compute sentiment scores for each word and adjust the score based on context (including the use of capital letters or punctuation marks) allows it to account for the more nuanced sentiment expressed in text.

In addition, data such as financial headlines often contain relatively straightforward information on stock movements, company earnings reports, or economic news, where emotional intensity is expressed in a few words. For example, the headline "Company X Announces Massive Layoffs" may evoke a strong negative

sentiment that a lexicon-based model like VADER can pick up on immediately. By assigning sentiment scores to words like "massive" and "layoffs," and adjusting them accordingly using VADER's contextual rules, the model can generate an overall negative sentiment score that reflects the public perception of potential financial fallout.

Another advantage of VADER in textual analysis is its ability to handle informal language. For example, many news sites have adapted to a more casual tone in headlines to capture attention quickly, much like other short-form texts such as tweets. Even with short-hand or stylistic writing choices such as the use of emoticons, exclamation points, or all-capitalized words, VADER is capable of adjusting sentiment scores for these stylistic choices. This is very useful for assessing the emotional tone in reports where writers may utilize dramatic language.

Lastly, VADER's rule-based methodology makes it incredibly easy to understand and trust the results it produces. Unlike more opaque machine learning models, the VADER model provides a direct mapping from the sentiment lexicon to final sentiment scores. This makes VADER particularly valuable in environments where quick insights are needed without sacrificing accuracy. Analysts and traders rely on these tools to extract sentiment from news sources to make timely decisions based on the market sentiment expressed in the headlines.

3.2.3 VADER Sentiment Score Calculation

VADER employs a rule-based approach combined with a pre-defined lexicon to determine sentiment polarity. Each word in the lexicon has a pre-assigned score that reflects its sentiment intensity on a scale, typically ranging from -4 (extremely negative) to $+4$ (extremely positive), with 0 representing neutrality [12]. Finally, VADER calculates an overall sentiment score for a text by combining the individual scores of the words, adjusting for factors such as negations, degree modifiers, and punctuation.

At the start of the processing, each word in the input text is looked up in the

lexicon, and its associated sentiment score is retrieved as:

$$S_i = \text{Lexicon Score of Word}_i \tag{3.1}$$

The sentiment of each word in the text is quantified using these pre-calculated scores from the lexicon. Several factors affect the final sentiment score, including negations, degree modifiers, and punctuation. Negating words like "not", "never", or "no" can reverse or weaken the sentiment of the words that follow them. When a negation is detected in the text near a sentiment-bearing word, its polarity is reversed or neutralized. For example, the adjustment for negation can be made as follows:

$$S_{\text{Adjusted}} = -S_i \tag{3.2}$$

This formula reverses the sentiment polarity for words affected by negations.

In addition, words like "very," "extremely," "slightly", or "somewhat" also influence the intensity of sentiment. Some words amplify the sentiment (e.g., "very") while others reduce the intensity (e.g., "slightly"). The sentiment score of a word can be modified by a scalar multiplier based on the degree modifier:

$$S_{\text{Adjusted}} = S_i \times \text{Scalar Multiplier} \tag{3.3}$$

For example, the scalar multiplier for "very" might be 1.2 but only 0.5 for "barely". These multipliers adjust the influence of the word according to its intensity.

Lastly, punctuation can also modify the sentiment of the text. For example, the presence of exclamation marks increases the emotional intensity. In VADER, each exclamation mark typically increases the sentiment intensity by +0.2. Similarly, all capitalized words are boosted by +0.5 to emphasize the sentiment expressed by capitalization.

After adjustments are made, VADER computes the individual components of sentiment: positive, negative, and neutral sentiment. For each sentiment class,

the scores are aggregated as follows:

$$S_{\text{positive}} = \sum_{\text{positive words}} S_{\text{Adjusted, positive}} \tag{3.4}$$

$$S_{\text{negative}} = \sum_{\text{negative words}} |S_{\text{Adjusted, negative}}| \tag{3.5}$$

$$S_{\text{neutral}} = \sum_{\text{neutral words}} S_{\text{Adjusted, neutral}} \tag{3.6}$$

The compound score is then calculated using the adjusted score values that have been previously calculated. It outputs a sentiment metric that ranges from -1 (very negative) to +1 (very positive). The formula to compute the compound score is as follows:

$$S_{\text{compound}} = \frac{\sum_{\text{all words}} S_{\text{Adjusted}}}{\sqrt{\sum_{\text{all words}} S^2_{\text{Adjusted}} + \alpha}} \tag{3.7}$$

where α is a normalization constant (set to 15) used to ensure the final score to be between -1 and 1 [5].

Once the sentiment components are calculated and the compound score is obtained, the result is interpreted as follows:

$$S_{\text{compound}} > 0.05 \quad \text{(positive sentiment)} \tag{3.8}$$

$$-0.05 \leq S_{\text{compound}} \leq 0.05 \quad \text{(neutral sentiment)} \tag{3.9}$$

$$S_{\text{compound}} < -0.05 \quad \text{(negative sentiment)} \tag{3.10}$$

This categorization allows the text to be classified as positive, neutral, or negative based on the compound score.

Chapter 4

Financial Analysis

4.1 Overview

Financial analysis is the process of studying financial data to understand market behavior, identify patterns, and make well-informed decisions. It plays a crucial role in trading and investment strategies, providing tools and methods to analyze trends, manage risks, and optimize returns. At its core, financial analysis helps to uncover how and why prices move in the marketplace and how different factors influence these changes.

One key goal of financial analysis is to interpret price movements and market trends. Markets often display certain behaviors, such as momentum, in which a stock's price continues to move in the same direction for a time, or mean reversion, in which prices return to a normal range after sharp changes. By recognizing these patterns, traders and analysts can create strategies that take advantage of predictable market behaviors.

Modern financial analysis often combines historical data with advanced predictive tools. By looking at historical trends, analysts try to understand how markets behaved in the past and use this knowledge to estimate what might happen in the future. Advancements in machine learning and artificial intelligence have also made it easier to create models that adjust to changing conditions and discover

hidden relationships between different factors in the market. The financial metrics utilized in this project play a major role in financial modeling and data prediction, and include the Open, High, Low, Close (OHLC), Adjusted Close prices, Volume, and the S&P 500 Index.

4.2 Financial Analysis Metrics

4.2.1 OHLC Overview

Open, High, Low, Close (OHLC) data is a widely used format in financial analysis and trading, capturing the critical price points of a stock or asset within a specified time frame [13]. These data points encapsulate the price movement of a financial instrument (e.g., stock, index, or derivative) over a specified period of time. Most often, these are calculated on a daily basis, but they can also represent shorter intervals, such as hourly or minute-by-minute windows. Analyzing OHLC data is fundamental in technical analysis as it makes patterns more discernible to traders who rely on these patterns for forecasting future price movements.

4.2.2 Open Price

The Open price represents the first transaction price of a security during the trading session. It is often computed by stock exchanges at the start of each trading day based on aggregated buy and sell orders received before the official market opening. The Open price acts as an important indicator of market sentiment at the start of trading. Market forces such as after-hours trading, news announcements, and earnings releases significantly affect the opening price, making it a reflection of overnight developments and pre-market activity [13].

4.2.3 High Price

The High price for the day reflects the maximum transaction price recorded for the security during the trading session. It is commonly interpreted as a measure of the highest level of investor willingness to purchase a stock on the given day [13].

4.2.4 Low Price

The Low price represents the minimum transaction price recorded during the trading session. It often serves as an indicator of selling pressure in the market. Like the High price, the Low price provides insight into market volatility and can be used as part of risk assessment models [13].

4.2.5 Close Price

The Close price represents the last transaction price of the stock before the trading session ends. It is one of the most important price metrics in financial analysis, as it encapsulates the final consensus value of the security at the end of trading. Moreover, the Close price is used to calculate daily returns:

$$R_t = \frac{C_t - C_{t-1}}{C_{t-1}} \tag{4.1}$$

where R_t is the daily return, C_t is the Close price for the current day, and C_{t-1} is the Close price for the previous trading day [14]. Although the close price is useful for observing daily price movements, it can be misleading when studying a stock over an extended period since it does not account for certain corporate actions. For example, if a company issues a dividend or undergoes a stock split, the raw close price does not reflect the actual financial impact of these events on an investor's holdings [13].

4.2.6 Adjusted Close Price

The Adjusted Close price is a critical financial metric that provides a more accurate reflection of a stock's true value compared to the traditional closing price. It represents the closing price of a stock, modified to account for corporate actions such as dividends, stock splits, or other adjustments. These actions can have a significant impact on a stock's historical price data, making the adjusted close essential for accurate analysis, especially when evaluating long-term performance or calculating returns. For example, a stock that consistently issues dividends might appear to have peaked in value when examining its traditional close price, as the dividends cause a proportional drop in the stock price on the previous dividend date. However, the adjusted close price accounts for these payouts, showing a more accurate depiction of the stock's overall performance.

When a dividend is paid, the adjusted closing price is calculated by subtracting the dividend amount from the traditional closing price. The formula for adjusting the close price due to a dividend is:

$$P_{\text{adj}} = P_{\text{close}} - D \tag{4.2}$$

where P_{adj} is the Adjusted Close Price, P_{close} is the Traditional Closing Price, and D is the Dividend per share [15].

For a stock split, the adjusted closing price is calculated by dividing the traditional closing price by the split factor, which is the number of shares into which each share is split. The formula for adjusting the close price in the case of a stock split is:

$$P_{\text{adj}} = \frac{P_{\text{close}}}{S} \tag{4.3}$$

where P_{adj} is the Adjusted Close Price, P_{close} is the Traditional Closing Price, and S is the Split Factor (e.g., for a 2-for-1 split, $S = 2$) [15].

In practice, both dividends and stock splits may either occur at the same time

or at different times. The formula for calculating the adjusted closing price when both actions have occurred is:

$$P_{\text{adj}} = \frac{P_{\text{close}} - D}{S} \tag{4.4}$$

where P_{adj} is the Adjusted Close Price, P_{close} is the Traditional Closing Price, D is the Dividend per share, and S is the Split Factor.

4.2.7 Volume

The volume is defined as the total number of shares or contracts traded for a given security during the time period. Volume data reflects market liquidity and is used to gauge investor interest. A high volume relative to historical averages is often seen as an indicator of the importance of a price move. For example, strong price movements accompanied by high trading volume often signal a potential trend shift or a significant market development [13].

4.2.8 The S&P 500 Index

The S&P 500 index is one of the most important benchmarks in global financial markets [16]. It is often used as a proxy for overall market performance and as a benchmark against which individual investment portfolios are evaluated. The value measures the performance of 500 of the largest publicly traded companies in the United States, weighted by their market capitalization. Its calculation involves summing up the market capitalization of each component stock and dividing it by a divisor, which accounts for stock splits and other adjustments:

$$\text{S\&P 500 Index} = \frac{\sum_{i=1}^{500} P_i \cdot S_i}{\text{Divisor}} \tag{4.5}$$

where P_i is the price of stock i, S_i is the number of shares outstanding for stock i, and the Divisor ensures the continuity of the index value over time.

4.2.9 Usage in Predictive Modeling

When building a stock price prediction model, OHLC data combined with technical indicators like volume and S&P 500 indices serve as critical inputs. Together, these indicators help capture complex market dynamics, providing deeper insight into possible price movement. The objective is to identify patterns in these price movements that may help predict future trends, such as bullish (upward) or bearish (downward) behavior in the stock adjusted closing prices.

In a transformer-based stock price prediction model, these financial data features serve as the sequence input, with each time interval treated as a position in the sequence. Embedding layers or other positional encodings are often used to preserve the order of time steps since transformers are inherently order-agnostic. Each point in the sequence (representing an OHLC, volume, and S&P 500 index data vector) is passed through the self-attention mechanism, which allows the model to weigh the importance of previous price data when predicting the next signal. For instance, certain price levels or combinations of highs and lows may signal a potential breakout or reversal, which the model can learn to recognize and weigh accordingly. This capability allows the transformer to detect both short-term fluctuations and long-term trends in stock prices, capturing meaningful relationships in the data that might be missed by simpler models.

