High Frequency Trading Via Transformer Deep Neural Networks

Víctor Domínguez Cámara

Bachelor's degree in Civil Engineering
Bachelor's degree in Mathematics

June 2023

Supervised by Prof. Daniel P. Palomar
Tutored by Prof. Esther Sala

Abstract

Natural Language Processing has been revolutionized by the Transformer architecture, as can be seen by the impact of Chat-GPT. On the other hand, High Frequency Trading (HFT) is a type of financial trading that takes place within the order of minutes or even less, unlike other types of trading which take place in the order of days or months. One way of storing this High Frequency information is the Limit Order Book (LOB), where one can see pending bid and ask orders. In this project, we will try to apply this new Deep Learning NLP architecture to the world of HFT. We first introduced and explained the Transformer architecture and its characteristic attention mechanism. Then, we explained several financial concepts relating to HFT. Finally, we applied the transformer architecture to try to solve two financial problems: predicting the hourly closing price of Bitcoin and predicting the mid-price movement of a Limit Order Book.

Keywords: Deep learning, Transformers, Quantitative Finance, High Frequency Trading.

MSC2020: 91G60, 91B25, 68T99

Resumen

El Procesamiento de Lenguajes Naturales se revolucionó con la introducción del "Transformer", tal y como se puede ver con el impacto de Chat-GPT. Por otro lado, la Negociación de Alta Frecuencia es un tipo de negociación financiera que se lleva a cabo en el orden de minutos, o hasta menos, a diferencia de otros tipos de negociación que ocurren en el orden de días o meses. Una manera de guardar información sobre la Negociación de Alta Frecuencia es con un Libro de Órdenes Límites, en el que uno puede ver las órdenes pendientes. En este proyecto, intentamos aplicar esta nueva arquitectura de Deep Learning de Procesamiento de Lenguajes Naturales al mundo de la Negociación de Alta Frecuencia. Primero, introducimos y explicamos el Transformer y su característico mecanismo de atención. Después, explicamos algunos conceptos financieros relativos al Libro de Órdenes Límites. Finalmente, aplicamos el Transformer para resolver dos problemas financieros: predecir el precio de cierre horario del Bitcoin y predecir el movimiento del precio medio de un Libro de Órdenes Límites.

Palabras clave: Deep Learning, Transformers, Finanzas Cuantitativas, Negociación de Alta Frecuencia

MSC2020: 91G60, 91B25, 68T99

Resum

El Processament de Llenguatges Naturals es va revolucionar amb la introducció del "Transformer", tal i com es pot veure amb l'impacte de Chat-GPT. D'altra banda, la Negociació d'Alta Freqüència és un tipus de negociació que es du a terme en l'ordre dels minuts, o encara menys, a diferència d'altres tipus de negociacions que es fan en l'ordre de dies o mesos. Una manera de guardar informació sobre la Negociació d'Alta Freqüència és amb un Llibre d'Ordres Límits, en el que un pot veure les ordres pendents. En aquest projecte, intentem aplicar aquesta nova arquitectura de Deep Learning de Processament de Llenguatges Naturals al món de la Negociació d'Alta Freqüència. Primer, introduïm i expliquem el Transformer i el seu característic mecanisme d'atenció. Després, expliquem alguns conceptes financers relatius al Llibre d'Ordres Límits. Finalment, apliquem el Transformer per resoldre dos problemes financers: predir el preu de tancament horari del Bitcoin i predir el moviment del preu mitjà d'un Llibre d'Ordres Límits.

Paraules clau: Deep Learning, Transformers, Finances Quantitatives, Negociació d'Alta Freqüència

MSC2020: 91G60, 91B25, 68T99

Acknowledgments

I would like to first thank Professor Daniel P. Palomar for hosting me and giving me the opportunity to come to Hong Kong, and for everything he has taught me in these five months, as well as his infinite patience when dealing with me. I would also like to thank all of the members of his research group for helping me when I needed it, and several other professors that welcomed me into the University and made me feel at home.

I would like to thank the Centre de Formació Interdisciplinària Superior (CFIS) as well as the Cellex foundation for making this opportunity possible in the first place, and for providing me with financial support.

Finally, I would like to thank my sister, Lucía, my mother, Montserrat, and my father, Rafael, for helping me get to where I am, as well as the rest of my family, my aunts, uncles, cousins and grandparents. I will be forever grateful for everything you have done for me. To Duna, for proving that distance is meaningless, and to all my friends and loved ones, I wish I could name all of you. Thank you for being there when I have needed it throughout these years. I am very lucky to be able to share this accomplishment with all of you.

Contents

1 Motivation **1**

2 Transformers background **2**
2.1 Tokenization . . . 3
2.2 Token embedding . . . 3
2.3 Positional encoding . . . 4
2.4 Multi-Head Attention . . . 6
2.5 Feed Forward Network layer . . . 10
2.6 Masked Multi-Head Attention . . . 11
2.7 The Encoder and the Decoder . . . 12

3 Finance background **15**

4 Prediction of the hourly closing price of Bitcoin **20**
4.1 Model and data . . . 20
4.2 Results . . . 25
4.3 Interpretation of Results . . . 27

5 Prediction of the Mid-Price of Limit Order Book data **29**
5.1 Motivation . . . 29
5.2 Data . . . 29
5.3 Models . . . 30
5.3.1 Model 1: Ridge Regression . . . 30
5.3.2 Model 2: Transformer . . . 31
5.3.3 Model 3: CNN+LSTM model . . . 32
5.3.4 Model 4: CNN+Transformer model . . . 32
5.4 Results . . . 34
5.4.1 Case k=10 . . . 35
5.4.2 Case k=20 . . . 41
5.4.3 Case k=50 . . . 46
5.4.4 Case k=100 . . . 52
5.5 Interpretation of Results . . . 57

6 Conclusions and further work **59**

1 Motivation

The Transformer has revolutionized the world of Deep Learning. At the time of writing this paper, Chat-GPT has already been released for a few months (since November of 2022) and GPT-4 has been recently released to the public. GPT stands for "Generative Pre-trained Transformer". The chat-bots, developed and offered by OpenAI, can answer with extreme precision almost any question it is posed (with some limitations, of course, such as personal opinions on topics). This ability has encouraged people to experiment with the bots, resulting in exponential growth of interest in both Artificial Intelligence and the Transformer architecture.

The Transformer architecture was first introduced in 2017 in the paper "Attention is all you need", by Vaswani et al. It has proven to be a revolution in Natural Language Processing models, displaying a higher efficiency and accuracy than previos architectures, such as Long Short-Term Memory (LSTM) networks or Convolutional Neural Networks (CNN). Nonetheless, this architecture can also be used to attempt to solve other problems, such as time series predictions or image processing, with various results.

The idea of this paper is to first understand the Transformer architecture and its attention method, and the apply it to the field of Finance and Financial Engineering. This paper will thus present attempts to use this architecture to predict the hourly closing price of Bitcoin and then to predict the virtual mid-price movement of Limit Order Book data.

2 Transformers background

First, we must understand the Transformer architecture, how and why it works. As stated before, this architecture was first introduced by Vaswani et al. in the "Attention is all you need" paper [1]. In the paper, authors provided this diagram to exemplify how the architecture is structured:

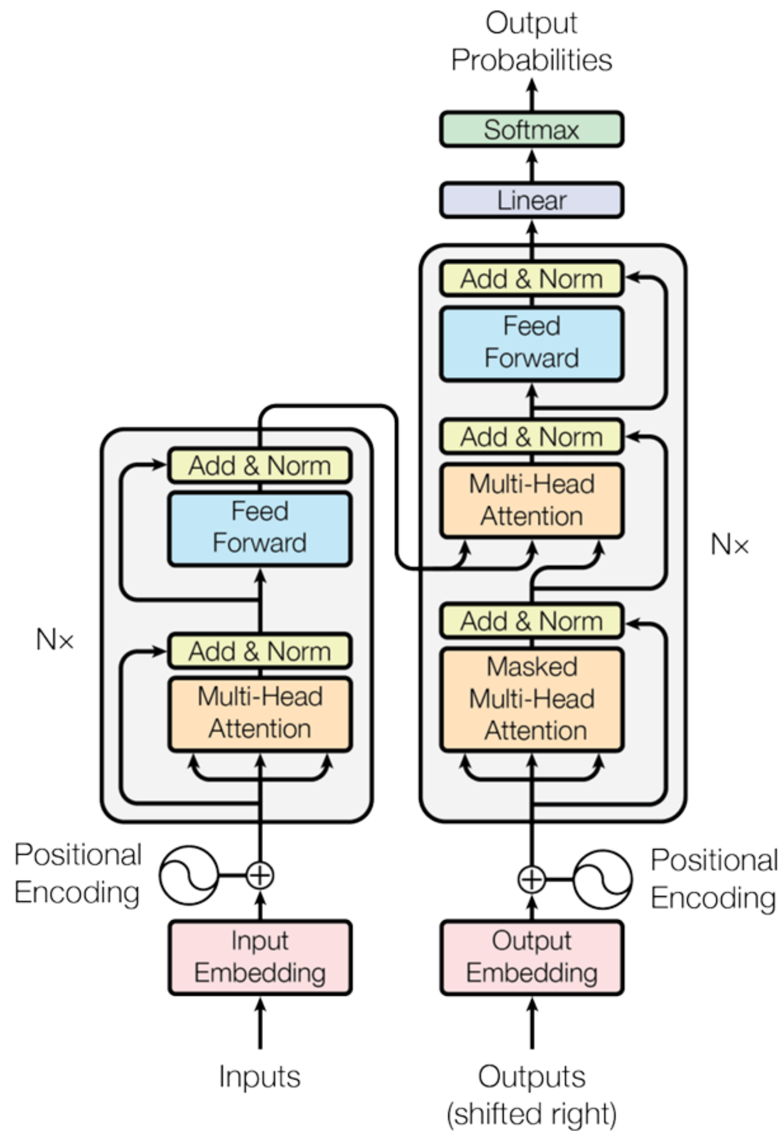

Figure 1: The Transformer architecture [1]

This architecture is compromised of several sub-parts and processes, such as, tokenization, embedding, positional encoding, an encoder (seen on Figure 1 as the left block), which consists of a multi-head attention mechanism, and a feed-forward network, and a decoder (seen on Figure 1 as the right block), consisting of a masked multi-head self attention mechanism, a multi-head attention mechanism, and a feed forward network. Thus, each process must be explained.

2.1 Tokenization

This first process is only necessary if the inputs of the Transformer are not numerical, as is the case of Chat-GPT and GPT-4, whose inputs are sentences made out of characters. Tokenization essentially consists of converting a series of characters into numbers. [4]

Let Ω be a dictionary, that is, a collection of all the possible combinations of characters that one may choose. Consider then the following injective (one-to-one) function:

$$T : \Omega^n \longrightarrow \mathbb{N}^n \tag{1}$$

where n is the number of dictionary elements. This function will map each element of the dictionary, to an associated natural number. The function T depends on the chosen dictionary. If the language of the input sentence is English written in lowercase letters, one can naively choose the dictionary to be the set of the 26 lowercase letters in the English alphabet, and then map each letter to its relative position in the alphabet (T(a)=1, T(b)=2, ..., T(z)=26). If we apply this tokenization to the word "earth", the result would be

$$T(earth)) = [5, 1, 18, 20, 8] \tag{2}$$

There are several possible tokenizations, alternatively, using the English alphabet as the alphabet and then associating each word with its positional order in the alphabet. Chat-GPT (as well as all of the GPT models by OpenAI), use a tokenization method which considers the token as common sequences found in text, after analyzing several texts and discovering several statistical relationships between those sequences. As such, full words may consist of more than one token. Approximately, each 4 characters will be a different token. To exemplify this, the tokenization of the word "indivisible" is $[521, 452, 12843]$, as "ind" becomes 521, "iv" becomes 452 and "isible" becomes 12843.

The tokenization mapping must be injective as, after performing every process and training the model, the outputs must be translated from numbers back to characters and words, which is a result of applying the inverse of the mapping, T^{-1}, to the set of tokens.

2.2 Token embedding

The embedding process consists on turning each input token (already a number) into a vector. Positional embedding consists on adding a positional component to the embedding process.

The idea behind embedding is, in the case of Natural Language Processing (the original objective of the Transformer architecture), to represent in each vector component some linguistic feature of the token. Given that, during training, the values in the embeddings will be modified by the model, the meaning of each component

may become apparent. For example, the value in the first position could be proximity in meaning, resulting in similar words having very similar embedding values, while very dissimilar words having very different values. The core idea of embedding is to receive a number (a token), and return a vector of a given dimension, which we will call d, with different values in each position of the vector. In the case that the problem to be solved has absolutely nothing to do with Language Processing, the values of each position of the embedding will not have such an obvious meaning.

2.3 Positional encoding

The positional encoding is an essential part of Natural Language Processing through the Transformer architecture. Other architectures, such as LSTMs or CNNs, do follow the order of the given language sequence, and thus, no positional encoding is required. The Transformer, however, does not have this intrinsic ordering ability. The order of words in sentences is essential to extract the meaning of the sentence, otherwise it is just a random collection of words, thus, when presented with a sentence, the Transformer requires that the order be explicitly shown.

The first idea that may come to mind is a naive approach of adding an extra layer to the embedding of each token with its absolute position in the sentence, the first token gets a 1, the second gets a 2... This approach, however, presents some flaws: first, that the length of the sentences may get increasingly large, and that during testing, the model may get sentences longer than it had during training.

The solution, proposed by the authors of the original Transformers paper, was to used another vector to apply the positional encoding. Let t be the position of a word in a sentence, d the encoding dimension (as mentioned before), with the condition that d be even. Then,

$$p_t(i) = \begin{cases} sin(\omega_k t) & \text{if } i = 2k \\ cos(\omega_k t) & \text{if } i = 2k+1 \end{cases} \tag{3}$$

with $\omega_k = \frac{1}{10000^{2k/d}}$ and $i \in \{1, ..., d\}$ the position of each value within p_t. This vector is then added to the original embedding, which is why the dimension d of the embedding must be the same as the dimension of the positional encoding layer. We can thus visualize this vector as pairs of sine and cosine functions, each with its own paired frequency,

$$p_t = \begin{pmatrix} sin(w_1 t) \\ cos(w_1 t) \\ sin(w_2 t) \\ cos(w_2 t) \\ ... \\ sin(w_{d/2} t) \\ cos(w_{d/2} t) \end{pmatrix} \tag{4}$$

which is why d must be divisible by 2.

The positional encoding layer is added to the original embedding, instead of just concatenating it with the original encoding, and one could think that this may lead to a modification or interference with the original encoding, resulting in the loss of information. To see why this is not the case, an example of positional encoding is shown in Figure 2.