In the end, the model will output buy, sell, or hold signals based on the model's interpretation of the OHLC, volume, and S&P 500 indices data. These signals guide whether to enter, exit, or maintain a position in a given stock. The transformer's attention mechanism helps it recognize specific price patterns associated with each action, which might prompt a buy, sell, or hold signal. In a financial application, these signals are particularly powerful when optimized not just for accuracy but also for profitability. In optimizing a cumulative profit function, reinforcement learning or a custom loss function can be implemented to reward sequences of actions that lead to profitable outcomes. Instead of focusing solely on minimizing prediction error, this approach encourages the model to generate

signals that return higher net gains across a series of trades. For instance, instead of simply predicting a price increase or decrease, the model might be trained to evaluate potential trades based on expected return and transaction costs. This profit-focused optimization aligns the model with real-world trading goals, where the ultimate goal is not just to predict price movements accurately but also to generate consistent profits from trading decisions.

Chapter 5

Implementation

5.1 Data Collection and Preprocessing

5.1.1 News Header Data Collection

News analysis headlines were collected from a popular platform for stock market analysis and commentary known as Seeking Alpha. The data was manually acquired and the time frame was set to 2013–2024, creating a comprehensive dataset that covers periods of both market stability and extreme volatility such as the COVID-19 pandemic. The compound sentiment scores were then calculated for each headline using the VADER sentiment analysis tool.

5.1.2 Aggregation of Daily Sentiment Scores

Given that multiple headlines are published for a single stock on most trading days, it was necessary to aggregate the compound sentiment scores calculated from the VADER sentiment analysis tool for each day. This aggregation was accomplished by using the mean:

$$S_{\text{avg}}(t) = \frac{1}{n_t} \sum_{i=1}^{n_t} S_{\text{compound},i} \tag{5.1}$$

where n_t is the total number of headlines on day t, and $S_{\text{compound},i}$ is the sentiment score for headline i.

5.1.3 Historical Stock Price Data Collection

Historical OHLC, Adjusted Close, Volume, and S&P500 Index data were taken from Yahoo Finance for various stocks using the `yfinance` Python library [17]. These data features were aligned with the sentiment scores on a day-to-day basis to construct a unified dataset. In addition, one-hot encoding was done by adding a binary indicator variable column to distinguish between days with missing sentiment data (0) and days with valid sentiment scores (1). This column allowed the model to avoid misinterpreting missing data as neutral sentiment. Figure 5.1 below shows an sample combined dataset used in the project showing the Amazon Historical Stock Price Data from January 1, 2013 to August 30, 2024:

Date	Adj Close	Close	High	Low	Open	Volume	GSPC Adj Close	Score	Encoded
2013-01-02	12.865500	12.865500	12.905000	12.663000	12.804000	65420000	1462.420044	0.39515	1
2013-01-03	12.924000	12.924000	13.044000	12.818500	12.863500	55018000	1459.369995	0.00000	0
2013-01-04	12.957500	12.957500	12.990000	12.832500	12.879000	37484000	1466.469971	0.00000	0
2013-01-07	13.423000	13.423000	13.486500	13.133500	13.148500	98200000	1461.890015	-0.28235	1
2013-01-08	13.319000	13.319000	13.449000	13.178500	13.353500	60214000	1457.150024	-0.10115	1
...	...	...	...	...	...	...	...	...	...
2024-08-26	175.500000	175.500000	177.470001	174.300003	176.699997	22366200	5616.839844	0.55615	1
2024-08-27	173.119995	173.119995	174.889999	172.250000	174.149994	29842000	5625.799805	0.26955	1
2024-08-28	170.800003	170.800003	173.690002	168.919998	173.690002	29045000	5592.180176	0.00000	0
2024-08-29	172.119995	172.119995	174.289993	170.809998	173.220001	26407800	5591.959961	0.00000	0
2024-08-30	178.500000	178.500000	178.899994	172.600006	172.779999	43429400	5648.399902	0.00000	0

Figure 5.1: Combined Amazon Historical Stock Data From 01/01/13 - 08/30/24

5.1.4 Data Transformation and Scaling

The final preprocessing steps ensured that the dataset was ready for ingestion by the machine learning model. First, the data was divided into overlapping sequences of 8 consecutive days (i.e., a time step T = 8). This is known as the sliding window technique, and it allows the model to capture temporal dependencies, providing context for understanding trends and patterns over time [18].

Each sequence contains features for the sentiment scores, binary encoded value for the sentiment scores, OHLC prices, Adjusted Close Prices, and Volumes. This method allows the model to learn relationships across time and features simultaneously and ensures that the dataset is fully utilized while preventing overfitting by introducing slight variations.

Next, the dataset was split into 80% training, 10% validation, and 10% testing sets to ensure a fair evaluation of model performance. Splitting the dataset into training, validation, and testing prepares the model to be trained on most of the data while keeping separate sets for model tuning and performance evaluation. The validation set helps fine-tune hyperparameters and prevents overfitting by ensuring the model generalizes well to unseen data during training whereas the testing set, which is unused until the end, provides an unbiased measure of the model's performance on completely new data. This approach avoids information leakage and ensures a fair evaluation of the model's true capabilities.

Finally, Min-Max scaling was applied to scale all numerical values into the range $[0, 1]$. This is a normalization technique used to rescale numerical values to a fixed range [19]. The transformation ensures all features are comparable by bringing their values to a consistent scale. For a variable X, the scaled value X_{scaled} is computed:

$$X_{\text{scaled}} = \frac{X - \min(X)}{\max(X) - \min(X)} \tag{5.2}$$

This process adjusts each value of X by subtracting the minimum value in the dataset and dividing by the range, which is the difference between the maximum and minimum values. The result ensures that values close to the minimum become near 0, while those close to the maximum approach 1.

Performing Min-Max scaling is beneficial as it leads to faster model convergence for algorithms such as gradient descent that are sensitive to feature magnitudes. It also preserves the linear relationships between values within a feature, thereby maintaining relative distances between data points. In addition, it is helpful for models that require inputs within a fixed range such as neural networks with

bounded activation functions like sigmoid or tanh. However, Min-Max scaling can be sensitive to outliers as extreme values in the data can disproportionately affect the computed minimum and maximum, which would distort the rescaled values.

5.2 Neural Network Architecture

5.2.1 Model Parameters

The Transformer encoder-decoder architecture is the core of this implementation. The architecture begins by applying layer normalization to the input, followed by a multi-head attention mechanism with a specific key dimension, number of heads, and dropout rate. After the attention mechanism, another dropout layer is applied, and the residual connection is added to the input. These layers implement a feed-forward network that introduces non-linearity and ensures dimensional consistency with the input. The residual connections also prevents the vanishing gradient problem and supports effective training for the encoder model.

To optimize the training process, a custom learning rate scheduler is implemented to dynamically adjust the learning rate as training progresses. The learning rate at a given epoch t is computed using the formula:

$$\text{learning\_rate}(t) = \text{initial\_learning\_rate} \times (\text{decay\_factor})^{\lfloor t/\text{step\_size} \rfloor} \tag{5.3}$$

where initial_learning_rate is the starting value of the learning rate, decay_factor determines the reduction rate, and step_size specifies the number of epochs after which decay is applied. The scheduler follows a step decay strategy in which the learning rate is reduced by a fixed factor after a predefined number of epochs. Initially, the learning rate is set to 1×10^{-3}. After every 20 epochs, the learning rate is multiplied by a decay factor of 0.5, essentially halving the value at each step. This stepwise reduction improves model convergence by allowing larger updates in the early training stages when rapid learning is beneficial while ensuring finer

updates in later stages to stabilize training and improve convergence.

In addition, to classify price movements into buy, sell, or hold actions, the implementation evaluates changes in adjusted closing prices. Positive changes trigger a buy signal, negative changes trigger a sell signal, and no changes trigger a hold signal. This logic directly supports the supervised learning framework by generating ground truth labels that represent trading actions.

5.2.2 Custom Loss Function

A key contribution is the custom profit-loss function that incorporates price changes in the stock market. The function is designed to align the predictions of the model with financial outcomes. It calculates the loss as a weighted sum of categorical cross-entropy and the profit term derived from the predicted probabilities of buy, sell, or hold actions. A transaction cost parameter accounts for the financial cost of trading, making sure that the loss function is sensitive to realistic trading conditions. This unique approach encourages the model to make profitable predictions by penalizing losses while promoting effective decision making.

Categorical Cross-Entropy Loss

Categorical Cross-Entropy Loss is a commonly used loss function for multi-class classification tasks. It measures the difference between the true probability distribution of the classes $y_{\text{true},i}$ and the predicted probability distribution $y_{\text{pred},i}$ output by the model [20]. The loss for a single prediction is defined as:

$$\text{Categorical Cross-Entropy} = -\sum_{i=0}^{C-1} y_{\text{true},i} \log(y_{\text{pred},i}) \tag{5.4}$$

where C is the total number of classes, $y_{\text{true},i}$ is a one-hot encoded vector indicating whether class i is the correct class (1 if it is correct, 0 otherwise), and $y_{\text{pred},i}$ is the model's predicted probability for class i. This function ensures the model learns to assign high probabilities to the correct class while penalizing incorrect

predictions. In particular, the logarithm in the formula guides the optimization process by penalizing confident but incorrect predictions more heavily when the predicted probability is low and the true probability is high.

Profit

Profit is the second term in the total loss equation and captures the trading model's performance by evaluating how its predictions align with stock price changes over time. For a single prediction, it is defined as:

$$\text{Profit} = \Delta p_t \left(p_{\text{buy}} - p_{\text{sell}} \right) \tag{5.5}$$

where p_{buy} and p_{sell} are the predicted probabilities for the actions "Buy" and "Sell" respectively, and Δp_t is the price change at time t. This equation quantifies the profit achieved (or loss incurred) based on the predictions of the model, considering that a "Buy" action benefits from positive price changes ($\Delta p_t > 0$) whereas a "Sell" action benefits from negative price changes ($\Delta p_t < 0$).

Total Loss

The Total Loss function then combines the Categorical Cross-Entropy Loss and the Profit, balancing classification accuracy and financial performance. It is defined as:

$$\text{Total Loss} = \alpha \cdot \text{Categorical Cross-Entropy Loss} - (1 - \alpha) \cdot \text{Profit} \tag{5.6}$$

where α is a weight parameter $0 \leq \alpha \leq 1$ that is manually set to control the trade-off between the importance of classification performance (Categorical Cross-Entropy) and profitability (Profit). A higher α places more emphasis on accurately classifying trades, while a lower α prioritizes maximizing profits.

5.2.3 Adam Optimizer Updates

The Adam (Adaptive Moment Estimation) algorithm was used to minimize the custom loss function, which combines the benefits of Momentum and RMSProp, both of which are optimization techniques that adapt the learning rate for each parameter [21]. Momentum takes into account previous gradients to smoothen the optimization process while RMSProp adjusts the learning rate based on the variance of the gradient [21]. Adam improves upon these methods by maintaining both the first moment (mean) and second moment (variance) of the gradients at each step, allowing it to adaptively adjust the learning rate for each parameter. This allows for efficient optimization, especially in complex models with large datasets.

At each step, Adam computes two moment estimates for each parameter. The first moment estimate (m_t) represents an exponentially weighted average of the gradients over time. It obtains the mean gradient at each time step, helping to smooth the updates. The second moment estimate (v_t) is the exponentially decaying average of the squared gradients, which reflects the variance or spread of the gradient. These moments are updated using decay rates β_1 and β_2, usually set to values such as 0.9 and 0.999, respectively. This ensures that the optimization takes into account both recent and past gradient information while controlling the memory of previous gradients.