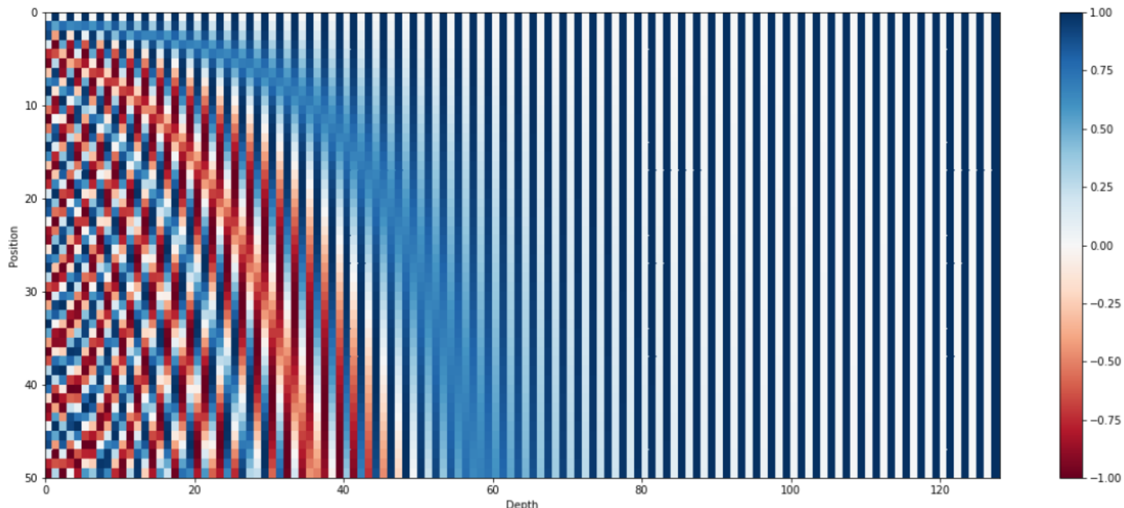

Figure 2: Positional encoding vectors (rows) for a 50 token succession with $d = 128$

In Figure 2 we see represented in each row the vector p_t for each position from 1 to 50 (vertical axis) with the values of each position shown by the color grid. This Figure shows that the positional encoding vectors are only significant in the first positions, meaning, the first values of the encoding. Therefore, the embedding and the training can be done in such a way that the most relevant information is stored only in the later positions of the embedding, so that there is no interference. Also, Figure 2 shows an embedding of dimension 128, while the original transformer proposed in the original paper had an embedding dimension of $d = 512$, which results in even more positions available in the embedding to store relevant information.

As stated before, there are alternatives to positional embedding, mainly when the problem to be solved has nothing to do with Natural Language Processing, for example, the study and prediction of Time Series. One proposed method of representing time is through the Time2Vec process, which is very similar in concept to the positional encoding layer. As its name suggests, Time2Vec provides a vector representation of time, in order to make sense of "when" did an event happen within a Time Series. The process is as follows: Let τ be a scalar time. Then, the Time2Vec representation of this τ, $t2v(\tau)$, will be a vector of size k+1:

$$t2v(\tau)(i) = \begin{cases} \omega_i \tau + \phi_i & \text{if } i = 0 \\ f(\omega_i \tau + \phi_i) & \text{if } i = 1, ..., k \end{cases} \tag{5}$$

where $i \in \{0, ..., k\}$ is the position of each value within $t2v(\tau)$, f is a periodic function (in the original Time2Vec paper, the sine function is considered) and ω_i and ϕ_i are learnable parameters. Therefore, there are several ways of building a positional/time encoding layer, depending on the problem at hand.

2.4 Multi-Head Attention

As stated before, the Encoder and Decoder are made up of a Multi-Head Attention layer and a Feed-Forward Network. Thus, we will begin by explaining what the Multi-Head Attention layer does.

In order to understand Multi-Head Attention, we must first tackle Single-Head Attention and then see how it generalizes to the Multi-Head case. Single-Head Attention can be explained in the following Figure, also appearing in the original Transformers paper.

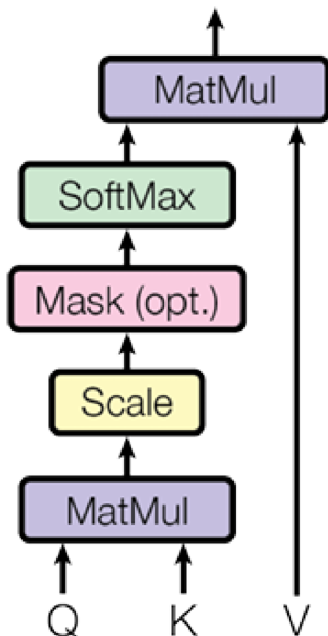

Figure 3: Single-head dot product attention [1]

As we can see in Figure 3, there are several layers to the dot-product attention mechanism, starting with the Q, K and V objects. The idea behind the encoder attention is to extract the maximum possible information out of the input tokens. Let's consider a sequence of n tokens that form a sentence, and let $e_1, e_2, ..., e_n$ be the embedding vectors (already positionally encoded) of each token. To obtain the Q, K and V objects of each vector, a linear layer is applied, which is the equivalent of applying a linear transformation L, as

$$L(e_i) = Ae_i + b \tag{6}$$

where A is a matrix and b is a vector. The values a_{ij} and b_i are scalars originally randomly distributed that the model will update during back-propagation as it learns. Thus, we can define the following objects:

$$Q_{e_i} = A_Q e_i + b_Q \tag{7}$$

$$K_{e_i} = A_K e_i + b_K \tag{8}$$

$$V_{e_i} = A_V e_i + b_V \tag{9}$$

As each embedding vector gets its own Q, K and V vector, if we concatenate all the e_i vectors into a matrix E, we can define the Q, K and V matrices as

$$Q = L_Q(E) \tag{10}$$

$$K = L_K(E) \tag{11}$$

$$V = L_V(E) \tag{12}$$

These matrices are called the Queries (Q), Keys (K) and Values (V) matrices. In the original paper, they have dimensions d_k (The Queries and Keys matrices) and d_v (the Values matrices). The idea behind these matrices is as follows: if we imagine that we want to search for a particular item, we may put the essential things we are looking for as the Queries of that item, and then each item of the list can display its qualities as the Keys. Then, the proximity of the searched qualities and the available qualities can be understood as the Values. Nonetheless, in this example, the Queries, Keys and Values come from different places. In the transformer encoder, the Queries, Keys and Values are all obtained from the embedding matrix. This is what's called Self-Attention. Self-Attention occurs when all the elements of the attention layer come from the same place (in the case of Natural Language Processing, from the training texts).

The idea is, then, to compute the similarities between the Queries (what we search for) and the Keys (what is offered). The solution is to compute the dot product between the two matrices. In Figure 3 this is represented as "MatMul". Thus, the idea of similarity is to compute

$$Similarity(Q, K) = QK^T \tag{13}$$

If we think in terms of vectors, this is the same idea behind finding the angle between 2 vectors. If we consider two vectors v and w, we recall that

$$cos(\theta) = \frac{v \cdot w}{|v||w|} \tag{14}$$

This idea is very similar to a "Similarity" as described in Equation 13, as two vectors are "most similar" when their cosine is 1 and "most dissimilar" when their cosine is -1. We see that the similarity between the Queries and the Keys uses the same idea as the cosine similarity between two vectors. In Equation 13, the Keys matrix is transposed so as to make the dimensions make sense and properly compute the dot product. We see a difference between the similarity described in Equation (13) and the cosine similarity described in Equation (14): The scaling factor. In computing cosine similarity, the result of the dot product is normalized by the factor of the norms of the two vectors, in order to get a result between -1 and 1 (and then be able to apply the *arccos* operation and get the actual angle between vectors), while the similarity described does not have a scaling factor. The original transformer paper does have a scaling factor in the similarity computation. In it, the similarity is defined as

$$Similarity(Q, K) = \frac{QK^T}{\sqrt{d_k}} \tag{15}$$

where d_k is the dimension of the Q and K matrices. We see that, in this case, the resulting values of the similarity may be larger than 1 and smaller than -1, but that won't be a problem because of the next operations in the Attention mechanism.

The next step is applying the SoftMax function (we will ignore Masking for the moment, as it will be explained in the decoder part of the Transformer). The SoftMax function is defined as follows:

$$\begin{aligned} \mathbb{S} : \mathbb{R}^N &\longrightarrow [0,1]^N \\ \mathbb{S}(\boldsymbol{x})_i &= \frac{e^{x_i}}{\sum_{k=1}^{N} e^{x_k}} \end{aligned} \tag{16}$$

Applying the SoftMax function to a vector, then, turns every value of the vector into a value between 0 and 1. Thus, the values can be understood as, for instante, a discrete probability distribution. Applying the SoftMax function to the similarity described in Equation 15 row-wise results in a matrix, as shown in the following equation

$$SoftMax\left(\frac{QK^T}{\sqrt{d_k}}\right) = \begin{pmatrix} s_{11} & s_{12} & \dots & s_{1n} \\ s_{21} & s_{22} & & . \\ . & & & . \\ . & & & . \\ . & & & . \\ s_{m1} & s_{m2} & \dots & s_{mn} \end{pmatrix} \tag{17}$$

Such that $s_{ij} \in [0,1]$ and $\sum_{j=1}^{n} s_{ij} = 1 \; \forall i \in 1, ..., m$.

The final step of the Single-Head Attention is to multiply the resulting matrix described in equation 24 with the Values matrix. Thus, we can define the attention as:

$$Attention(Q, K, V) = SoftMax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{18}$$

Now, as explained before, this process is a Single-Head Attention mechanism, and now, we will see the generalization to a Multi-Head case. The Multi-Head case can be visualized in Figure 4, as it was shown in the original Transformers paper.

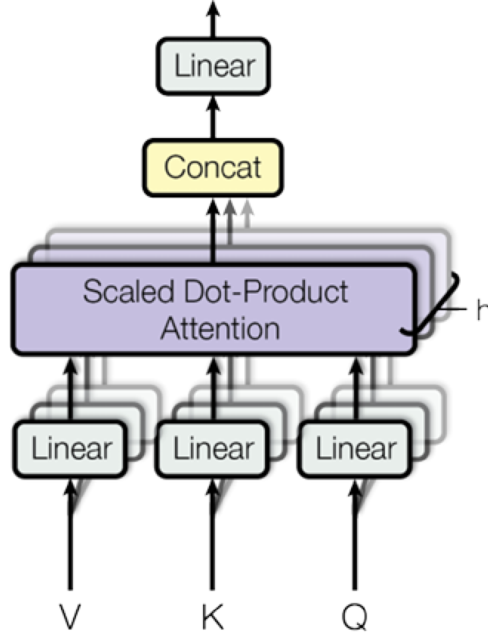

Figure 4: Multi-Head Attention process [1]

The idea behind this is to get the Q, K and V matrices and then project them h times using different learned linear functions. We then define an Attention Head as

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{19}$$

where W_i^Q, W_i^K and W_i^V are the different learned linear functions that project the Queries, Keys and Values, with $i \in 1, ..., h$. These different projections can me understood as the modifiers of the Q, K and V matrices such that they "focus" on different aspects of text, for example.

We may then define the Multi-Head Attention mechanism as

$$MultiHeadAttention(Q, K, V) = concat(head_1, head_2, ..., head_h)W^O \tag{20}$$

where W^O is a matrix of $\mathbb{R}^{hd_v \times d}$, a learned matrix applied also so that the dimensions of the Multi-Head Attention mechanism match, as concatenating several matrices will result in large dimensions.

If we look at Figure 1, we will see a "Add and Norm" layer shown in yellow. This step consists of normalizing each row of the resulting matrix, and helps in the efficiency of the model, as normalizing the layers results in a better running time and reduces the possibility of errors. The next layer is a Feed Forward Network.

2.5 Feed Forward Network layer

The last part of the encoder layer is passing the resulting data matrix through a Feed Forward Neural Network (FFNN). A Feed Forward Neural Network is the simplest type of Neural Networks, in which data is passed through the input nodes, the hidden layer, and the output nodes without forming any cycles. Thus, the data only moves in one direction, the forward direction (hence, the name) [2]. There are other types of Neural Networks where nodes do form cycles, such as Recurrent Neural Networks (RNNs). The flow of data of a FFNN can be visualized in Figure 5

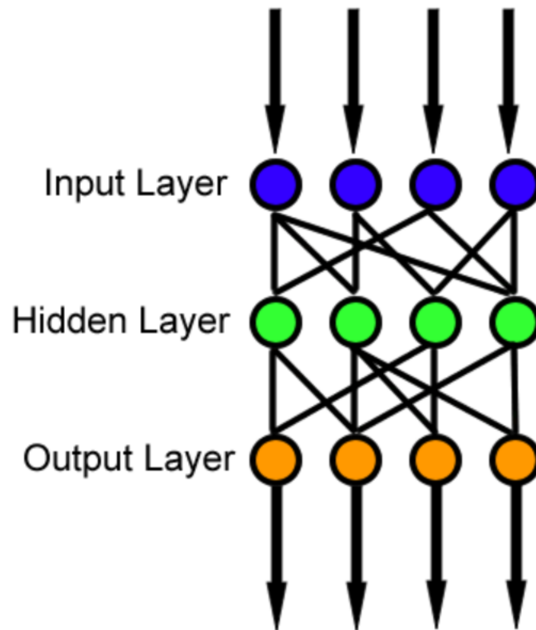

Figure 5: Example of a Feed Forward Neural Network (the arrows mark the flow of data)

In the original Transformers paper, the dimension of both the input layer and the output layer is $d = 512$ (same as the output dimension of the Multi-Head Attention process) and the inner layer has a dimension of $d_{in} = 2048$. Of course, these numbers can be changed depending on the problem at hand.

This network applies three operations to the inputs of the first layer. First, it applies a linear transformation to the input $\boldsymbol{x}$, as explained before and shown in Equation 6. Then, it applies a ReLU activation to this result. The ReLU activation function consists of the following:

$$ReLU(x) = max\{0, x\} \tag{21}$$

meaning that the value will only be passed forward if it is positive. Otherwise, it will pass 0. Then, the network applies another linear transformation to the result of the activation. Thus, the output can be summarized as

$$FFNN(x) = L_2(ReLU(L_1(x))) \tag{22}$$

This concludes the Encoder block of the Transformer. As Figure 1 indicates, the encoder block can be stacked N times to maximize the model's learning and precision. The original Transformer was composed of $N = 6$ layers. Now we may move to the decoder layer.

2.6 Masked Multi-Head Attention

If we look, once again, at Figure 1, we will notice that the left block (the encoder) is very similar, in components, to the right block (the decoder). The main component difference is the Masked Multi-Head Attention. If we return to Figure 3, there is a layer of the Single-Head Attention called the Mask layer, and there is the key of masked self attention.