The update rules for these moments are as follows:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{5.7}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{5.8}$$

where g_t is the gradient at time step t, and m_{t-1} and v_{t-1} are the previous first and second moment estimates, respectively. The parameters of the model are then updated using these moment estimates. The update rule for the parameter θ_t involves the learning rate (η) and the moment estimates (m_t and v_t). The

learning rate is scaled by the square root of the second moment estimate, and the update is further adjusted by the first moment estimate to guide the optimization. The parameter update rule is:

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{5.9}$$

where ϵ is a small constant added to prevent division by zero.

5.2.4 Relative Cumulative Profit Formula

Custom metrics for evaluation include visualizing the strategy's relative cumulative profit or loss over time. The relative cumulative profit is calculated:

$$P_c = \frac{1}{\text{Price}_1} \cdot \sum_{t=1}^{T} \begin{cases} \text{Price}_{t+1} - \text{Price}_t - \text{Transaction Cost}, & \text{if Action}_t = \text{Buy} \\ \text{Price}_t - \text{Price}_{t+1} - \text{Transaction Cost}, & \text{if Action}_t = \text{Sell} \\ 0 & \text{if Action}_t = \text{Hold} \end{cases} \tag{5.10}$$

where P_c is the cumulative profit, Price_1 represents the initial stock price at the beginning of the time period being analyzed, and Transaction Cost is a flat fee (set to 0.01) that is paid for each buy or sell action. This formula was chosen due to its ability to reflect cumulative profit or loss in a way that accounts for both absolute and relative changes in stock price. By applying the decisions of the model over time, the function calculates cumulative gains or losses for the trading strategy, normalized against the initial price. The normalization is crucial because it allows the performance of the trading strategy to be understood as a percentage of the starting price as opposed to just a raw dollar amount. Without this normalization, the raw difference in price may be misleading. For example, an increase of \$10 on a stock priced at \$100 would represent a 10% gain, which is more significant than the same gain on a stock that is priced higher. Therefore, dividing by the initial price provides a fair comparison across different time steps

and different stocks, making the metric more universally interpretable.

5.2.5 Trading Strategy Evaluation

Lastly, the effectiveness of the trading strategy is evaluated by comparing the relative cumulative profit function over time for scenarios with and without sentiment data as input features. In addition, the buy, sell, and hold prediction accuracies are compared to the true labels to evaluate the model's fit in both cases, and are computed:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \times 100 \tag{5.11}$$

These visualizations reveal key performance differences and gives insight into whether the model with sentiment data outperforms the one without.

5.2.6 Processor and Libraries Used

The training was performed on Google Colab's T4 Tensor Processing Unit (TPU), which significantly accelerated the process. With TPU acceleration, the model required approximately 2-3 minutes to complete 40 epochs, whereas training without TPU took longer. The batch size used was 8 to ensure stable training without excessive resource consumption. Training for 40 epochs on TPU took around 6-8 minutes. The final model demonstrated stable convergence, with early stopping mechanisms implemented to prevent overfitting.

Several Python libraries were used in this project, each serving a critical function in data retrieval, preprocessing, model training, and evaluation. TensorFlow (v2.14.0) was the main deep learning framework used for implementing and training the Transformer model. PyTorch (v2.1.0) was initially explored for prototyping, but the final implementation was done in TensorFlow due to its compatibility with TPUs in Colab. YFinance (v0.2.51) was used to fetch historical stock data, with a crucial caveat being the need to explicitly specify `auto_adjust = False`

when calling `yf.download` to get the Adjusted Close Stock Prices. This was necessary since the model was designed to predict adjusted close prices, and using the default setting `auto_adjust = True` would have altered the data in a way that made predictions on adjusted close values impossible.

For sentiment analysis, the VADER lexicon from the NLTK (v3.8.1) package was used to compute sentiment scores from financial news headlines. The SentimentIntensityAnalyzer extension from NLTK's vader lexicon was then used to assess the polarity of each headline, generating sentiment scores that include positive, negative, neutral, and compound sentiment values. Data preprocessing relied heavily on NumPy (v1.26.0) and Pandas (v1.5.3), which were used for feature engineering, normalization, and merging datasets. One particular challenge was handling timestamps, as Pandas occasionally introduced timezone mismatches when merging stock data with sentiment scores, requiring explicit timestamp conversion to maintain data consistency. Matplotlib (v3.8.0) was used to visualize stock trends, sentiment distributions, and model performance. Lastly, Scikit-learn (v1.3.0) provided essential utilities for feature scaling and evaluating model performance using various regression metrics.

Chapter 6

Evaluating the Proposed Method

6.1 Results

This section presents the results of the multi-head transformer model trained to predict buy/sell/hold signals for two stocks from the technology sector (Apple and Amazon) and two stocks from the banking sector (JP Morgan and Goldman Sachs). The model was trained on a combination of stock price data (open, high, low, close, volume, S&P 500 index) and sentiment scores derived from financial news headlines using a VADER sentiment analysis model.

To evaluate the the performance of the model, predictions were compared with and without sentiment analysis. Figure 6.1 below shows the cumulative profit percentage over time for both scenarios for Apple stock data, with the input dataset spanning 5 years from January 1, 2013 to January 1, 2018 and the testing dataset from July 1, 2017 to January 1, 2018.

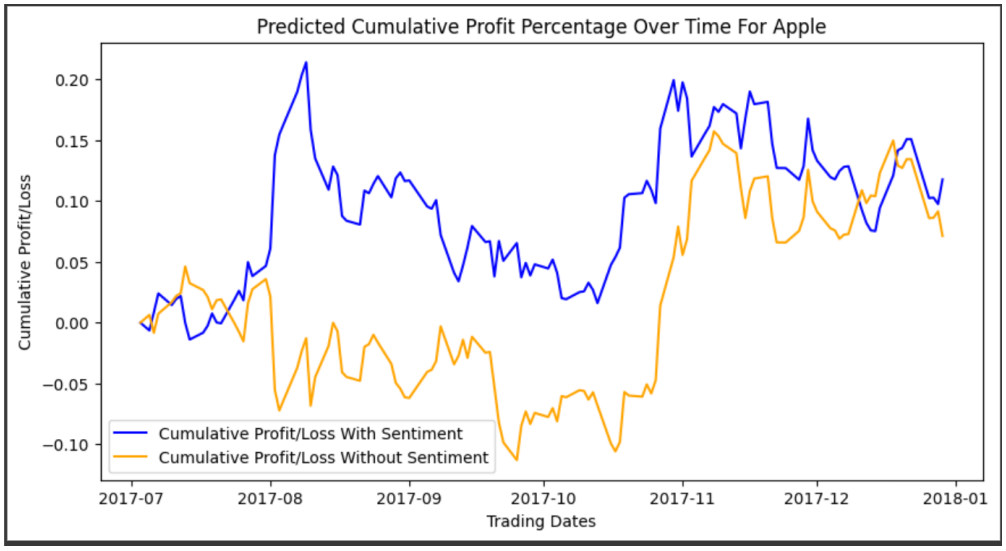


Figure 6.1: Predicted Relative Cumulative Profit Over Time for Apple from July 2017 - January 2018

As shown in Figure 6.1, the model incorporating sentiment analysis generates more profit and outperforms the model without sentiment for most of the time period. Further evidence of the sentiment analysis's positive impact is provided by Figures 6.2 and 6.3, which illustrates the buy (1)/sell (2)/hold (2) predictions for both models for the same time period.

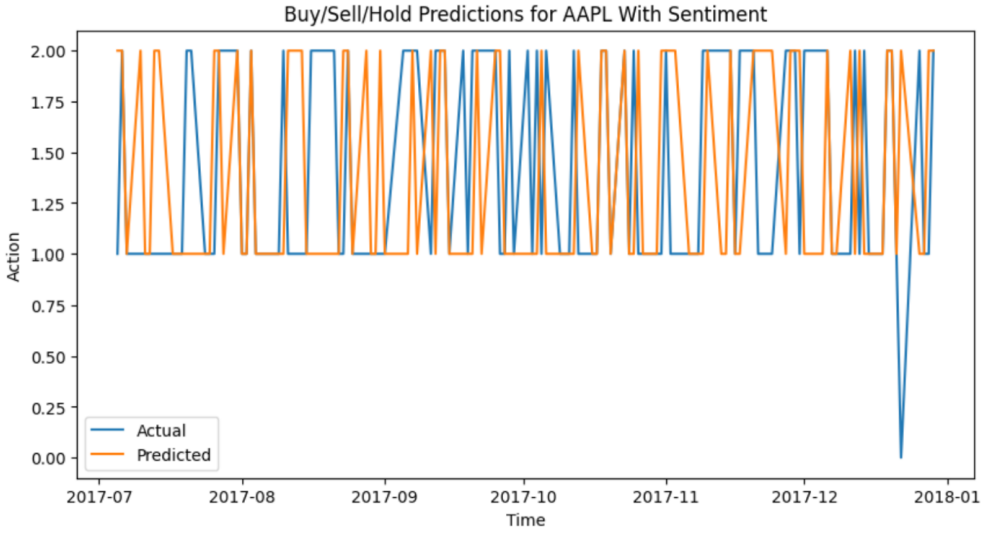


Figure 6.2: Buy/Sell/Hold Predictions for AAPL With Sentiment from July 2017 - January 2018

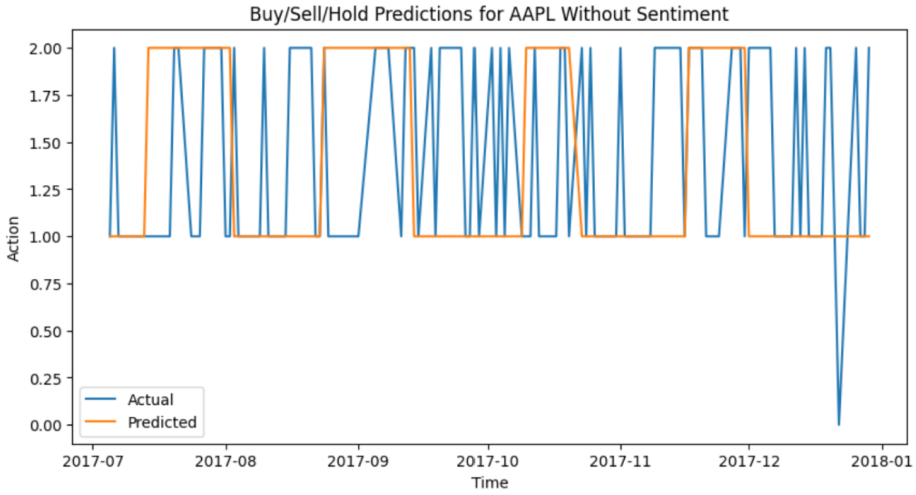


Figure 6.3: Buy/Sell/Hold Predictions for AAPL Without Sentiment from July 2017 - January 2018

The model with sentiment show in Figure 6.2 demonstrates a closer alignment between the predicted and actual buy/sell/hold signals than the model without sentiment shown in Figure 6.3. The predicted values in the "With Sentiment" plot follows the graph of the true labels more closely, suggesting that the model is better able to capture the underlying trends and patterns in the data. Conversely, the model without sentiment is not a good fit for the graph of the true label values. Table 6.1 shown below compares the evaluation metrics for both models.