If we follow the same process as before, we will calculate the scaled similarity, as computed in Equation (15). Now, before applying the SoftMax function, we will first add a masking matrix to the similarity matrix. This matrix will be

$$M = \begin{pmatrix} 0 & -\infty & -\infty & ... & -\infty \\ 0 & 0 & -\infty & ... & -\infty \\ . & & & & . \\ . & & & & . \\ . & & & & -\infty \\ 0 & 0 & 0 & ... & 0 \end{pmatrix} \tag{23}$$

this is, a lower triangular matrix with all values equal to zero, and the rest of the values equal to $-\infty$. The reasoning behind this matrix is that, after applying the SoftMax function to the similarity plus this matrix, the result will be

$$SoftMax\left(\frac{QK^T}{\sqrt{d_k}} + M\right) = SoftMax\begin{pmatrix} s_{11} & -\infty & -\infty & ... & -\infty \\ s_{21} & s_{22} & -\infty & ... & . \\ . & & & & . \\ . & & & & . \\ . & & & & -\infty \\ s_{m1} & s_{m2} & s_{m3} & ... & s_{mn} \end{pmatrix} \tag{24}$$

If we recall Equation (16), the SoftMax function assigns to each position of the vector the value of e^{x_i}. In this case, as several values of the matrix are $-\infty$, applying the SoftMax will map them to 0, resulting in

$$SoftMax\left(\frac{QK^T}{\sqrt{d_k}} + M\right) = \begin{pmatrix} 1 & 0 & 0 & ... & 0 \\ \widehat{s}_{21} & \widehat{s}_{22} & 0 & ... & . \\ . & & & & . \\ . & & & & . \\ . & & & & 0 \\ \widehat{s}_{m1} & \widehat{s}_{m2} & \widehat{s}_{m3} & ... & \widehat{s}_{mn} \end{pmatrix} \tag{25}$$

where $\widehat{s}_{ij} \in [0,1]$ and $\sum_{j=1}^{n} \widehat{s}_{ij} = 1 \ \forall i \in 2,...,m$. The reason the value of $\widehat{s}_{11}$ automatically becomes 1 is that the entire row is composed only of $\widehat{s}_{11}$ and zeroes, and applying the SoftMax function to this vector results in $(1, 0, 0, ...0)$.

2.7 The Encoder and the Decoder

In this final subsection, after understanding each component of the model individually, we will review each block as a whole and explain why it is designed that way.

Beginning with the Encoder block, this block performs its operations on unmasked data. This is, if it is fed a series of tokens, it will analyze all the relations and possible interactions of all the tokens will each other, no matter their position. This idea can be visualized in Figure 6

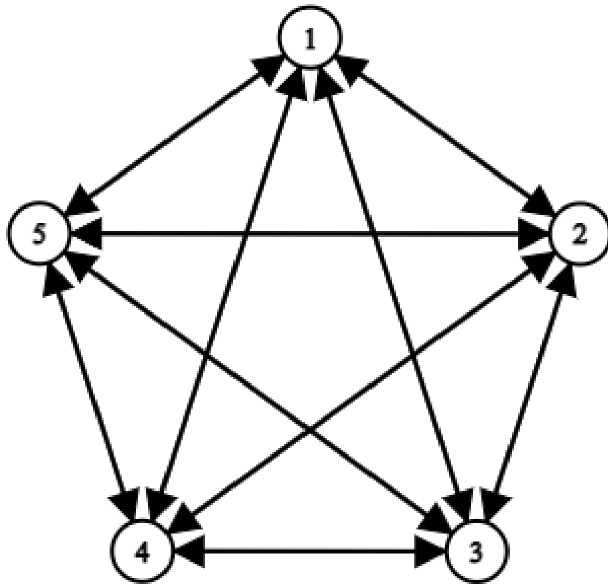

Figure 6: Example of the attention relations between tokens in the Encoder

In Figure 6, the vertices of the graph represent the tokens (the numbered vertices represent the position of the tokens in the sequence), and the edges represent the attention communication between these tokens. As we can see, this is a complete graph (K_5 in this case, it would be K_n in the case of n tokens). This exemplifies

how, in the encoder attention mechanism, all the tokens receive information from all the others, no matter the position of the sequence that they are in.

On the other hand, the masked attention of the Decoder makes it so that the information can not be shared totally between all the positions. That is, each token can only receive information from itself and the previous ones. This is exemplified in Figure 7. Here, the communication between nodes is represented by the direction of the graph. If node A is connected to node B, it means it can get information from that node. Thus, node 1 is not connected to any other nodes, and can only get information from itself. Recalling Equation (25), this is why the value of $\widehat{s}_{11} = 1$ directly, as it can not get information from any future nodes. Likewise, node 2 is only connected to node 1, and can only get information from itself and node 1. This is the reason as to why the upper triangular part of the Attention matrix is composed of zeros, so as to avoid communication with any future nodes.

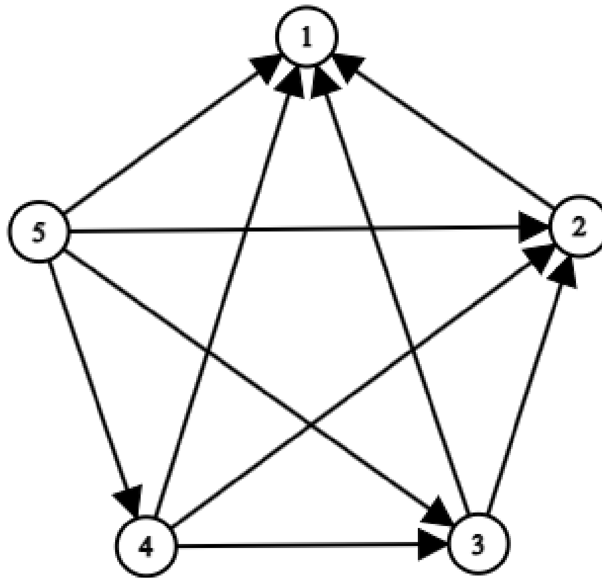

Figure 7: Example of the attention relations between tokens in the Decoder

The reasoning behind this difference between the encoder and the decoder can be understood with the following example: the original Transformer was trained and tested on Machine Translation problems, mainly English-to-German and English-to-French. Thus, after training, the Transformer was to be fed a sentence in English and be able to output its translation in French. For example, if given the sentence "I am terribly sorry", it should return "je suis vraiment désolé", but the process as to how it reaches that answer is the key to understanding the Encoder/Decoder relation. The input sentence is known, thus, un-masked attention can be applied to it in order to fully understand it. The output sentence, on the other hand, is sequen-

tially passed through the Decoder, which outputs the following token every time. In summary, the model will first output "je", then pass that sentence through the decoder, output "suis", then pass "je suis" through the decoder, output "vraiment", and so on. That's why the Decoder has a masked self-attention layer, as the tokens are only allowed to communicate with the past tokens, not the future ones. We can also see in the Decoder layer that the un-masked Multi-Head Attention Layer receives information from the encoder. In that layer, the Queries, Q matrices, are obtained from the decoder, but the Keys and Values, K and V matrices, are obtained from the encoder. Following the example of the translation, the Transformer gets information from the input text in English in order to get the correct output in French or German.

Thus, we now understand the Transformer architecture, and how the Attention mechanism works in order to maximize learning.

3 Finance background

Trading consist of exchanging securities, that is, buying or selling different market options, such as stocks (Apple, Tesla...), commodities or futures of those commodities (such as oil) or currencies (such as Dollars (US$) or, nowadays, Bitcoin (₿)). When dealing with these kinds of financial data, we must consider several facts as to why it acts that way. To better explain these facts, we will focus on the S&P 500 index and its stock prices. The S&P 500 index (Standard and Poor's 500 index) is an index based on the market capitalization of the 500 largest companies that trade on the New York Stock Exchange (NYSE) or the National Association of Securities Dealers Automated Quotation (NASDAQ), the two largest trading markets in the United States. This index captures approximately 80% of the United States market, and is thus considered as the most representative index for the market's situation. Firstly, we won't work with the value of the prices, but with the log-prices. Let p_t be the price of any asset at time t. We then define the log-price as

$$y_t = log(p_t) \tag{26}$$

The closing log-prices of the S&P 500 Index are shown in Figure 8

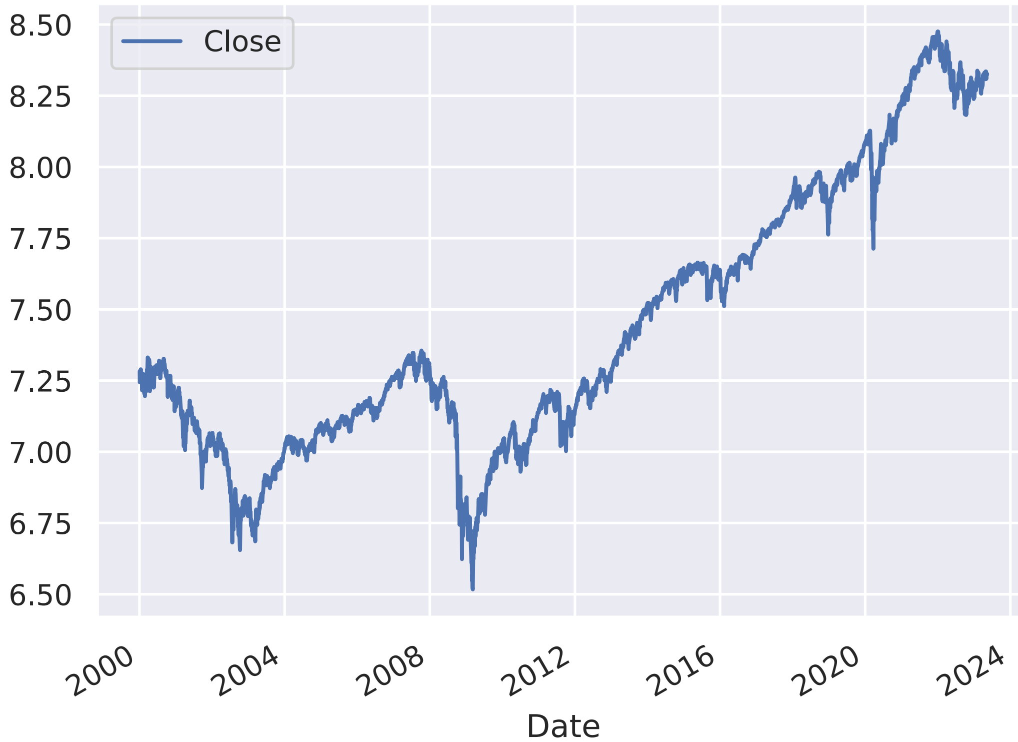

Figure 8: Closing log-prices of the S&P 500 Index from 01/01/2000 to 15/05/2023

After computing the log-prices, we can compute the log-returns, which are defined as

$$r_t = y_t - y_{t-1} \tag{27}$$

With these concepts, we may list a few financial stylized facts. These are characteristics of financial data which have been observed to happen by several independent studies across markets and in different time periods.

Firstly, there is lack of stationarity: past returns do not necessarily reflect future performance [5]. The second one is volatility clustering: high volatility events tend to cluster in time, this is, high volatility events tend to cause high volatility events, while low volatility events tend to cause low volatility events [6]. Volatility can be seen by time events in which the log-returns are wider, more volatile. Figure 9 shows the log-returns of the S&P-500.

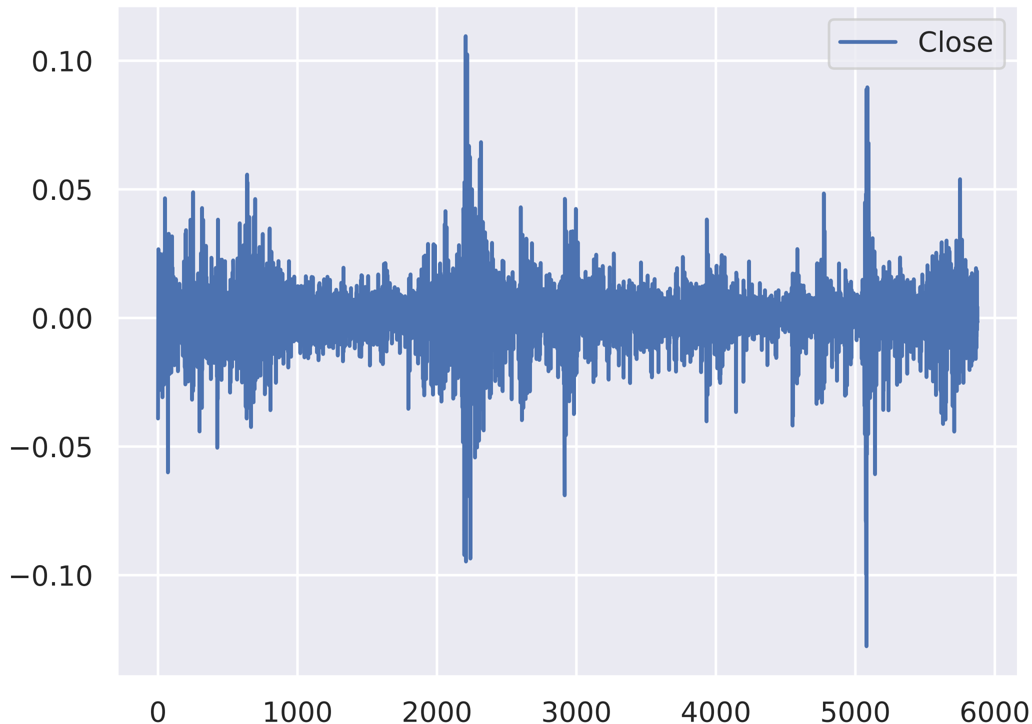

Figure 9: Log-returns of the S&P 500 Index from 01/01/2000 to 15/05/2023

In the previous figure we can see the clusters of volatility (very high and very low returns clumped together) as well as the periods of low volatility (small returns in absolute value).

The final stylized fact that will be presented is the fact that the distributions of log-returns are not Gaussian, but Heavy-Tailed [7]. A distribution is Heavy-Tailed if its tails are not exponentially bounded, this is, it has a heavier tail than the exponential distribution. Mathematically, a random variable X with distribution function $F(x) = P(X > x)$ is Heavy-Tailed if

$$\lim_{x \to \infty} e^{xt} F(x) = +\infty \tag{28}$$

The fact that financial data is usually Heavy-Tailed means that a Gaussian distribution (which isn't Heavy-Tailed) will not fit the data correctly. We can see this

fact by comparing a normal distribution to the log-returns seen in Figure 9.

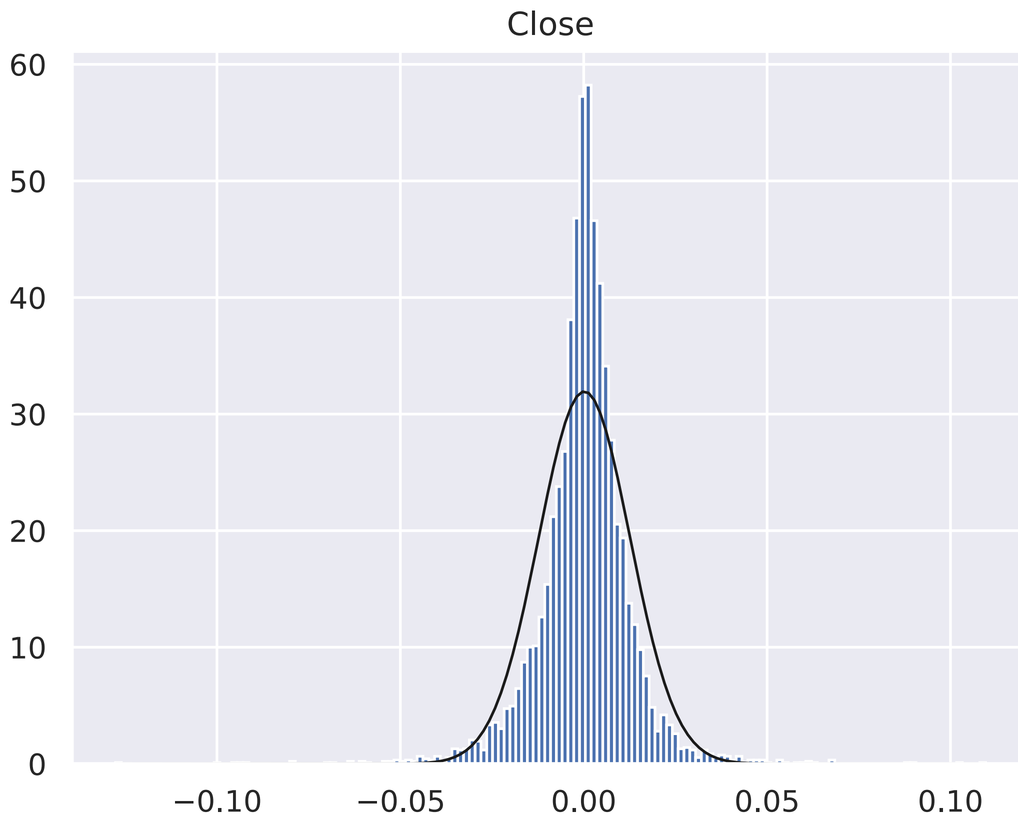

Figure 10: Attempt to fit a Gaussian distribution to the log-returns

From both Figure 10 and the Q-Q plot shown in Figure 11, we can conclude that the S&P 500 data is Heavy-Tailed.

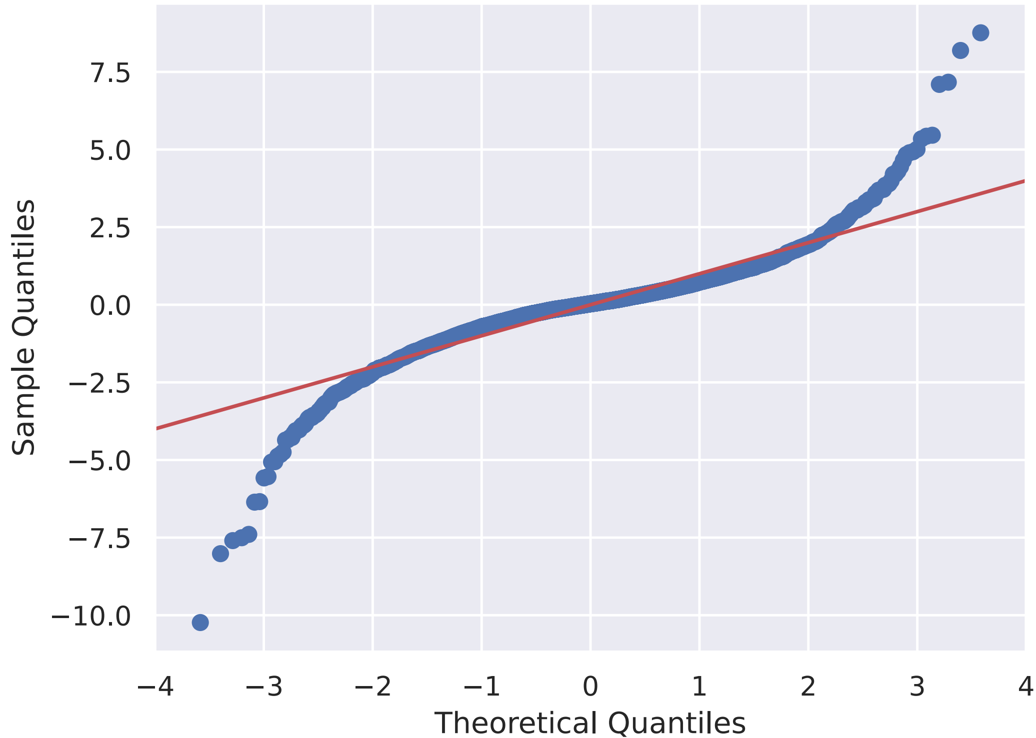

Figure 11: Q-Q plot of the fitted Gaussian distribution and the log-returns

After these financial data facts, we may now define what High Frequency Trading is. High Frequency Trading refers to the frequency by which we obtain the data. We may retrieve weekly, monthly or quarterly data. This would be considered low frequency data. Daily data would be considered medium frequency data. Finally, intra-day data, as hourly, half-hourly, 5 minute data or even tick-data would be considered high frequency data. Thus, High Frequency Trading consist on developing strategies and conducting trades with high frequency data. Of course, each type of data has its own characteristics. Lower frequency data is not as heavy-tailed, and thus the Gaussian distribution may fit. Medium frequency data is Heavy-Tailed and asymmetrical. High frequency data is Heavy-Tailed, and, when going below 5-minute data, the noise starts to interfere with the data. The first part of the paper consists on working with high frequency Bitcoin (₿) data, and the second part consist on working with Limit Order Book Data. Thus, now we will explain the basis of Limit Order Books and its relation to High Frequency Trading.

To start, we first must understand the market structure and its dynamics. A market can be separated into two sides: the "buy" side and the "sell" side. The buy side is driven by market agents who wish to purchase market options, be it investors, borrowers, hedgers or speculators. The sell side is driven by agents who wish to sell market options, be it dealers or brokers. We can then define an order o as a commitment to either buy or sell a specified asset. Thus, an order can be represented as a triplet

$$o_x = (p_x, q_x, t_x) \tag{29}$$

If o_x is a sell order placed at time t_x, $q_x > 0$ is the quantity of option the agent is willing to sell, and p_x will be the minimum price the agent is willing to sell at. On the other hand, if o_x is a buy order placed at time t_x, $q_x < 0$ is the quantity of option that the agent is willing to buy and p_x is the maximum price the agent is willing to buy at. The highest bid price is called the best bid price or bid quote, we will refer to it now as b_t, while the lowest ask price is called the best ask price, a_t. We can thus define a Limit Order Book (LOB) $\mathbb{L}(t)$ as the set of all active orders in a market at each time t. [9]

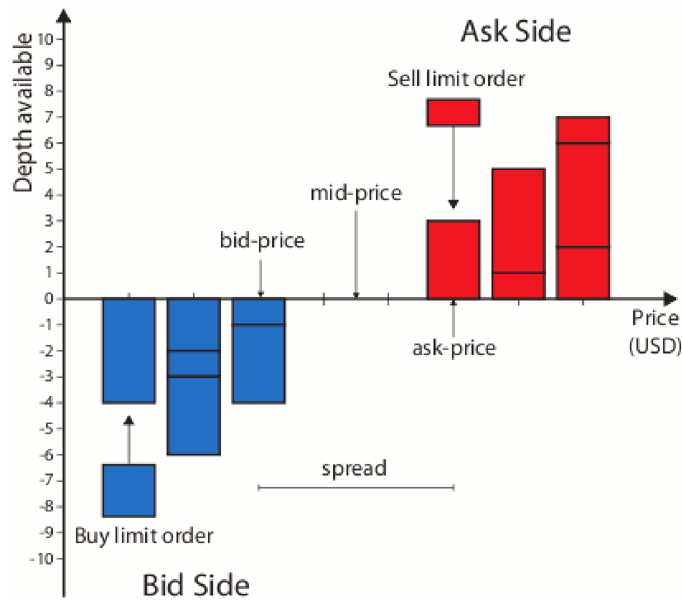

Figure 12: Representation of a LOB snapshot

When a new order is introduced to the LOB, it is processed as follows: if it is a buy order, the system will check if there are any active sell orders with a price smaller than the offered buy price. If there are, the system will execute the trade. It will continue to execute until the quantity q_x is reached or until there are no more possible trades. If it is a sell order, the system will check if there are any active buy orders with a price larger that the offered sell price, and will execute the trade until the quantity q_x is reached or until there are no more possible trades. In Figure 12, we can see a visual representation of the state of a Limit Order Book at a particular moment in time. The figure also shows the virtual midprice, which will be an important concept in Section 5. We define the mid-price as

$$m_t = \frac{a_t + b_t}{2} \tag{30}$$

4 Prediction of the hourly closing price of Bitcoin

As first approach to the use of transformers, we decided to code a transformer to attempt to predict the closing price of Bitcoin using hourly data. The idea behind this attempt is to test if the Transformer can extract information out of apparently uncorrelated data, and use it to properly predict movements in prices.

4.1 Model and data

The data was obtained from [8], consisting of a csv file that contained the open, high, low and closing price of Bitcoin in $US from the 15/05/2018 to the 28/02/2023. For this paper, we chose to work only with the closing hourly data. This file, thus, provided $41,997$ values, arranged in a vector. Figure 13 shows this data.

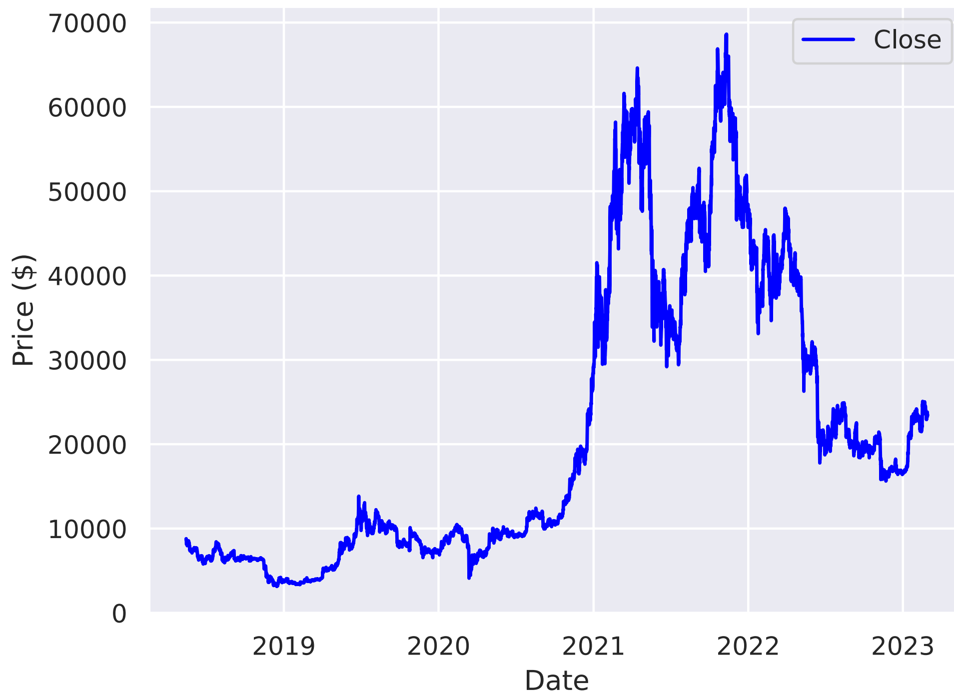

Figure 13: The evolution of the closing price of Bitcoin during the used time frame

In order to work with this data, first the logarithm of the prices must be computed. Then, the log-returns can be computed. The log-returns are calculated via

$$r_t = y_t - y_{t-1} \tag{31}$$

Figure 14 shows this result.

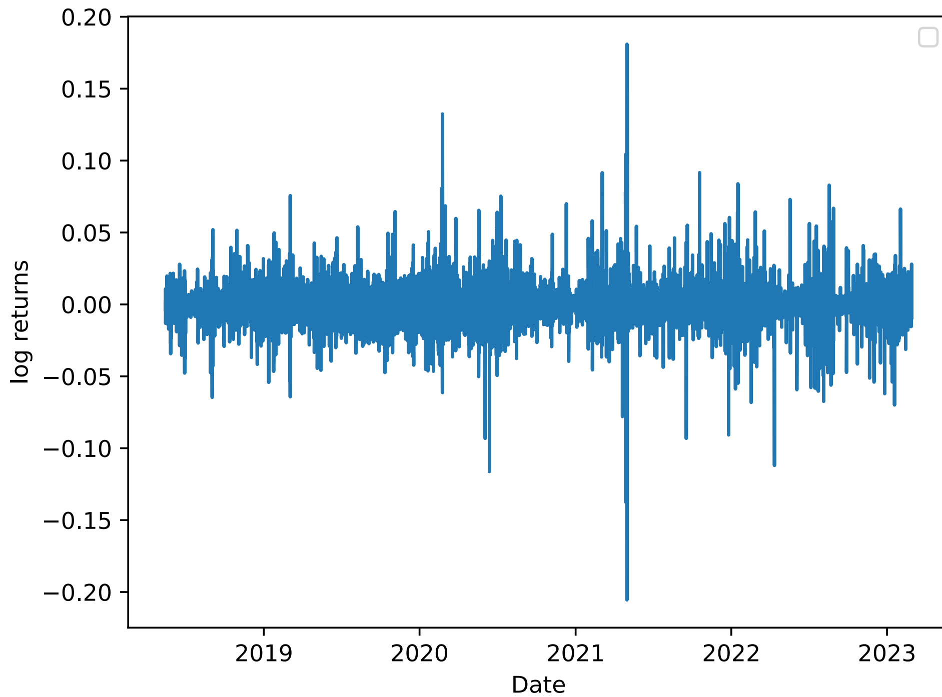

Figure 14: Log-returns of Bitcoin during the used time frame

Using the log-returns, we can also compute the autocorrelation and partial autocorrelation of the data. The autocorrelation function is the correlation of a signal with a delayed copy of itself as a function of the delay, whereas the partial autocorrelation function computes the conditional correlation between the signal and its delayed copy, but conditioned on all other delayed copies [7]. We can see these results in Figure 15.

As we can see, both the Autocorrelation and Partial Autocorrelation are practically zero when even the slightest lag is introduced, thus pointing to the fact that the predictability of the next prices is extremely low. Nonetheless, the point of this attempt is to test if the transformer architecture will be able to extract information even with such a low autocorrelation.