Metric	**Model with Sentiment**	**Model without Sentiment**
Final Relative Cumulative Return %	11.78%	7.14%
Buy/Sell/Hold Accuracy	0.5550	0.5193

Table 6.1: Model Performance Metrics for Apple Data for Testing Period July 2017 - January 2018

To ensure that the model generalizes to data from different time periods, further experiments were run for both the Apple and Amazon stocks. The chosen

input dataset for both stocks spans a five year interval from September 1, 2018 to September 1, 2023. Correspondingly, the testing dataset for this time period will be from Match 2023 to September 2023. The predicted relative cumulative profit and buy/sell/hold results for Apple are shown below in Figures 6.4, 6.5, 6.6, and the predicted cumulative profit and buy/sell/hold results for Amazon are shown below in Figures 6.7, 6.8, 6.9. Tables 6.2 and 6.3 also contain the model evaluation metrics for both Apple and Amazon for this testing period, respectively.

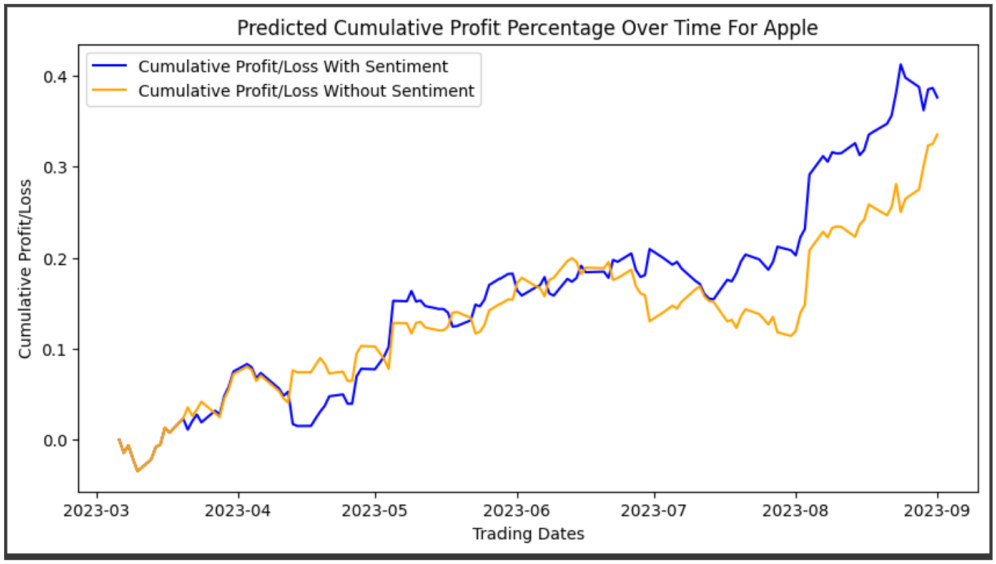


Figure 6.4: Predicted Relative Cumulative Profit Over Time for Apple from March 2023 - September 2023

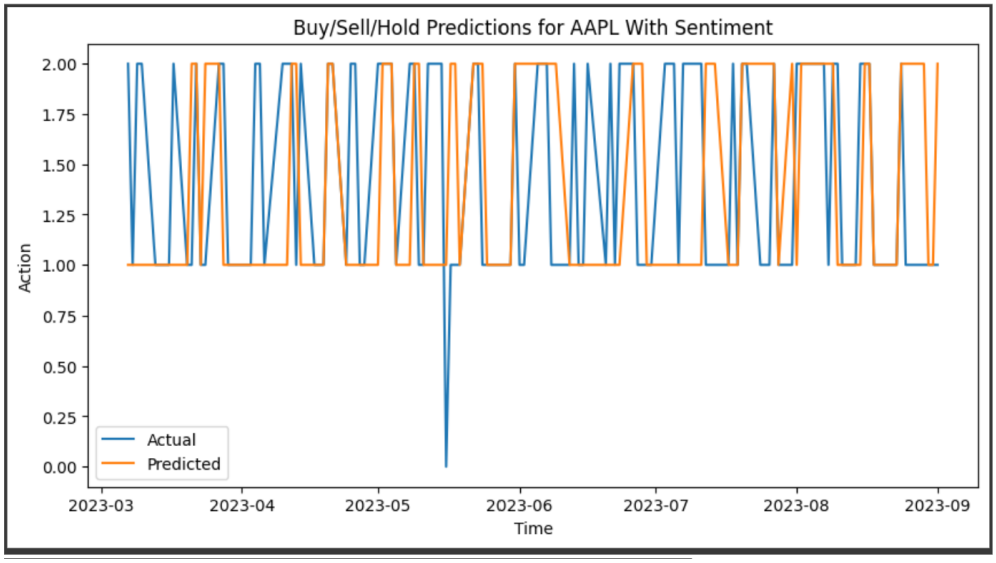


Figure 6.5: Buy/Sell/Hold Predictions for Apple from March 2023 - September 2023 With Sentiment

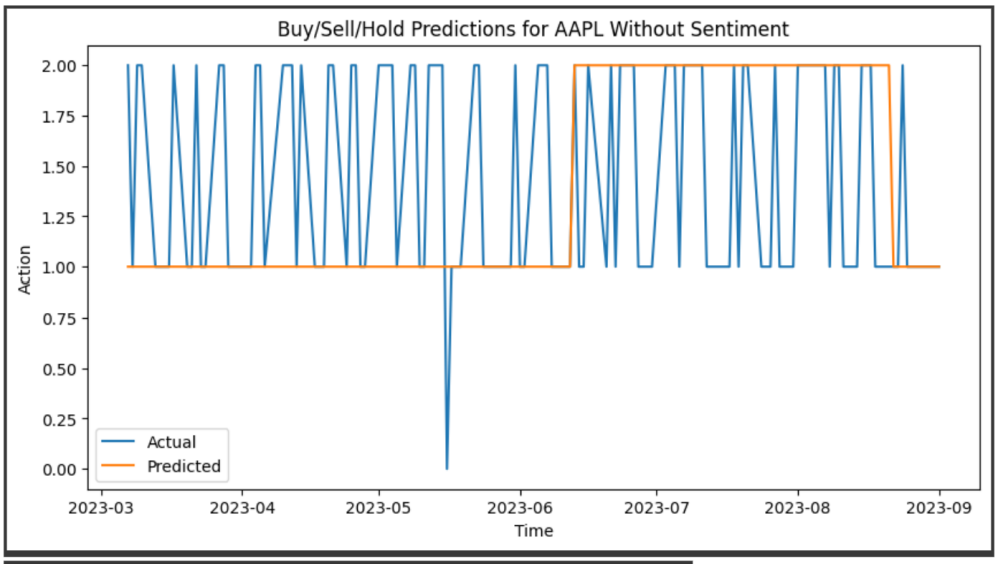


Figure 6.6: Buy/Sell/Hold Predictions for Apple from March 2023 - September 2023 Without Sentiment

Metric	**Model with Sentiment**	**Model without Sentiment**
Final Relative Cumulative Return %	37.62%	33.54%
Buy/Sell/Hold Accuracy	0.5393	0.5698

Table 6.2: Model Performance Metrics for Apple Data for Testing Period March 2023 - September 2023

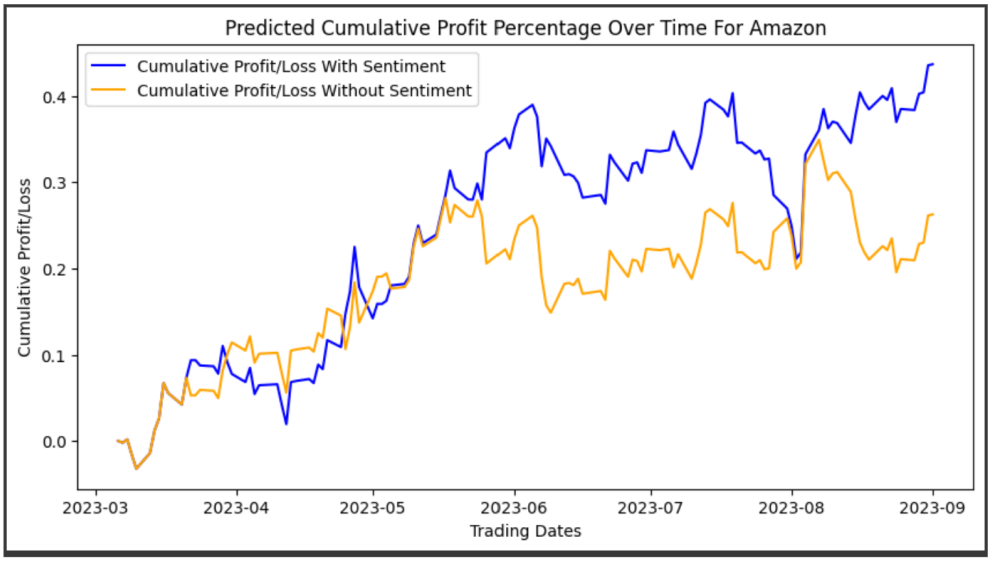


Figure 6.7: Predicted Relative Cumulative Profit Over Time for Amazon from March 2023 - September 2023

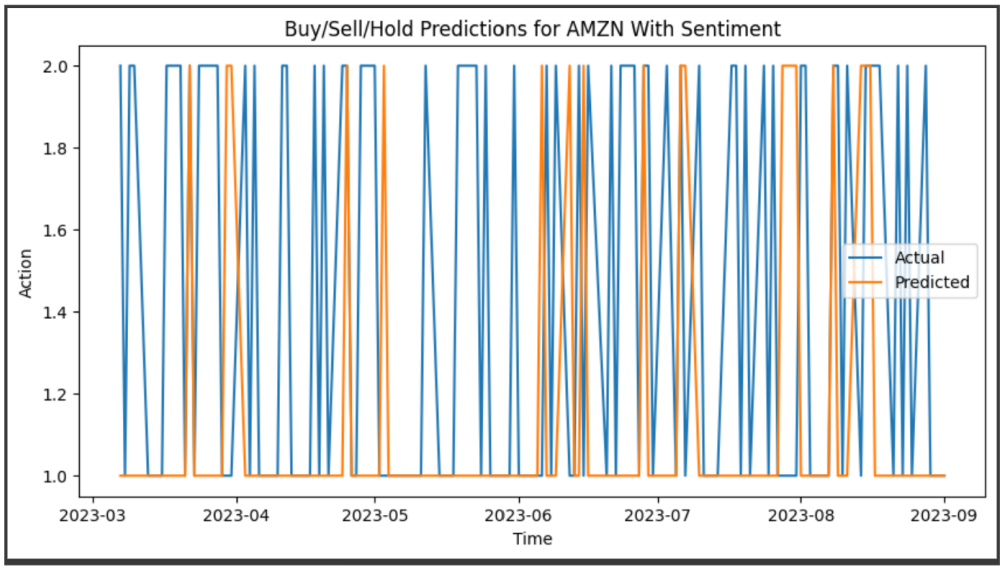


Figure 6.8: Buy/Sell/Hold Predictions for Amazon from March 2023 - September 2023 With Sentiment

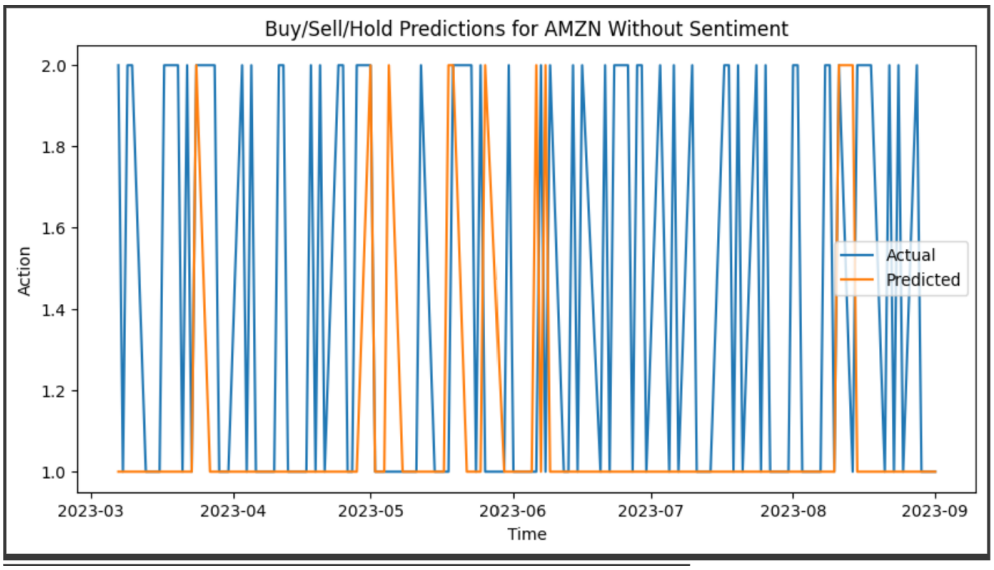


Figure 6.9: Buy/Sell/Hold Predictions for Amazon from March 2023 - September 2023 Without Sentiment

Metric	**Model with Sentiment**	**Model without Sentiment**
Final Relative Cumulative Return %	43.66%	26.25%
Buy/Sell/Hold Accuracy	0.5416	0.5625

Table 6.3: Model Performance Metrics for Amazon Data for Testing Period March 2023 - September 2023

The results in Figures 6.4 and 6.7 for Apple and Amazon, respectively, both show that the model with sentiment generated more profit the model without sentiment. Interestingly, the buy/sell/hold accuracy metrics in both cases slightly favor the model without sentiment. This further signifies the importance of optimizing the model based on a custom loss function that incorporates aspects of both a traditional classification objective function a profit function. By doing so, the model will be guided to predict buy, sell, and hold signals based on profitability.