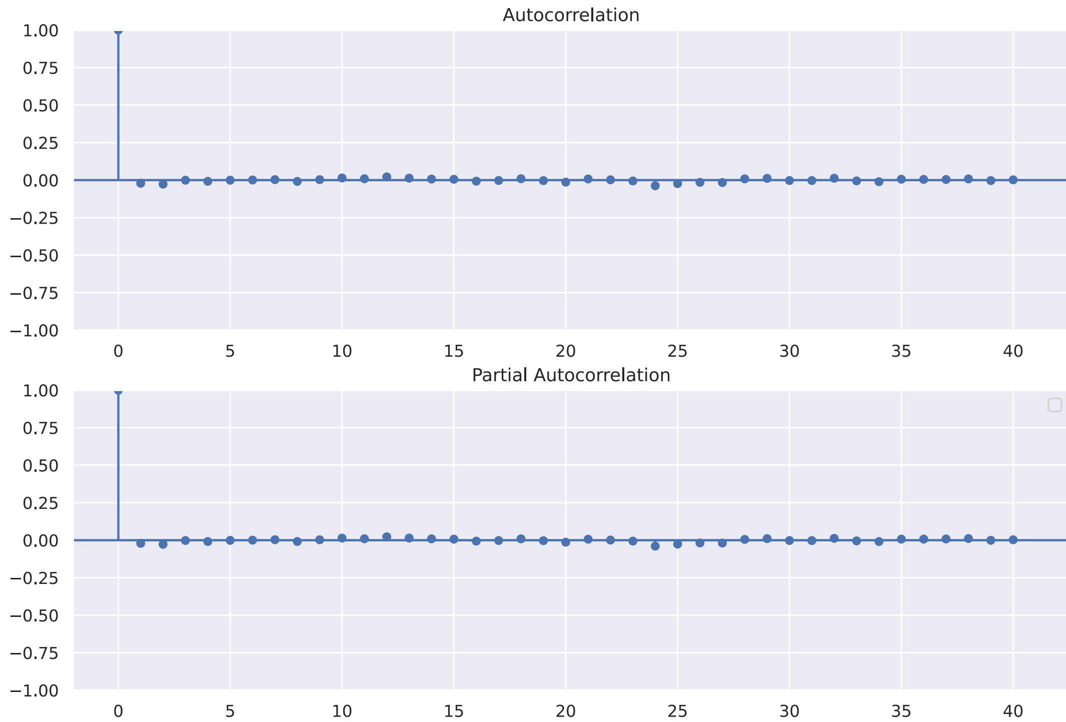

Figure 15: Autocorrelation and Partial Autocorrelation of the data used

The transformer was coded using the Python libraries TensorFlow and Keras, designed specifically for Deep Learning architectures. The built-in *MultiHeadAttention* function in Keras allows for the user to specify the parameters of the Transformer, such as the number of heads in the Multi Head Attention part, the number of Feed Forward Layers, and the number of transformers blocks that the model can work with. To best see the evolution and the different accuracy of the prediction, the transformer was compiled several times while modifying those parameters.

For this case, all the transformers were trained for 30 epochs. All had a fixed Feed Forward dimension of 150, and the number of attention heads was set to 12. The variable was the number of Transformer blocks N, as explained previously in the Transformer section. This number of Transformer blocks was set to 5, 10, 20 and 30, to see if there was any improvement in the accuracy of the predictability. The transformers were coded using the following algorithm:

Algorithm 1 The transformer architecture

```
function TRANSFORMER(Inputs, HeadSize, NumHeads, FFdim)
    x ← LayerNorm(Inputs)
    x ← MultiHeadAttention(HeadSize, NumHeads)(x)
    res ← x + Inputs      ▷ LayerNorm & MultiHeadAttetion are Keras functions
    x ← LayerNorm(res)
    x ← FeedForward(FFdim)
    return x + res
end function
```

The way that the transformer was given the data is as follows: the vector of data was sub-divided into several vectors of a determined length, and those were given as the input data. The target data for each price vector was the price on the following hour. To exemplify, let's consider the following hourly closing prices:

$$[25,000.00, 24,900.50, 23,798.09, 27,986.99, 35,677.55, 33,345.67] \tag{32}$$

Let's assume that, in this case, the length of the subdivided vector is 3. Then, the input data to the transformers and its corresponding target data would be:

$$[25,000.00, 24,900.50, 23,798.09] \longrightarrow [27,986.99] \tag{33}$$

$$[24,900.50, 23,798.09, 27,986.99] \longrightarrow [35,677.55] \tag{34}$$

$$[23,798.09, 27,986.99, 35,677.55] \longrightarrow [33,345.67] \tag{35}$$

This way, the Transformer would attempt to predict the next closing price considering only the three previous closing prices.

In the real case, the first $41,000$ closing prices were used as training data and vectorized as previously explained, and the remaining 997 values were used as testing data. Figure 16 shows the distribution of this data.

The transformer will then perform 2 tasks: first, predict if the price will go up or down, and then try to predict the exact price.

As for the accuracy of the up/down criteria, a naive method was implemented to compare itself to the transformer. This benchmark method consisted of the following: let p_t be the closing price at time t. Let y_t be the natural logarithm of p_t, that is,

$$y_t = log(p_t) \tag{36}$$

We can then model the log-prices as a random walk:

$$y_t = \mu + y_{t-1} + \epsilon_t \tag{37}$$

where μ is a fixed vector and ϵ_t is a Gaussian random variable with zero mean and constant covariance matrix. We can calculate the sample mean of a batch of M log-prices using the formula:

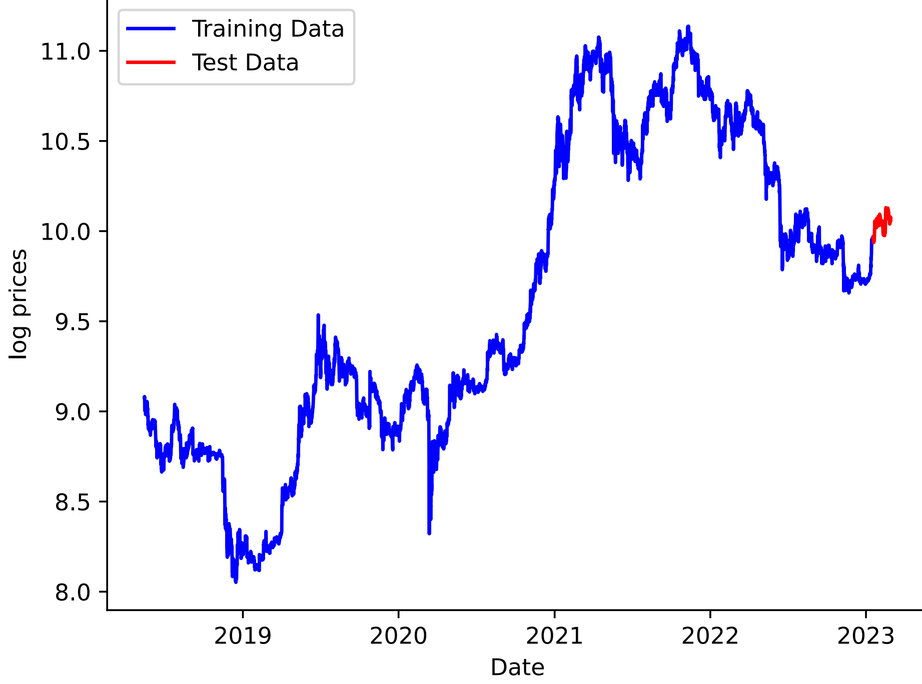

Figure 16: Training and Testing data of the log-prices

$$\widehat{\mu} = \frac{1}{M}\sum_{t=1}^{M} r_t \tag{38}$$

The idea of this method is to calculate $\widehat{\mu}$ for each batch of test data (the same sequence length as the transformer uses) and follow the same prediction:

$$Pred(t) = \begin{cases} up, & \text{if } \widehat{\mu}_{t-1} \geq 0 \\ down, & \text{if } \widehat{\mu}_{t-1} < 0 \end{cases} \tag{39}$$

By comparing the transformer predictions to this method, we can see if it is actually predicting at a good rate of if it is just "being lucky" because of the data distribution. As for the accuracy of the predictability of the actual price, another naive method was implemented, the exponential smoothing method. For this case, the price at time t $\widehat{y}_t$ will be calculated as follows:

$$\widehat{y}_t = \alpha y_{t-1} + (1-\alpha)\widehat{y}_{t-1} \tag{40}$$

where α is a hyper-parameter such that $0 < \alpha < 1$. This method is itself equivalent to the following formula:

$$\widehat{y}_t = \alpha y_{t-1} + \alpha(1-\alpha) y_{t-2} + \alpha(1-\alpha)^2 y_{t-3} + ... + \alpha(1-\alpha)^{(m-1)} y_{t-m} \tag{41}$$

thus the name of "exponential" smoothing. This will be the method used to predict the next closing price of Bitcoin. As α is not fixed, in this paper several different

values were chosen, so as to see which one fit the data better. The chose values were $\alpha = 0.25$, $\alpha = 0.33$, $\alpha = 0.50$, $\alpha = 0.66$, and $\alpha = 0.75$. To compare these results to the Transformer prediction, the Root Mean Squared Error (RMSE) will be used. The RMSE is defined as

$$RMSE = \sqrt{\frac{\sum_{i=1}^{M}(\widehat{y_i} - y_i)^2}{M}} \tag{42}$$

where $\widehat{y_i}$ is the predicted value and y_i is the measured value.

4.2 Results

The results of the accuracy of the "up/down" test is showed in Table 1 and Figure 17:

Seq length	Benchmark	T 5 blocks	T 10 blocks	T 20 blocks	T 30 blocks
12	50.87%	48.01%	48.11%	48.73%	48.62%
24	50.77%	49.94%	52.22%	48.40%	49.95%
48	50.69%	51.10%	49.89%	51.85%	51.87%
72	49.58%	52.77%	49.40%	49.73%	49.58%

Table 1: Results of the accuracy of the methods

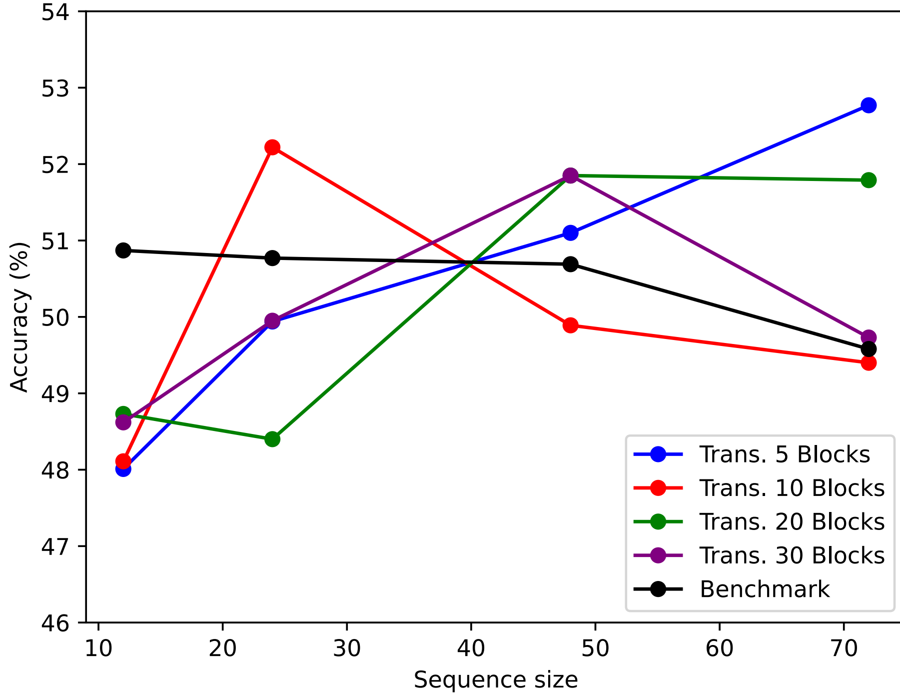

Figure 17: Results of the accuracy of the methods

With respect to the prediction of the real prices, first the RMSE of the different exponential smoothing methods was computed and plotted, as shown in Table 2 and Figure 18.

Seq length	$\alpha = 0.25$	$\alpha = 0.33$	$\alpha = 0.50$	$\alpha = 0.66$	$\alpha = 0.75$
12	0.3186	0.0827	0.0061	0.0052	0.0050
24	0.0129	0.0068	0.0057	0.0052	0.0159
48	0.0076	0.0067	0.0056	0.0052	0.0051
72	0.0077	0.0068	0.0057	0.0053	0.0051

Table 2: RMSE of the benchmark methods with different values of α

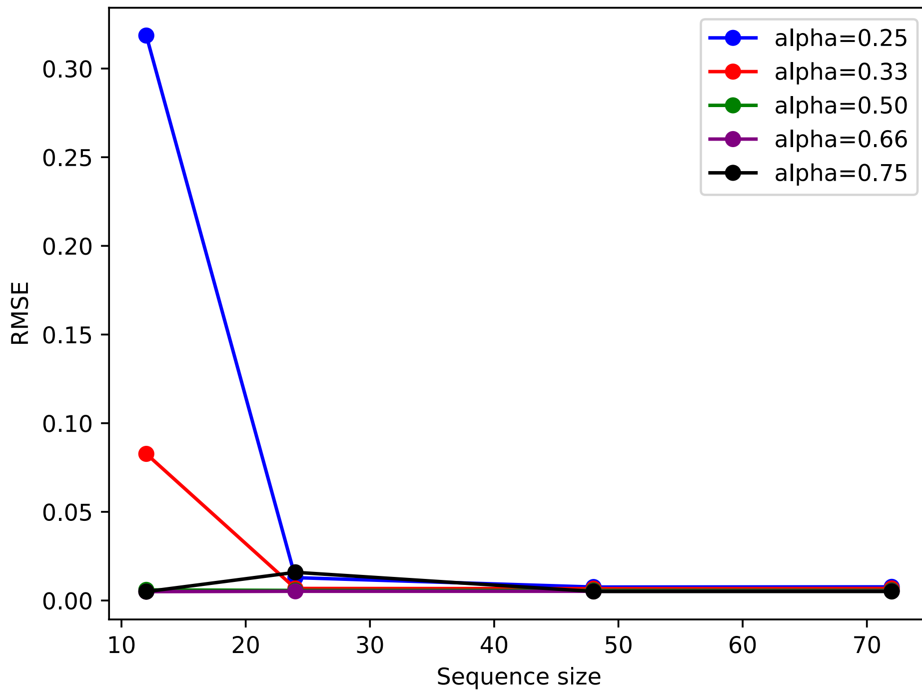

Figure 18: RMSE of the benchmark methods for different values of α

To compare the results of the Transformer, only the benchmark method with a value of $\alpha = 0.66$ will be considered, as it is the most precise of all. Table 3 and Figure 19 show the results of the RMSE of the different Transformers.