Next, stock and sentiment data from both Goldman Sachs (GS) and JP Morgan

(JPM) were collected, preprocessed, and used for model training. These experiments are to ensure that the trading strategy performance generalizes to data from a different sector. The first of these experiments involve preparing an input Goldman Sachs dataset that spans a five year interval from January 1, 2013 to January 1, 2018. The corresponding testing dataset for this time period will be from July 2017 to January 2018. The results are shown below in Figures 6.10, 6.11, and 6.12 and the evaluation metrics recorded for this experiment are shown below in Table 3.4.

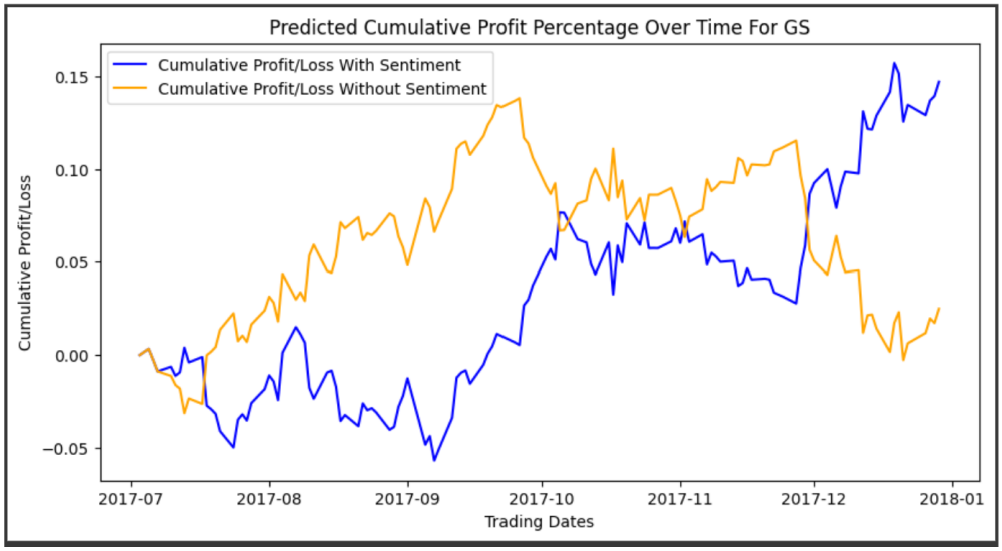


Figure 6.10: Predicted Relative Cumulative Profit Over Time for Goldman Sachs from July 2017 - January 2018

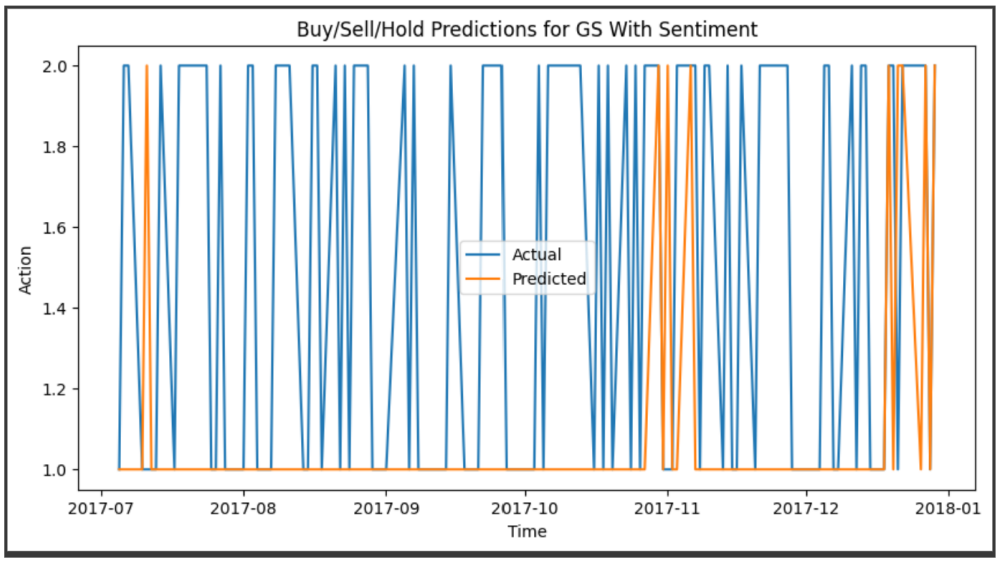


Figure 6.11: Buy/Sell/Hold Predictions for Goldman Sachs from July 2017 - January 2018 With Sentiment

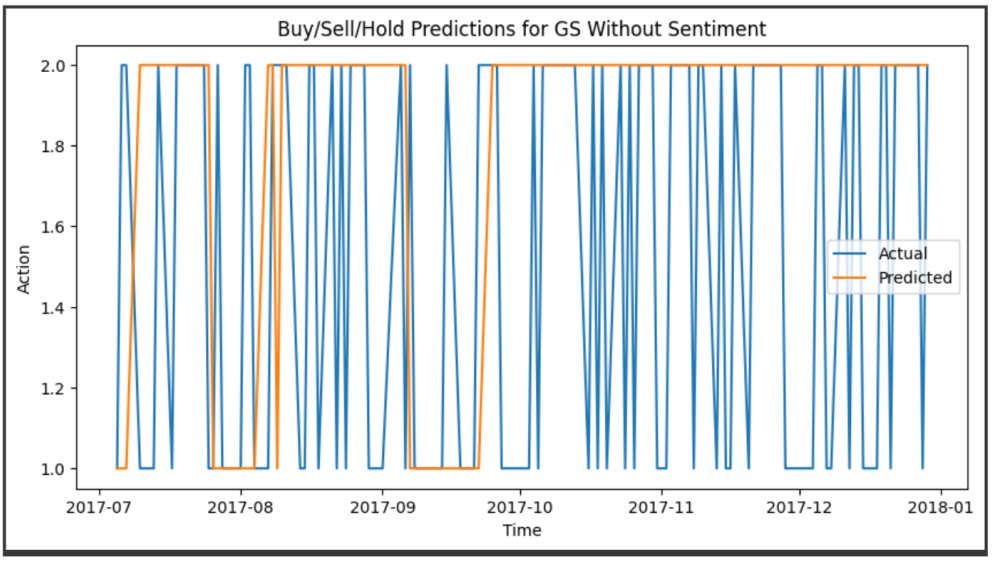


Figure 6.12: Buy/Sell/Hold Predictions for Amazon from July 2017 - January 2018 Without Sentiment

Metric	**Model with Sentiment**	**Model without Sentiment**
Final Relative Cumulative Return %	14.69%	2.49%
Buy/Sell/Hold Accuracy	0.5154	0.5353

Table 6.4: Model Performance Metrics for Goldman Sachs Data for Testing Period July 2017 - January 2018

As shown in Figure 6.10, although the model without sentiment generally outperformed the model with sentiment, the performance of the model without sentiment began to heavily decline right before December 2017, which is also when the model with sentiment started to outperform it and generate more profitable outcomes. This could be because the news header data available for Goldman Sachs from the selected five year interval was very sparse, The lack of sentiment data could have had an adverse effect on the predictive results due to the way in which missing sentiment data is being handled in the data preprocessing steps. Nonetheless, the final relative cumulative return is still considerably higher for the sentiment model than the final relative cumulative return for the model without sentiment. This suggests that effectiveness of this trading strategy is rather delayed as more profitable outcomes tend to occur later in the testing period. In addition, the buy/sell/hold accuracy metrics here slightly favor the model without sentiment as well, which also supports the importance of optimizing the model based on a custom loss function.

Finally, stock and sentiment data from both Goldman Sachs (GS) and JP Morgan (JPM) were taken from a five year interval from November 1, 2015 to November 1, 2020. The purpose of choosing this time period is to evaluate the performance of the trading strategy during the COVID-19 pandemic since the testing period spans from May 2020 to November 2020, which coincides with the

time period in which the effects of the pandemic started to take place in the market. The predicted relative cumulative profit and buy/sell/hold results for Goldman Sachs are shown below in Figures 6.13, 6.14, 6.15, and the predicted cumulative profit and buy/sell/hold results for JP Morgan are shown below in Figures 6.16, 6.17, 6.18. Tables 6.5 and 6.5 also contain the model evaluation metrics for both Apple and Amazon for this testing period, respectively.

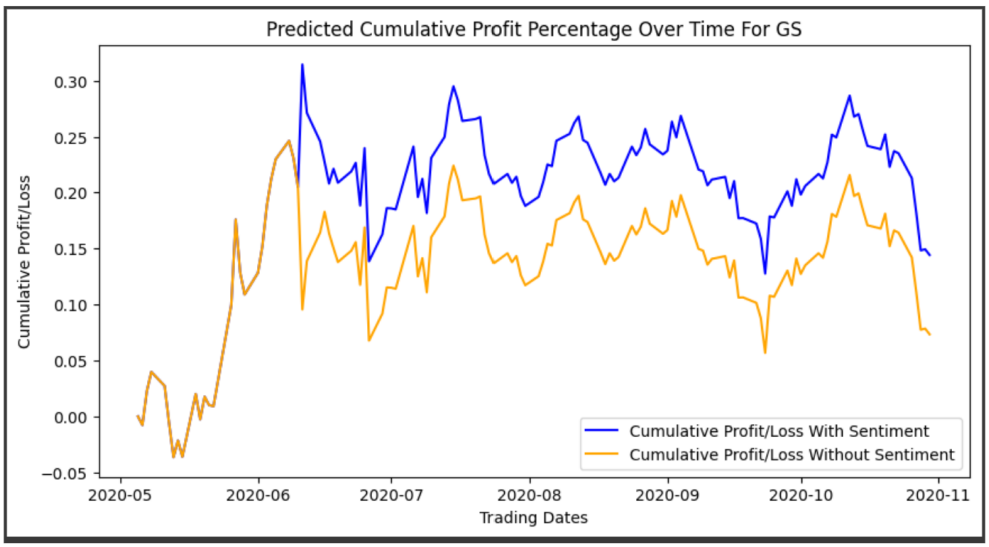


Figure 6.13: Predicted Relative Cumulative Profit Over Time for Goldman Sachs from May 2020 - November 2020

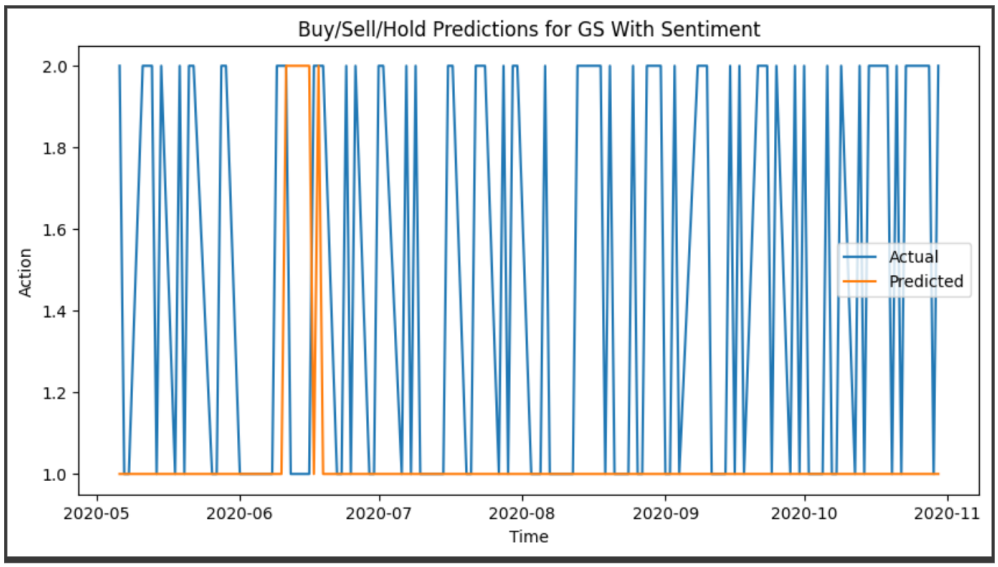


Figure 6.14: Buy/Sell/Hold Predictions for Goldman Sachs from May 2020 - November 2020 With Sentiment

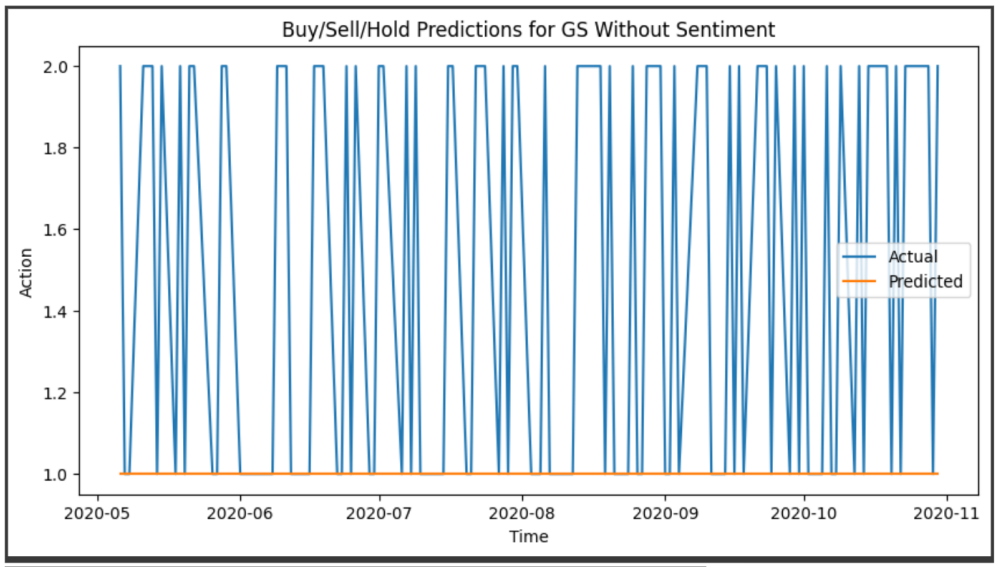


Figure 6.15: Buy/Sell/Hold Predictions for Goldman Sachs from May 2020 - November 2020 Without Sentiment

Metric	**Model with Sentiment**	**Model without Sentiment**
Final Relative Cumulative Return %	14.42%	7.33%
Buy/Sell/Hold Accuracy	0.4942	0.5088

Table 6.5: Model Performance Metrics for Goldman Sachs Data for Testing Period May 2020 - November 2020

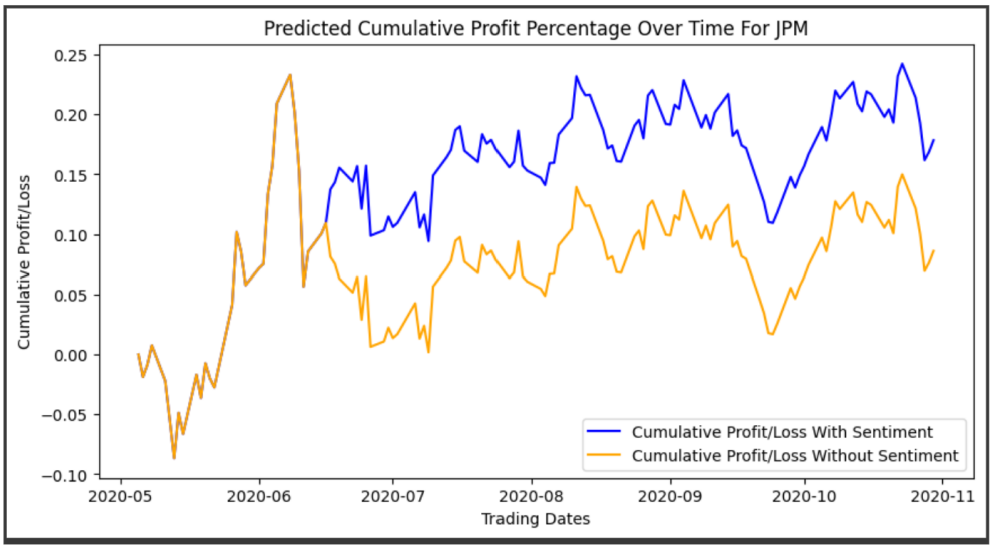


Figure 6.16: Predicted Relative Cumulative Profit Over Time for JP Morgan from May 2020 - November 2020

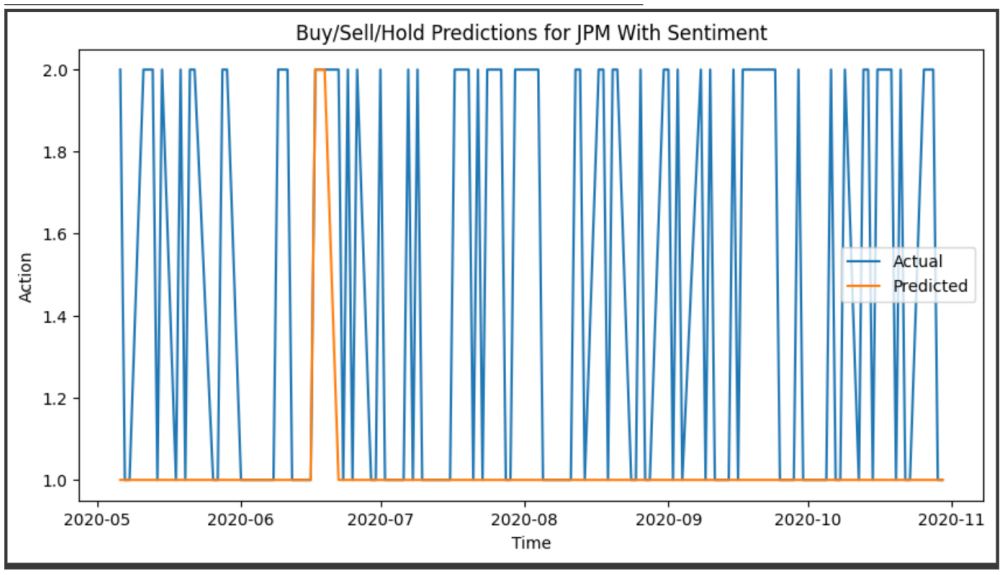


Figure 6.17: Buy/Sell/Hold Predictions for JP Morgan from May 2020 - November 2020 With Sentiment

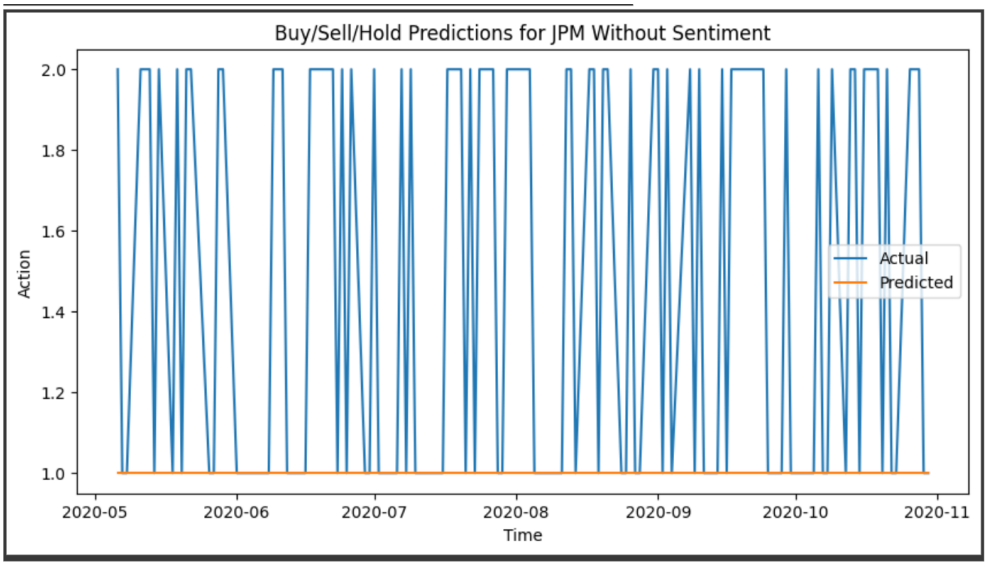


Figure 6.18: Buy/Sell/Hold Predictions for JP Morgan from May 2020 - November 2020 Without Sentiment

Metric	**Model with Sentiment**	**Model without Sentiment**
Final Relative Cumulative Return %	17.84%	8.60%
Buy/Sell/Hold Accuracy	0.5540	0.5100

Table 6.6: Model Performance Metrics for Goldman Sachs Data for Testing Period May 2020 - November 2020

As shown by the results in Figures 6.13 and 6.16 for Goldman Sachs and JP Morgan, respectively, both show that the model with sentiment generated more profit the model without sentiment. The model was able to still generate profitable outcomes for both sets with and without sentiment data and is still comparatively more profitable with sentiment added. However, it is worth noting that the model predicts to only buy (1) for the entire testing dataset for the case in which sentiment data was not passed in as a training feature. This further implicates the importance of having sentiment data, even though the sentiment data were sparse

for these two stocks during this five year span as well. Furthermore, it proves that the model is able to help generate profitable outcomes during a period of global crisis in which markets are being tremendously affected.

Chapter 7

Conclusion

This project focused on creating and testing a multi-head transformer model to predict buy, sell, or hold signals for stocks in the technology and banking sectors. By combining traditional stock price data with sentiment scores from VADER's analysis of news headlines, the study aimed to understand the effect of sentiment on trading performance. The results showed that the use of sentiment data improved overall profitability and worked well across different types of stocks and sectors.

Adding sentiment to the models consistently led to higher profits, especially for stocks such as Apple and Amazon. Although the models with sentiment data as an added input did not always have the highest accuracy in predicting buy, sell, or hold decisions, they still performed better at making profitable trading decisions. This suggests that in trading, focusing on profits rather than just prediction accuracy is more important.

The models also worked well in the banking sector for stocks like Goldman Sachs and JP Morgan, which shows that combining sentiment data with stock predictions is a strong approach that can be used across different industries. It shows the potential of sentiment analysis as a tool to better understand market behavior and improve trading outcomes.