Seq length	T 5 blocks	T 10 blocks	T 20 blocks	T 30 blocks
12	0.0279	0.0248	0.0220	0.0186
24	0.0159	0.0159	0.0240	0.0203
48	0.0286	0.0163	0.0211	0.0230
72	0.0321	0.0366	0.0210	0.0290

Table 3: RMSE of the Transformer predictions

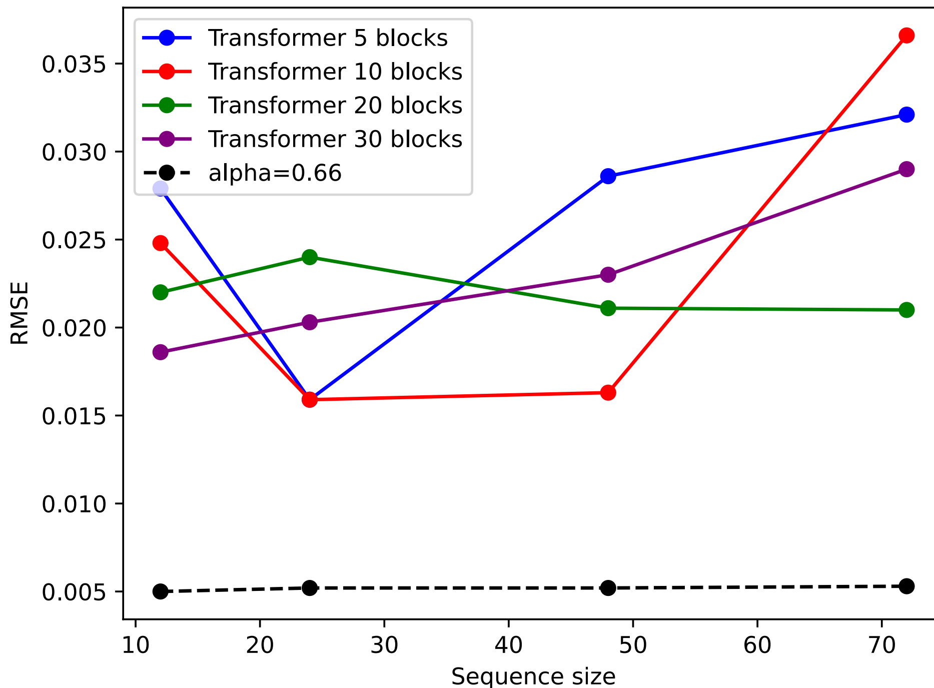

Figure 19: RMSE of the Transformer predictions

4.3 Interpretation of Results

The first thing noticed from these results is that there appears to be no correlation between increasing the number of transformer blocks and the increase of predictability power or reducing the Root Mean Squared Error. As a final attempt to see how much could the predictability power increase, we ran the best model, which, according to Table 1 and Figure 5 is the Transformer with 5 blocks using a sequence size of 72 hours, during 150 epochs instead of 30. The results of this transformer was an accuracy of 53.75% and a RMSE of 0.0298. This, of course, constitutes and improvement over the previous attempt, but not an amazing one. Assuming that this “up/down” 53.75% accuracy was to hold from now on, an investor using this information would eventually make money, but, given the disparity of results obtained, it is unlikely that this accuracy will hold.

That being said, the conclusion of this attempt is that, while the transformer

is an extremely potent tool when it comes to speech and pattern recognition and prediction, it is not useful when it comes to the prediction of Bitcoin. By looking at the figures, we can see that the maximum accuracy is not higher than 54%, thus pointing to the fact that the model cannot extract information, and is only slightly better than the naive method in very specific cases.

As expected, just as many other stock prices, Bitcoin can not be successfully predicted using only its own data (as this model didn't consider any external factors that may also affect prices), and thus other methods will be required in order to achieve an effective trading strategy.

5 Prediction of the Mid-Price of Limit Order Book data

5.1 Motivation

Previously, an unsuccessful attempt was made to predict the closing price of Bitcoin. The attempt failed, mainly, because of the lack of autocorrelation of the data and the fact that no other extra factors where considered into the decision making process. For example, the prices of other stocks were not considered, or the effect of international events, such as establishing Bitcoin as legal currency in a country. Thus, other than the lack of autocorrelation, the lack of extra information was the issue.

A Limit Order Book, as it contains previous orders, their price ask/bid, and the amount to be bought/sold, contains a greater amount of information that can be exploited by the models. Therefore, the transformer architecture should be better at predicting changes in a Limit Order Book.

In this section, several models will be tested in the task to predict the change of the mid-price of a Limit Order Book.

5.2 Data

The FI-2010 dataset is made up of 10 days of trading data from the Helsinki Stock Exchange, operated by Nasdaq Nordic, consisting of 10 levels on each side of the Limit Order Book. The possible order are executions, order submissions and order cancelations, and are not uniformly distributed in time. The data set is restricted only to normal trading hours (No Auction).

The data is organized such that there are 10 orders in the ask side, and 10 orders in the buy side. These orders are organized as vectors. Let E_t be an event of the Limit Order Book at time t. Then, $E_t = \{p_a^i, v_a^i, p_b^i, v_b^i\}_{i=1}^{10}$, where $p_a^i(t)$ is the price of sell orders at time t at level i, $v_a^i(t)$ is the volume of sell orders at time t at level i, $p_b^i(t)$ is the price of buy orders at time t at level i, $v_b^i(t)$ is the volume of buy orders at time t at level i. The structure of the data can be seen in Figure 20.

The data was obtained from [10], and was already pre-processed by its authors. Each event is normalized by the z-score, that is,

$$\bar{E}_t = \frac{E_t - \bar{y}}{\sigma_{\bar{y}}} \tag{43}$$

where $\bar{y}$ is the mean of the previous days' data and $\sigma_{\bar{y}}$ is the standard deviation of the previous days' data. The data included other extra hand-crafted features, but those were ignored.

The objective is to predict the mid-price movement. The mid-price is defined as:

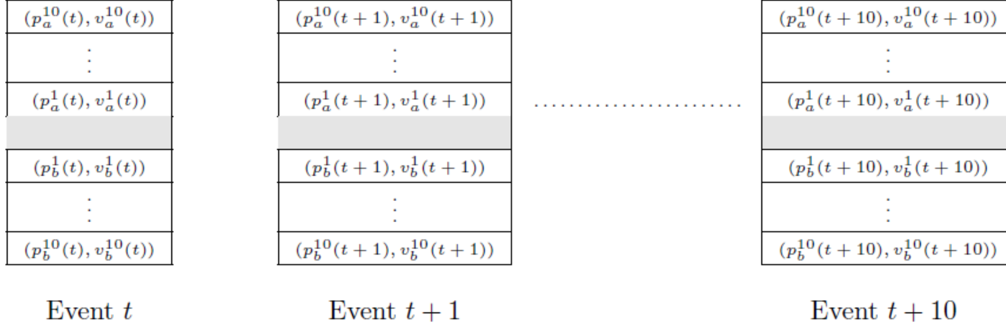

Figure 20: Structure of the data set through 10 ticks of time [13]

$$p_m(t) = \frac{p_a^1(t) + p_b^1(t)}{2} \tag{44}$$

That is, the average of the bid-price and ask-price of the first event. To calculate the change and direction of the mid-price, the mean of the k next mid-prices was calculated. This is:

$$m_k(t) = \frac{1}{k}\sum_{i=1}^{k} p_m(t+i) \tag{45}$$

Thus, the direction of the mid-price is calculated according to the following ratio:

$$R_k(t) = \frac{m_k(t) - p_m(t)}{p_m(t)} \tag{46}$$

If $R_k(t) > 0.002$, then a value of 2 is assigned (the direction is “up”), If $R_k(t) < 0.002$, then a value of 0 is assigned (the direction is “down”), and a value of 1 is assigned otherwise (the direction is “neutral”). The pre-processed FI-2010 data already had the following values added.

The dataset offered 4 different horizons, that is, values of k. These are $k = \{10, 20, 50, 100\}$, meaning that the directions are calculated with $10, 20, 50$ or 100 events into the future.

5.3 Models

The input data for each of the three models consisted of the 100 most recent events, meaning that the data consisted of 100×40 matrices, each of them being 1 input to the models. All the models were trained on the same data.

5.3.1 Model 1: Ridge Regression

The first model will be a Ridge Regression between the values of each event (which will be represented by a vector x) and the target values (represented by a vector t) [11]. The idea behind a Ridge Regression is to find a matrix M such that it maps each event to its target, that is,

$$t_i = M^t x_i \tag{47}$$

for all i. This matrix M is calculated solving the following optimization problem:

$$\widehat{M} = \arg\min_{M} ||M^T X - T||^2 + \lambda ||M||^2 \tag{48}$$

where X is the matrix formed by the event vectors x as columns and T is the matrix formed by the target vectors t as columns.

In this case, the vectors x are formed by the LOB data of each tick in the FI-2010 dataset. The target vectors, t, have 3 dimensions and a value of $+1$ or -1 depending on the classification of x. For example, if the prediction of x is that the mid-price will stay constant, that is, a 1, then $t = (-1, +1, -1)^t$. If the prediction is classified as up, that is, a 2, then $t = (-1, -1, +1)^t$. If it is a 0, then $t = (+1, -1, -1)^t$.

The solution of this Ridge Regression can be expressed in closed form, via the following formula:

$$\widehat{M} = (XX^t + \lambda I)^{-1})XT^t \tag{49}$$

This matrix can be best estimated using the training data, and then checked using the testing data. That is, after obtaining $\widehat{M}$, using x_{test}, we will calculate

$$p = \widehat{M}^t x_{test} \tag{50}$$

which will be a matrix of $\mathbb{R}^{C \times 3}$, to which we will apply the argmax function to determine the classification into 0, 1 or 2:

$$\widehat{p} = \arg\max_{k} p_k \tag{51}$$

This model will serve as a Baseline model to which we can compare the Transformer model, and see if it outperforms it or not.

5.3.2 Model 2: Transformer

This model is very similar to the one used to try to predict the price of Bitcoin. The idea is simple: use a transformer without any modifications.

Algorithm 2 The Transformer architecture

function TRANS(Inputs, NumHeads, NumBlocks FFdim)
- $\mathtt{x} \leftarrow PosEcoding(\mathtt{Inputs})$
- $\mathtt{x} \leftarrow LayerNorm(\mathtt{x})$
- $\mathtt{x} \leftarrow MultiHeadAttention(\mathtt{NumBlocks}, \mathtt{NumHeads})(x)$
- $\mathtt{res} \leftarrow \mathtt{x} + \mathtt{Inputs}$
- $\mathtt{x} \leftarrow LayerNorm(\mathtt{res})$
- $\mathtt{x} \leftarrow FeedForward(\mathtt{FFdim})$
- **return** $\mathtt{x} + \mathtt{res}$

end function

In this case, given than the data is presented as $E_t = \{p_a^i, v_a^i, p_b^i, v_b^i\}_{i=1}^{10}$, where the prices occupy the even positions (starting with zero) and the volumes the odd

positions, the positional encoding was applied only to the odd positions. This is due to the fact that the prediction has to be made on the mid-price, which the volume does not affect. That way, we get both a positional encoding that still does to modify the prices value. Recalling Equation (3), in this case, the encoding vector is as follows:

$$p_t = \begin{pmatrix} 0 \\ cos(w_1 t) \\ 0 \\ cos(w_2 t) \\ \dots \\ 0 \\ cos(w_{d/2} t) \end{pmatrix} \tag{52}$$

The rest of the Algorithm is exactly the same idea as in Algorithm 1. The following Table shows the chosen hyperparameters.

Hyperparameter	Value
Num Heads	4
Num Blocks	4
FFdim	500
Dropout	0.1

Table 4: Hyperparameters of the Transformer model

5.3.3 Model 3: CNN+LSTM model

This model is based on the "DeepLOB" model presented by Zihao Zhang, Stefan Zohren and Stephen Roberts [12]. This model uses a convolutional neural network structure to predict the Limit Order Book midprice followed by an LSTM. The model architecture is explained in Figures 21 and 22. In those Figures, the notation "$1 \times 2@16$" indicates a convolutional layer of 16 filters of size 1×2, respectively with the rest of possibilities.

The paper shows the results of the model, as already trained by the authors. Nonetheless, we re-coded de model and trained it the same way as the rest of the models, to have a better comparison. That way, the results shown, may differ from the ones in the original paper [12]

5.3.4 Model 4: CNN+Transformer model

This model consists of the same Convolutional Neural Network as the previos model, with the difference that in this case, a Transformer is applied instead of an LSTM. The hyperparameters are shown in Table 5.

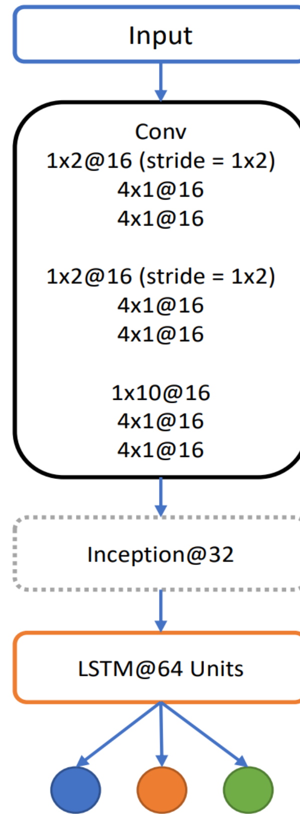

Figure 21: Structure of the DeepLOB architecture

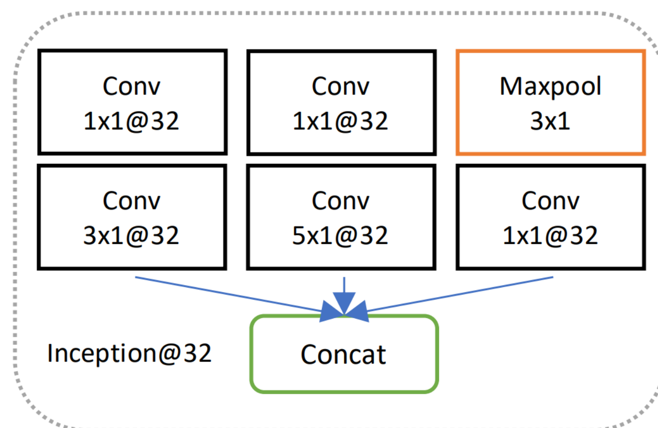

Figure 22: Structure of inception module presented in the DeepLOB architecture

Hyperparameter	Value
Num Heads	2
Num Blocks	5
FFdim	200
Dropout	0.1

Table 5: Hyperparameters of the CNN+Transformer model

5.4 Results

The results are displayed in the following tables. Four parameters were used to examine the efficiency of the models: accuracy, precision, recall and F1 score.
Accuracy is defined as the number of correct predictions over the total number of attempts.

$$Acc = \frac{\#correct}{\#total} \tag{53}$$

Precision is defined as the number of true positives (P_T) over the number of true positives (P_T) + false positives (P_F). Precision is thus the ability of the model not to label as positive a sample that is negative.

$$Prec = \frac{P_T}{P_T + P_F} \tag{54}$$

Recall is defined as the number of true positives (P_T) over the number of true positives (P_T) plus false negatives (N_F). Recall is thus the ability of the model to find all positive samples.

$$Recall = \frac{P_T}{P_T + N_F} \tag{55}$$

The F1 score is defined as

$$F1 = 2\frac{Prec * Recall}{Prec + Recall} \tag{56}$$

It can be interpreted as a type of harmonic mean between Precision and Recall.
All indicators reach a maximum value of 1 (indicating the best possible performance) and a minimum value of 0 (indicating the worst possible performance).

In order to evaluate the model, the chosen loss function is the Sparse Categorical Cross-Entropy function. We must first define the Entropy of a system. Let X be a random variable, with $p(x)$ its probability distribution function. Entropy H is defined as

$$H(X) = -\sum_x p(x) log_2 p(x) \tag{57}$$

The entropy measures the degree of uncertainty of the probability distribution. The bigger the entropy, the more uncertainty, an the smaller the entropy, the lesser the

uncertainty.

The reason behind the negative sign is the fact that the probabilities $p(x)$ are always between 0 and 1, and, thus, the $log_2p(x)$ would return negative values. In order to keep the entropy as a positive value, the negative sign is added.

The categorical cross-entropy is then defined as

$$H_C = -\sum_{i=1}^{N} t_i log_2 p_i \tag{58}$$

where N is the number of classes, t_i is the true label and p_i is the probability of each class. The following example will clarify this formula. Suppose we have two vectors, one made with probabilities, and one with the true labels. The first vector S=(0.87, 0.1, 0.01, 0.06) and the second vector T=(1, 0, 0, 0). Then, the cross-entropy value will be

$$H_C = -[1log_2(0.87) + 0log_2(0.1) + 0log_2(0.01) + 0log_2(0.06)] = 0.2 \tag{59}$$

In this example, the value of the functions is 0.2. The idea is to minimize the value of the cross-entropy. A model with a value of cross-entropy equal to 0, would be a perfect model without error.

The Sparse Categorical Cross-Entropy, the one used in this problem, is defined exactly as the Categorical Cross-Entropy, but the only difference is the type of data used. The data must be in one-hot vector form (i.e., [1, 0, 0], [0, 1, 0] and [0, 0, 1] for three classes) to use the Categorical Cross-Entropy, and must be in integer form ([0], [1], [2] for three classes) for the Sparse Categorical Cross-Entropy. The objective is still to minimize it and try to get it as close to 0 as possible.

The results of the predictions can be seen in the following tables:

5.4.1 Case k=10

The data distribution is shown in the following table.

	labeled 0	labeled 1	labeled 2	Total
Train data	40,957 (20.1%)	121,923 (59.8%)	40,919 (20.1%)	203,799
Val data	9,305 (18.3%)	32,282 (63.4%)	9,362 (18.3%)	50,949
Test data	21,167 (15.2%)	98,637 (70.7%)	19,782 (14.1%)	139,586

Table 6: Data distribution for k=10

We show here the different variations of Training & Validation Accuracy and Training & Validation Loss of each of the 3 models and their evolution for a large number of epochs.

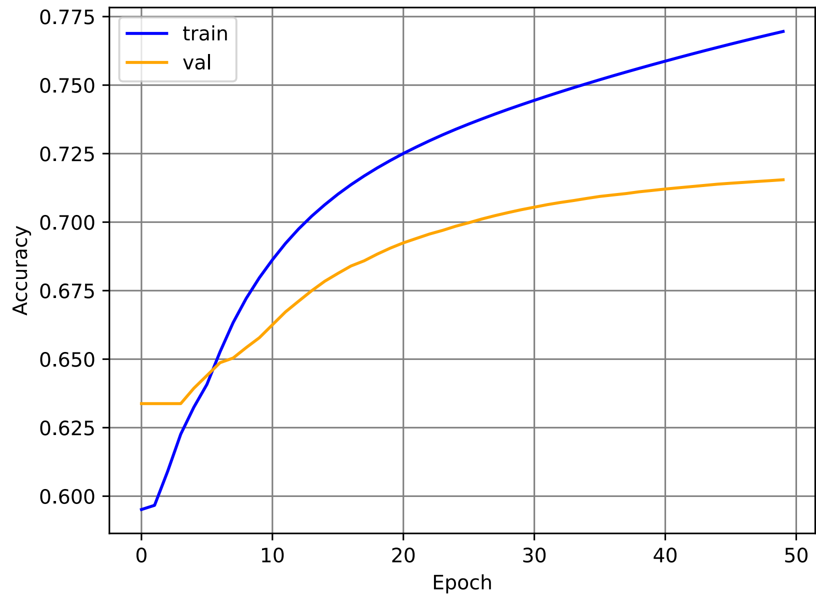

Figure 23: Train & Val Accuracy vs Epochs of the CNN+LSTM model when k=10 for 50 epochs

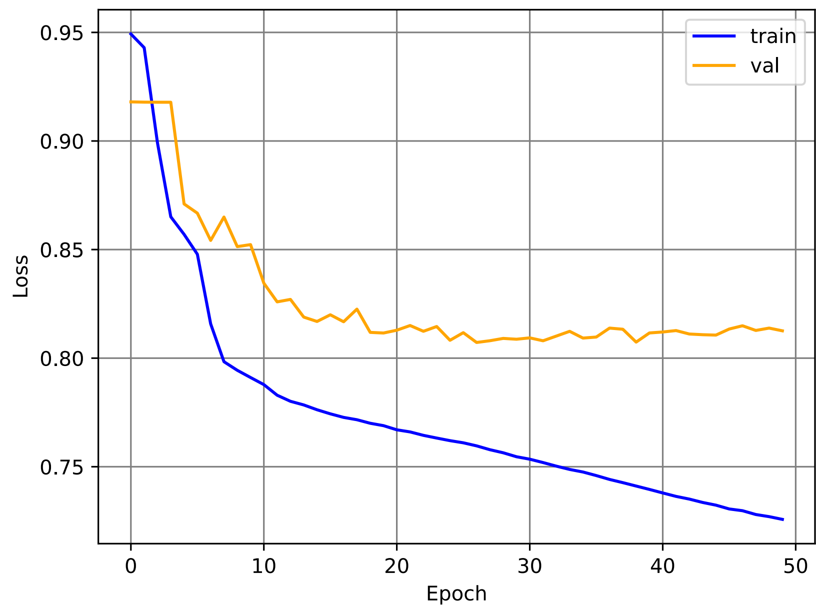

Figure 24: Train & Val Loss vs Epochs of the CNN+LSTM model when k=10 for 50 epochs

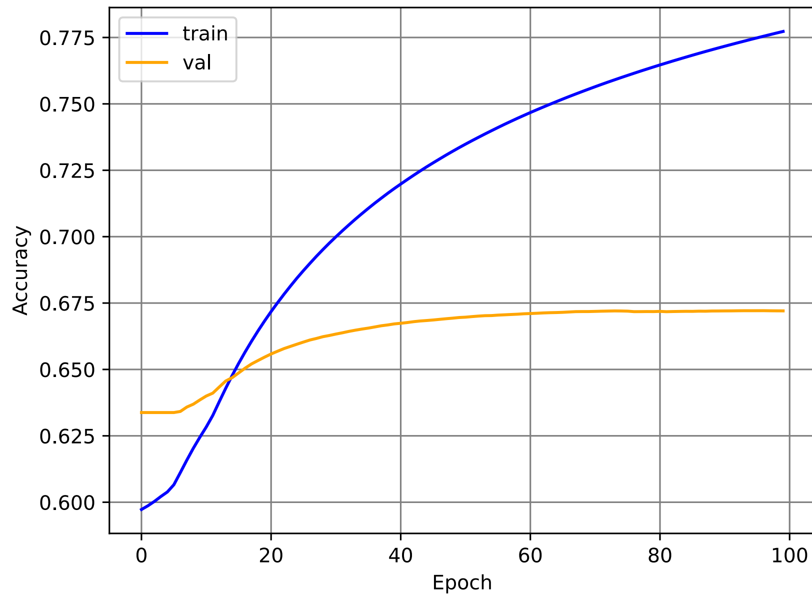

Figure 25: Train & Val Accuracy vs Epochs of the Transformer model when k=10 for 100 epochs

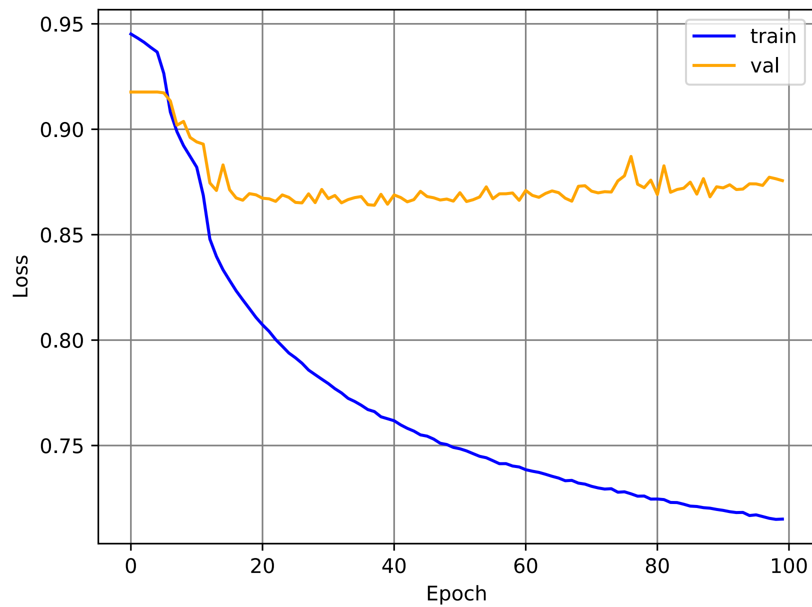

Figure 26: Train & Val Loss vs Epochs of the Transformer model when k=10 for 100 epochs

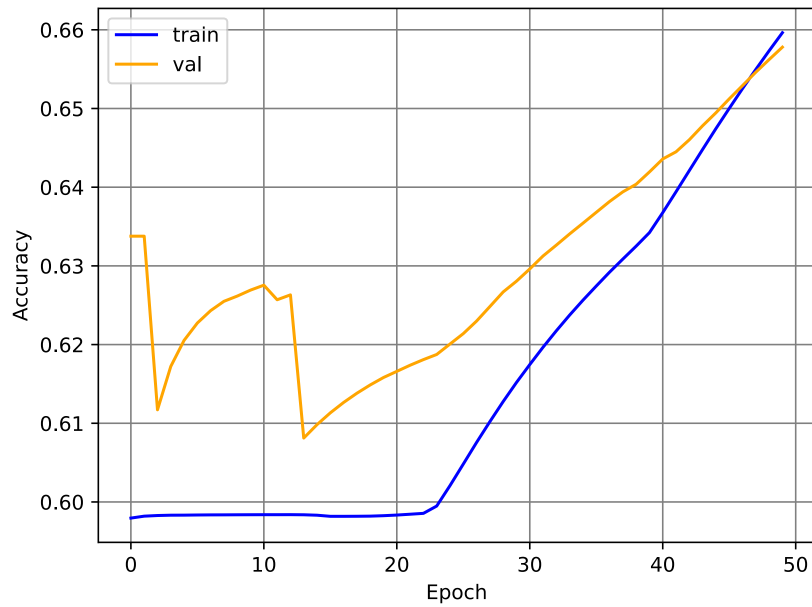

Figure 27: Train & Val Accuracy vs Epochs of the CNN+Transformer model when k=10 for 50 epochs

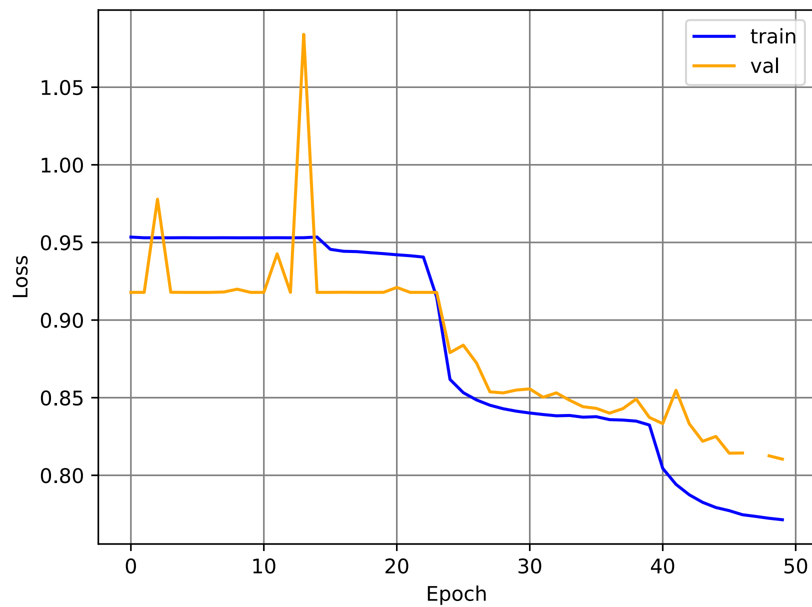

Figure 28: Train & Val Loss vs Epochs of the CNN+Transformer model when k=10 for 50 epochs

All the models were trained again up until the point where they reached the minimum value of the Validation Loss. The results are shown below.

Model	Accuracy	Precision	Recall	F1
R Regression	0.48	0.42	0.44	0.41
CNN+LSTM	0.80	0.81	0.82	0.81
Transformer	0.77	0.75	0.77	0.74
CNN+Transformer	0.82	0.81	0.82	0.81

Table 7: Results of the predictions for k=10 [11]