These findings highlight a few ways that future work can build on this. For

example, exploring different custom loss functions that focus on maximizing profits could improve the model's effectiveness. Expanding sentiment data to include texts from sources such as social media could also help increase prediction accuracy.

In conclusion, this research shows that including sentiment analysis in stock prediction models can enhance trading performance by quite a bit. With further improvements, this approach could definitely provide greater value for trading strategies in the future.

Bibliography

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *ArXiv e-prints*, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[2] Ezako, "A basic introduction to neural networks," Ezako Website, 2025, accessed: 2025-01-05. [Online]. Available: https://ezako.com/en/basic-introduction-neural-networks/

[3] R. Xiao, Y. Feng, L. Yan, and Y. Ma, "Predict stock prices with arima and lstm," *arXiv*, vol. 2209.02407, August 2022, arXiv preprint. [Online]. Available: https://doi.org/10.48550/arXiv.2209.02407

[4] C. B. Vennerød, A. Kjærran, and E. S. Bugge, "Long short-term memory rnn," *arXiv*, vol. 2105.06756, May 2021, arXiv preprint. [Online]. Available: https://doi.org/10.48550/arXiv.2105.06756

[5] N. Swarnkar, "Vader sentiment analysis: A complete guide, algo trading and more," https://www.sentimenttrading.com/vader-sentiment-analysis, 2020, accessed: 2025-01-10.

[6] A. Ashraf, "Tokenization in nlp: All you need to know," October 2023, accessed: 2024-01-22. [Online]. Available: https://medium.com/@abdallahashraf90x/tokenization-in-nlp-all-you-need-to-know-45c00cfa2df7

[7] B. Gao and L. Pavel, "On the properties of the softmax function with application in game theory and reinforcement learning," *arXiv*,

vol. 1704.00805, April 2017, arXiv preprint. [Online]. Available: https://doi.org/10.48550/arXiv.1704.00805

[8] J. He, L. Li, J. Xu, and C. Zheng, "Relu deep neural networks and linear finite elements," *Journal of Computational Mathematics*, vol. 38, no. 3, pp. 502–527, 2020, preprint available at arXiv:1807.03973. [Online]. Available: https://doi.org/10.48550/arXiv.1807.03973

[9] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv:1607.06450 [stat.ML]*, 2016. [Online]. Available: https://doi.org/10.48550/arXiv.1607.06450

[10] W. Medhat, A. Hassan, and H. Korashy, "Sentiment analysis algorithms and applications: A survey," *Ain Shams Engineering Journal*, vol. 5, no. 4, pp. 1093–1113, 2014, under a Creative Commons license, open access. [Online]. Available: https://doi.org/10.1016/j.asej.2014.04.011

[11] D. C. Youvan, "Understanding sentiment analysis with vader: A comprehensive overview and application," jun 2024, sentiment analysis using VADER for academic paper titles and other forms of text. [Online]. Available: https://www.researchgate.net/publication/381650914_Understanding_Sentiment_Analysis_with_VADER_A_Comprehensive_Overview_and_Application

[12] C. Hutto and E. Gilbert, "VADER: A parsimonious rule-based model for sentiment analysis of social media text," in *Proceedings of the 8th International Conference on Weblogs and Social Media (ICWSM-14)*, Ann Arbor, MI, USA, June 2014. [Online]. Available: https://doi.org/10.1609/icwsm.v8i1.14550

[13] L. Smigel, "What is open high low close in stocks?" https://www.analyzingalpha.com/blog/what-is-open-high-low-close-in-stocks, 2023, accessed: 2025-01-10.

[14] W. Kenton, "Rate of return (ror): Meaning, formula, and examples," https://www.investopedia.com/terms/r/rateofreturn.asp, Jul. 2024, accessed: 2025-01-10.

[15] K. Balasubramaniam, "How to calculate a stock's adjusted closing price," https://www.investopedia.com/terms/a/adjusted-closing-price.asp, May 2024, accessed: 2025-01-22.

[16] A. Z. Robertson, "The (mis)uses of the s&p 500," *The University of Chicago Business Law Review*, vol. 2.1, 2024, accessed: 2025-01-22. [Online]. Available: https://www.uchicagolawreview.org

[17] G. Bland, "yfinance library – a complete guide," Online, Nov 2020, accessed: 2024-03-01. [Online]. Available: https://algotrading101.com/learn/yfinance-guide/

[18] D. Overload, "Sliding window technique — reduce the complexity of your algorithm," *Data Overload*, 2022, accessed: 2025-01-22. [Online]. Available: https://medium.com/@dataoverload/sliding-window-technique-reduce-the-complexity-of-your-algorithm-cf243223129b

[19] H. Pelletier, "Data scaling 101: Standardization and min-max scaling explained," *Towards Data Science*, Aug 2024, accessed: 2025-01-22. [Online]. Available: https://towardsdatascience.com/data-scaling-101-standardization-and-min-max-scaling-explained-bb567c71b1fc

[20] J. R. Terven, D. M. Cordova-Esparza, A. Ramirez-Pedraza, E. A. Chavez-Urbiola, and J. A. Romero-Gonzalez, "Loss functions and metrics in deep learning," *arXiv preprint arXiv:2307.02694*, 2024, accessed: 2025-01-22. [Online]. Available: https://arxiv.org/pdf/2307.02694

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

Appendix A

Code Appendix

This appendix includes the code used to obtain the results displayed in this thesis for the Amazon stock prices. The code structure remains the same across all test cases, with only minor changes, such as updating the stock name, sentiment dataset, or time period required to perform analysis for each case.

```
import pandas as pd

df = pd.read_csv("amzn-news.csv", header = None)
df2 = df.iloc[::2]
df2.reset_index(drop = True, inplace = True)
df3 = df2.drop(columns = [1])

for index, row in df3.iterrows():
  if row[2].startswith("May") and len(row[2]) == 13:
    df3.at[index, 2] = row[2][:12]


def convert_to_datetime(df3):
    try:
      return pd.to_datetime(row[2], format='%b. %d, %Y').date()
    except ValueError:
      return None

for index, row in df3.iterrows():
```

```
    converted_date = convert_to_datetime(df3)

    if converted_date is not None:
        df3.at[index, 2] = converted_date

df4 = df3
df5 = df4

import datetime as datetime

for index, row in df4.iterrows():
  if isinstance(row[2], str):
    df5.drop(index, inplace=True)

df6 = df5.reindex(columns= [2, 0])
df7 = df6.rename(columns={2: "Date", 0: "News Header"})
df7.set_index("Date", inplace = True)
df7

out = df7.to_csv('out.csv', index = True)

from tqdm import tqdm
import nltk
nltk.download('vader_lexicon')
import pandas as pd
df7 = pd.read_csv("out.csv")
df7 = df7.set_index('Date')
df7.index = pd.to_datetime(df7.index)
from nltk.sentiment.vader import SentimentIntensityAnalyzer as SIA
results = []
for headline in df7["News Header"]:
    pol_score = SIA().polarity_scores(headline) # Runs VADER
        analysis
    pol_score['headline'] = headline # Add headlines for viewing
    results.append(pol_score)
```

```
54
55 results
56
57 comp = pd.DataFrame(results)['compound'].to_list()
58
59 df7["Score"] = comp
60
61 df7
62
63 df8 = df7.groupby(["Date"]).mean(numeric_only=True)
64 df8['Encoded'] = 1
65 df8.drop(columns = ['Encoded'])
66 df8
67
68 #Installing the yfinance library and importing other necessary
      libraries
69 !pip install yfinance
70 import yfinance as yf
71 import pandas as pd
72 import numpy as np
73 import matplotlib.pyplot as plt
74 import tensorflow as tf
75 from tensorflow import keras
76 from keras import layers
77 from keras import backend as K
78 from keras.layers import Conv1D, Flatten, MaxPooling1D, LSTM,
      TimeDistributed, Dense
79 from keras.models import Sequential
80 from keras.optimizers import Adam
81
82 #Pulling down stock data from the yfinance library
83 def get_stock_data(stock_name, start_date, end_date):
84   print(str(stock_name) + " Stock Data:\n")
85   stock_df = yf.download(stock_name, start_date, end_date,
        auto_adjust=False).dropna()
```

```
86
87    stock_df.index = stock_df.index.date
88    stock_df.index = pd.to_datetime(stock_df.index)
89
90    stock_df.columns = stock_df.columns.map('_'.join)
91    stock_df.index.name = 'Date'
92
93    stock_df.columns = stock_df.columns.str.split('_').str[0]
94
95    display(stock_df)
96    print("\nTotal Number of Trading Days: ", len(stock_df))
97    plt.plot(stock_df['Adj Close'])
98    plt.title("Adjusted Close Stock Prices from 2013-2023 for " +
          str(stock_name))
99    plt.xlabel("Time")
100   plt.ylabel("Stock Prices")
101   plt.show()
102
103   return stock_df
104
105 GSPC_stock_df = get_stock_data(stock_name = '^GSPC', start_date =
      '2013-01-01', end_date = '2024-08-31')
106
107 AMZN_stock_df = get_stock_data(stock_name = 'AMZN', start_date = '
      2013-01-01', end_date = '2024-08-31')
108
109 merged_df = JPM_stock_df.merge(df9, how = 'left', left_index =
      True, right_index = True)
110
111 merged_df["Score"].fillna(0, inplace = True)
112
113 merged_df['Encoded'] = encoded
114
115 merged_df.loc[merged_df['Score'] == 0, 'Encoded'] = 0
116
```

```
merged_df

GSPC_df = GSPC_stock_df.drop(['Open', 'High', 'Low', 'Close', '
    Volume'], axis = 1)
GSPC_df.rename(columns={'Adj Close': 'GSPC Adj Close'}, inplace=
    True)


merged_df1 = merged_df.merge(GSPC_df, how = 'left', left_index =
    True, right_index = True)
merged_df1 = merged_df1[["Adj Close", "Close", "High", "Low", "
    Open", "Volume", "GSPC Adj Close", "Score", "Encoded"]]

merged_df1

out3 = merged_df1.to_csv('out3.csv', index = True)

out3 = pd.read_csv("out3.csv")
out3.set_index('Date', inplace = True)
out3.index = pd.to_datetime(out3.index)
out3

out4 = out3[0:1259] #Extracting dataset from Jan. 2013 - Jan. 2017
     (With Sentiment Scores and Hot-One Encoded Value)
out4

out5 = out3
out6 = out5[out5.columns[:-2]] #Extracting dataset from Jan. 2013
    - Jan. 2017 (Without Sentiment Scores and Hot-One Encoded
    Value by removing the last two columns)
out7 = out6[0:1259]
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

```
145 import matplotlib.pyplot as plt
146 from keras.losses import SparseCategoricalCrossentropy
147 from tensorflow.keras.optimizers import RMSprop
148 from sklearn.preprocessing import MinMaxScaler
149
150 # Defining the Transformer Encoder Architecture
151 def transformer_encoder(inputs, head_size, num_heads, ff_dim,
        dropout):
152     # Normalization and Multi-Head Attention Layers
153     x = layers.LayerNormalization(epsilon=1e-6)(inputs)
154     x = layers.MultiHeadAttention(key_dim=head_size, num_heads=
            num_heads, dropout=dropout)(x, x)
155     x = layers.Dropout(dropout)(x)
156     res = x + inputs
157
158     return res
159
160 # Decaying the Learning Rate over time
161 def LR_scheduler(epoch):
162     initial_learning_rate = 0.001
163     decay_factor = 0.5
164     step_size = 20
165     learning_rate = initial_learning_rate * (decay_factor ** np.
            floor(epoch / step_size))
166     return learning_rate
167
168 #Determining Buy/Sell/Hold Action Labels
169 def classify_action(price_change):
170     if price_change > 0:
171         return 1  # Buy
172     elif price_change < 0:
173         return 2  # Sell
174     else:
175         return 0  # Hold
176
```

```
177 #Defining the Custom Loss Function
178 def custom_profit_loss(close_prices_changes_n, train_set_length,
        batch_size, transaction_cost=0.01, alpha=0.3):
179
180     # Ensure close_prices_changes is a Tensor
181     close_prices_changes = tf.convert_to_tensor(
            close_prices_changes_n, dtype=tf.float32)
182
183     def loss_function(y_true, y_pred):
184
185         batch_size_y_true = tf.shape(y_true)[0]  # Number of
                samples in the current batch
186
187         # Generate indices for the current batch items
188         batch_indices = tf.range(batch_size_y_true)
189
190         # Calculate the sliding window or shift the batch index to
                retrieve a new price change
191         shifted_indices = batch_indices + tf.expand_dims(tf.range(
                train_set_length), axis=-1)
192
193         # Ensure no out-of-range issues
194         shifted_indices = tf.clip_by_value(shifted_indices, 0,
                train_set_length - 1)
195
196         # Gather the corresponding price changes for each batch
                item (over time)
197         price_change_batch = tf.gather(close_prices_changes,
                shifted_indices, axis=0)
198
199         # Categorical Cross-Entropy Loss
200         categorical_crossentropy = keras.losses.
                categorical_crossentropy(y_true, y_pred)
201
202         hold_prob = y_pred[:, 0]
```

```
203         buy_prob = y_pred[:, 1]
204         sell_prob = y_pred[:, 2]
205 
206         # Profit Loss
207         profit = buy_prob * price_change_batch - sell_prob *
                price_change_batch
208 
209         profit_loss = tf.reduce_mean(profit)
210         categorical_loss = tf.reduce_mean(categorical_crossentropy
                )
211 
212         # Total loss
213         total_loss = alpha * categorical_loss - (1 - alpha) *
                profit_loss
214 
215         return total_loss
216 
217     return loss_function
218 
219 # Plot cumulative profit/loss over time
220 def plot_cumulative_profit_loss(dates, actual_prices,
    strategy_returns, stock_name, sentiment):
221     cumulative_strategy_returns = np.cumsum(strategy_returns)/
            actual_prices.iloc[0]
222     plt.figure(figsize=(10, 5))
223     plt.plot(dates, cumulative_strategy_returns, label="Strategy
            Profit/Loss", color='b')
224     if sentiment == True:
225       plt.title('Cumulative Profit/Loss Over Time For ' + str(
              stock_name) + ' With Sentiment')
226     else:
227       plt.title('Cumulative Profit/Loss Over Time For ' + str(
              stock_name) + ' Without Sentiment')
228     plt.xlabel('Time')
229     plt.ylabel('Cumulative Profit/Loss')
```

```
230     plt.legend()
231     plt.show()
232     final_strategy_return = cumulative_strategy_returns[-1]
233     print("Final Cumulative Strategy Return:",
            final_strategy_return)
234     return dates, cumulative_strategy_returns
235
236 # Preprocessing data for buy/sell/hold classification
237 def preprocess_data(stock_df, num_features):
238     timesteps = 8
239     X = []
240     Y = []
241     price_changes = []
242
243     stock_df1 = stock_df.drop(['Adj Close'], axis=1)
244
245     # Min-Max scaler for numerical features
246     scaler = MinMaxScaler(feature_range=(0, 1))
247
248     for i in range(timesteps, len(stock_df) - 1):
249         X.append(stock_df1.iloc[i-timesteps:i, :])
250         price_change = (stock_df['Adj Close'].iloc[i + 1] -
                stock_df['Adj Close'].iloc[i])
251         price_changes.append(price_change)
252         Y.append(classify_action(price_change/stock_df['Adj Close'
                ].iloc[i]))
253
254     X, Y = np.array(X), np.array(Y)
255
256     # Splitting the data into training, validation, and testing
            sets
257     split_fraction = 0.9
258     split_index = int(len(X) * split_fraction)
259
260     train_X, test_X = X[:split_index], X[split_index:]
```

```
261     train_label, test_label = Y[:split_index], Y[split_index:]
262
263     original_train_size = len(train_X)
264     new_train_size = len(train_X) - len(test_X)
265
266     val_X, val_label = train_X[new_train_size:original_train_size
            ], train_label[new_train_size:original_train_size]
267     train_X, train_label = train_X[0:new_train_size], train_label
            [0:new_train_size]
268
269     train_X, val_X, test_X = np.array(train_X), np.array(val_X),
            np.array(test_X)
270
271     # Apply min-max scaling to features (excluding 'Encoded')
272     for i in range(train_X.shape[2]):  # Loop over features
273         if 'Encoded' not in stock_df.columns or stock_df.columns[i
                ] != 'Encoded':
274             # Fit scaler only on training set
275             train_X[:, :, i] = scaler.fit_transform(train_X[:, :,
                    i])  # Fit and transform on train data
276             val_X[:, :, i] = scaler.transform(val_X[:, :, i])  #
                    Transform validation data
277             test_X[:, :, i] = scaler.transform(test_X[:, :, i])  #
                     Transform test data
278
279     train_X = train_X.reshape((train_X.shape[0], train_X.shape[1],
             num_features - 1))
280     val_X = val_X.reshape((val_X.shape[0], val_X.shape[1],
            num_features - 1))
281
282     test_X = test_X.reshape((test_X.shape[0], test_X.shape[1],
            num_features - 1))
283
284     #Convert labels to categorical (one-hot encoding)
```

```
285     train_label = keras.utils.to_categorical(train_label,
            num_classes=3)
286     val_label = keras.utils.to_categorical(val_label, num_classes
            =3)
287     test_label = keras.utils.to_categorical(test_label,
            num_classes=3)
288 
289     #Printing the length of the 3 sets
290     print("
            ----------------------------------------------------------------------
            ")
291     print("Length of Training Set: ", len(train_X))
292     print("Length of Validation Set: ", len(val_X))
293     print("Length of Testing Set: ", len(test_X))
294     print("
            ----------------------------------------------------------------------
            ")
295 
296     return train_X, val_X, test_X, train_label, val_label,
            test_label, price_changes
297 
298 # Build the transformer model for classification (buy/sell/hold)
299 def build_model(stock_df, stock_name, num_features, sentiment):
300     print('Running the Model for the ' + str(stock_name) + ' Data
            :\n')
301 
302     train_X, val_X, test_X, train_label, val_label, test_label,
            price_changes = preprocess_data(stock_df, num_features)
303 
304     close_prices_changes1 = price_changes[-len(test_label):]
305     print(close_prices_changes1)
306     close_prices_changes = price_changes
307     print(close_prices_changes)
308 
```

```
309     training_close_prices_changes = price_changes[0:len(
            train_label)]
310     print(training_close_prices_changes)
311
312     closing_prices = stock_df['Adj Close'][-(len(test_label) + 1)
            :]
313     print(closing_prices)
314     initial_price = closing_prices[0]
315
316
317     adam = keras.optimizers.Adam(learning_rate=1e-3)
318     callbacks = [keras.callbacks.EarlyStopping(patience=30,
            restore_best_weights=True),
319                  keras.callbacks.LearningRateScheduler(
                         LR_scheduler)]
320
321     num_transformer_blocks = 2
322     inputs = keras.Input(shape=train_X.shape[1:])
323     x = inputs
324
325     # Stacking the transformer blocks
326     for _ in range(num_transformer_blocks):
327         x = transformer_encoder(inputs=x, head_size=512, num_heads
                =2, ff_dim=8, dropout=0.2)
328
329     x = layers.GlobalAveragePooling1D(data_format="channels_first"
            )(x)
330     x = layers.Dense(units=512, activation="relu")(x)
331     x = layers.Dropout(0.4)(x)
332
333     # Output layer for buy/sell/hold classification
334     outputs = layers.Dense(units=3, activation="softmax")(x)
335
336     model = keras.Model(inputs, outputs)
337
```

```
338     batch_size = 8
339     model.compile(loss=custom_profit_loss(
            training_close_prices_changes, len(train_X), batch_size),
            optimizer=adam, metrics=['accuracy', 'Precision', 'Recall'
            ])
340
341     model.summary()
342
343     # Train the model
344
345     print(train_label.shape[0])
346     model.fit(train_X, train_label, validation_data=(val_X,
            val_label), epochs=40,
347             batch_size=batch_size, shuffle=False, verbose=1,
                callbacks=callbacks)
348
349     print("
            ----------------------------------------------------------------
            ")
350     print("\nPrediction Step:")
351     print("
            ----------------------------------------------------------------
            ")
352
353     print(model.predict(test_X, verbose=1))
354     predictions = np.argmax(model.predict(test_X, verbose=0), axis
            =1)
355
356     print(predictions)
357
358     transaction_cost = 0.01
359     # Evaluate strategy using custom profit/loss
360     strategy_returns = [0]
361     for i in range(0, len(closing_prices) - 1):
362         if predictions[i] == 1:  # Buy action
```

```
363             profit = closing_prices.iloc[i + 1] - closing_prices.
                    iloc[i] - transaction_cost # Profit for buy
364             strategy_returns.append(profit)
365         elif predictions[i] == 2:  # Sell action
366             profit = closing_prices.iloc[i] - closing_prices.iloc[
                    i + 1] - transaction_cost
367             strategy_returns.append(profit)
368         else:
369             strategy_returns.append(0)  # Hold, no profit
370
371     # Plot cumulative profit/loss
372     dates, cumulative_strategy_returns =
            plot_cumulative_profit_loss(stock_df.index[-(len(
            close_prices_changes1) + 1):], closing_prices,
            strategy_returns, stock_name, sentiment)
373
374     scores = model.evaluate(test_X, test_label, verbose=1)
375
376     print("
            ----------------------------------------------------------------
            n")
377
378     # Displaying predicted vs actual data for the testing set
379     plt.figure(figsize=(10, 5))
380     plt.plot(stock_df.index[-len(test_label):], np.argmax(
            test_label, axis=1), label="Actual")
381     plt.plot(stock_df.index[-len(predictions):], predictions,
            label="Predicted")
382     if sentiment == True:
383       plt.title('Buy/Sell/Hold Predictions for ' + str(stock_name)
              + ' With Sentiment')
384     else:
385       plt.title('Buy/Sell/Hold Predictions for ' + str(stock_name)
              + ' Without Sentiment')
386     plt.xlabel('Time')
```

```
387     plt.ylabel('Action')
388     plt.legend()
389     plt.show()
390 
391     return scores, dates, cumulative_strategy_returns
392 
393 #Amazon Results With Sentiment
394 AMZN_score_sentiment, AMZN_dates_sentiment,
    AMZN_cumulative_strategy_returns_sentiment = build_model(out4,
     'AMZN', 9, True)
395 
396 #Amazon Results Without Sentiment
397 AMZN_score_no_sentiment, AMZN_dates_no_sentiment,
    AMZN_cumulative_strategy_returns_no_sentiment = build_model(
    out7, 'AMZN', 7, False)
398 
399 plt.figure(figsize=(10, 5))
400 plt.plot(AMZN_dates_sentiment,
    AMZN_cumulative_strategy_returns_sentiment, label="Cumulative
    Profit/Loss With Sentiment", color='blue')
401 plt.plot(AMZN_dates_no_sentiment,
    AMZN_cumulative_strategy_returns_no_sentiment, label="
    Cumulative Profit/Loss Without Sentiment", color='orange')
402 plt.title('Predicted Cumulative Profit Percentage Over Time For
    AMZN')
403 plt.xlabel('Trading Dates')
404 plt.ylabel('Cumulative Profit/Loss')
405 plt.legend()
406 plt.show()
```

Listing 1: Code to Generate Prediction Results for Amazon Stock Data