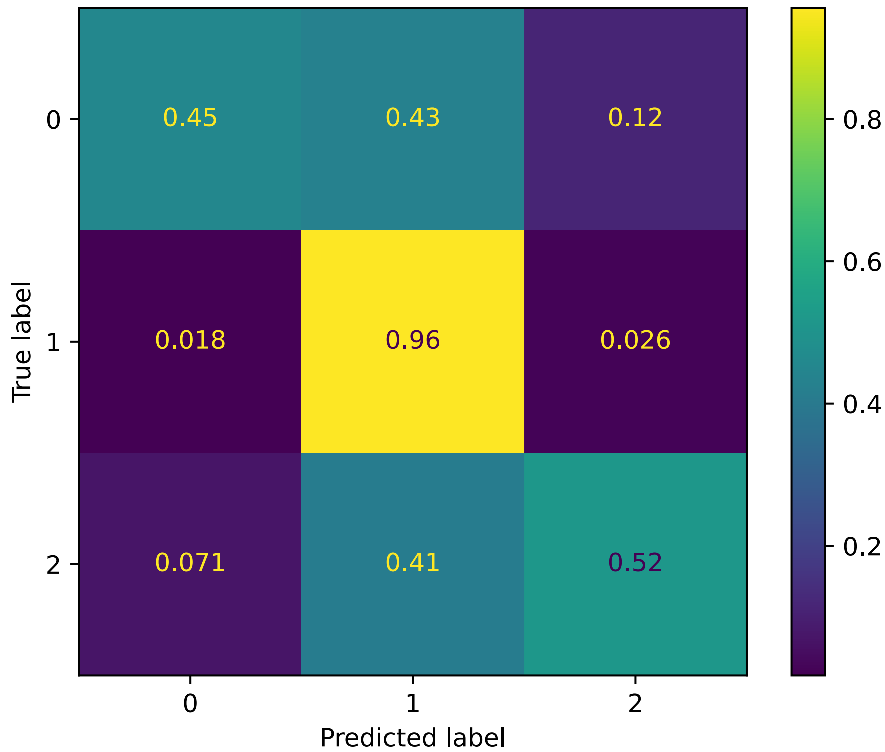

Figure 29: Confusion matrix for the CNN+LSTM model when k=10

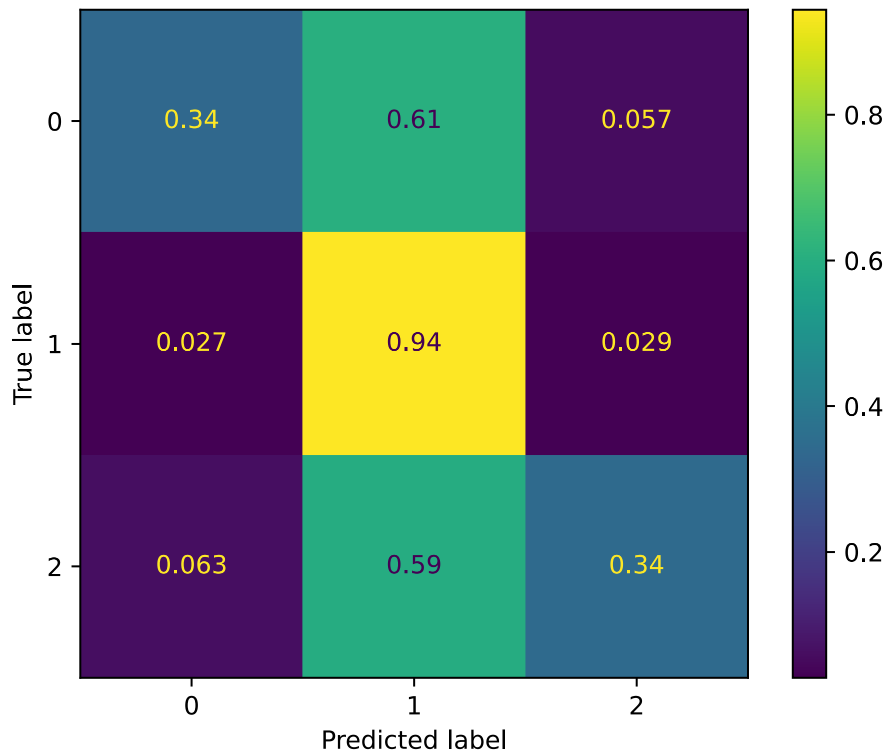

Figure 30: Confusion matrix for Transformer model when k=10

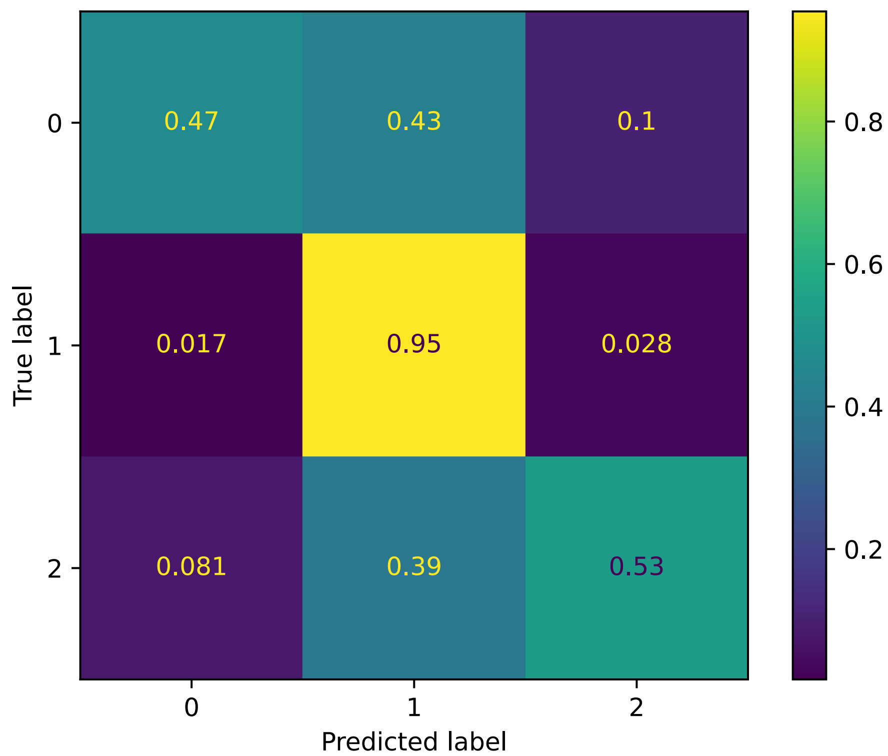

Figure 31: Confusion matrix for CNN+Transformer model when k=10

5.4.2 Case k=20

The data distribution is shown in the following table.

	labeled 0	labeled 1	labeled 2	Total
Train data	52,535 (25.6%)	99,043 (48.6%)	52,221 (25.8%)	203,799
Val data	12,074 (23.7%)	26,817 (52.6%)	12,058 (23.7%)	50,949
Test data	27,470 (19.7%)	86,617 (62.1%)	25,499 (18.2%)	139,586

Table 8: Data distribution for k=20

We show here the different variations of Training & Validation Accuracy and Training & Validation Loss of each of the 3 models and their evolution for a large number of epochs.

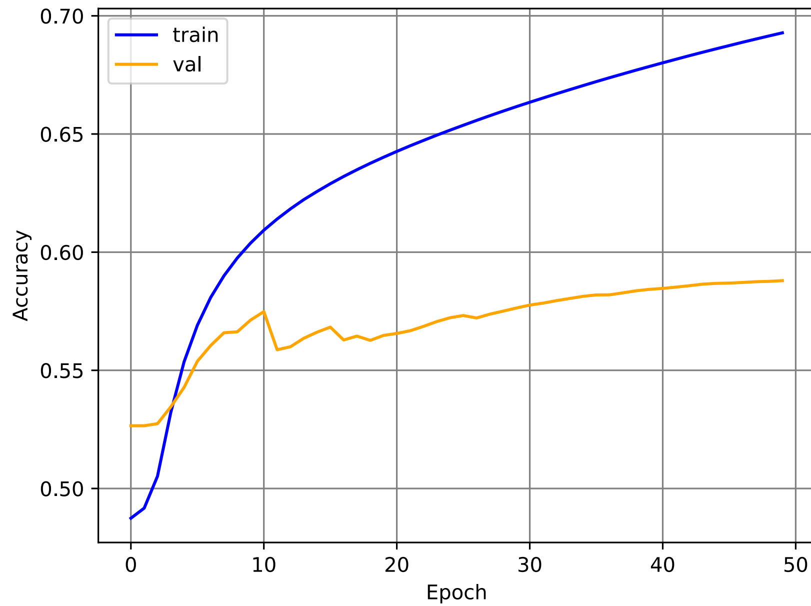

Figure 32: Train & Val Accuracy vs Epochs of the CNN+LSTM model when k=20 for 50 epochs

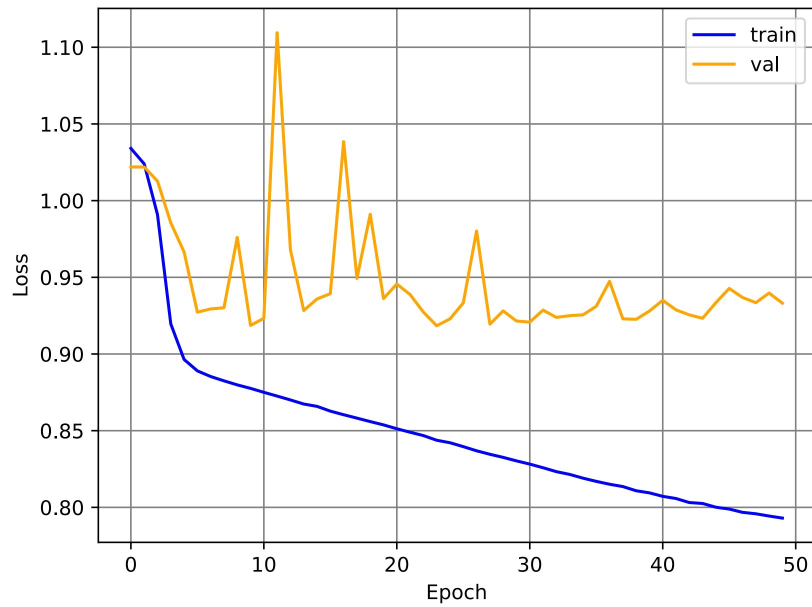

Figure 33: Train & Val Loss vs Epochs of the CNN+LSTM model when k=20 for 50 epochs

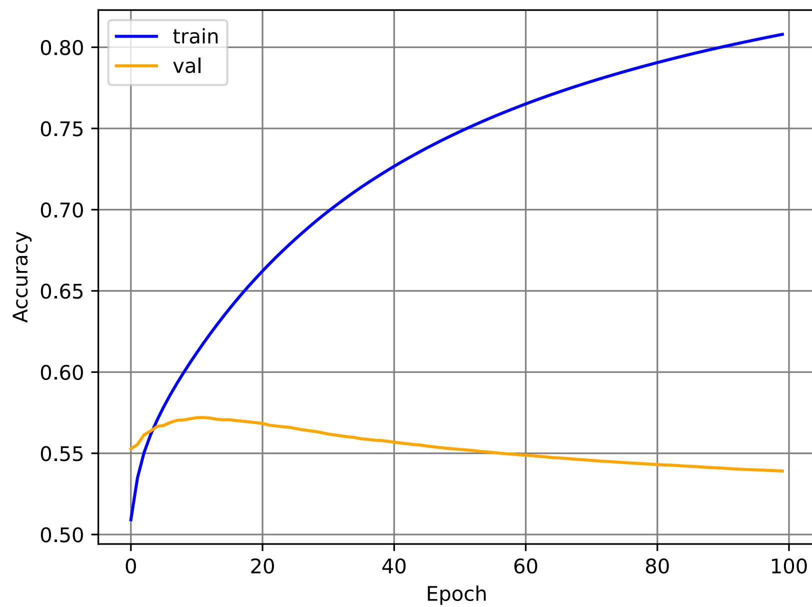

Figure 34: Train & Val Accuracy vs Epochs of the Transformer model when k=20 for 100 epochs

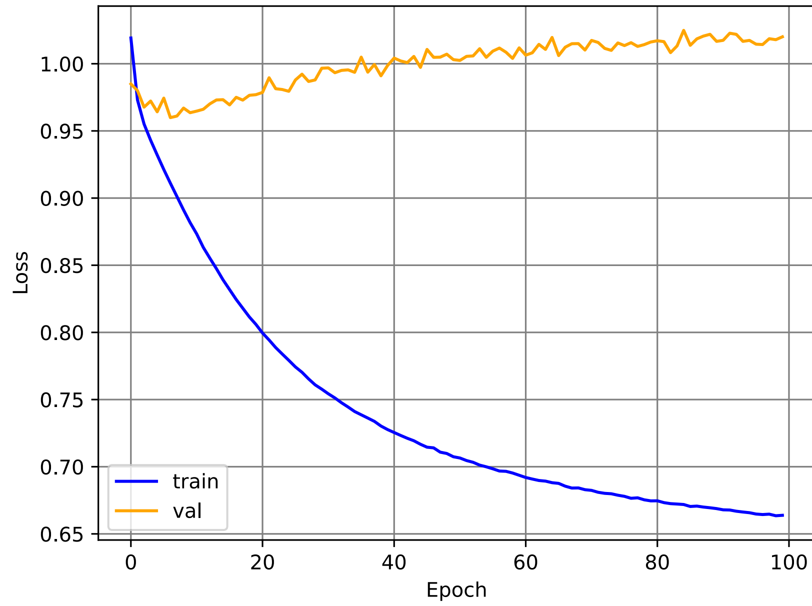

Figure 35: Train & Val Loss vs Epochs of the Transformer model when k=20 for 100 epochs

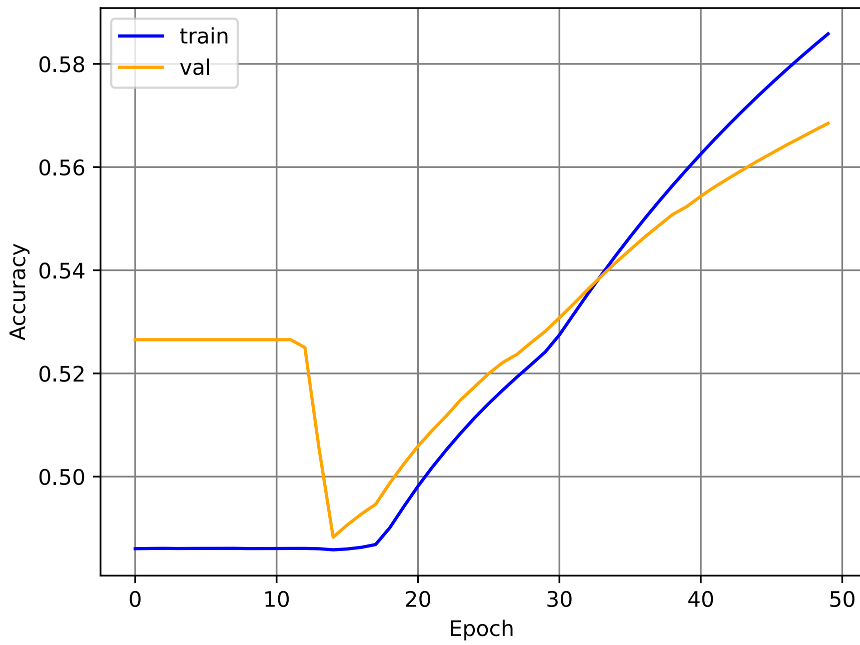

Figure 36: Train & Val Accuracy vs Epochs of the CNN+Trans model when k=20 for 50 epochs

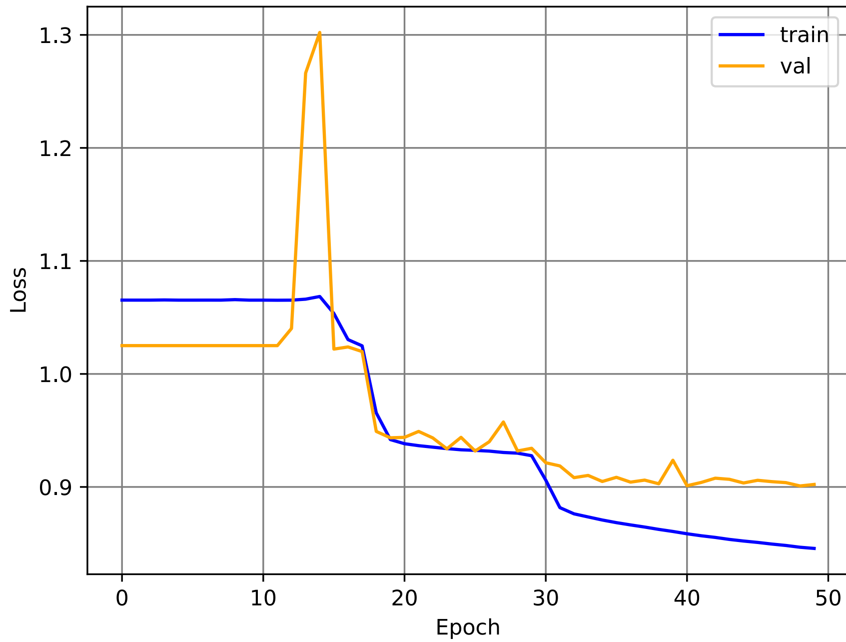

Figure 37: Train & Val Loss vs Epochs of the CNN+Trans model when k=20 for 50 epochs

All the models were trained again up until the point where they reached the minimum value of the Validation Loss. The results are shown below.

Model	Accuracy	Precision	Recall	F1
R Regression	0.50	0.44	0.44	0.44
CNN+LSTM	0.73	0.72	0.73	0.71
Transformer	0.67	0.64	0.67	0.63
CCN+Transformer	0.74	0.72	0.74	0.72

Table 9: Results of the predictions for k=20 [11]

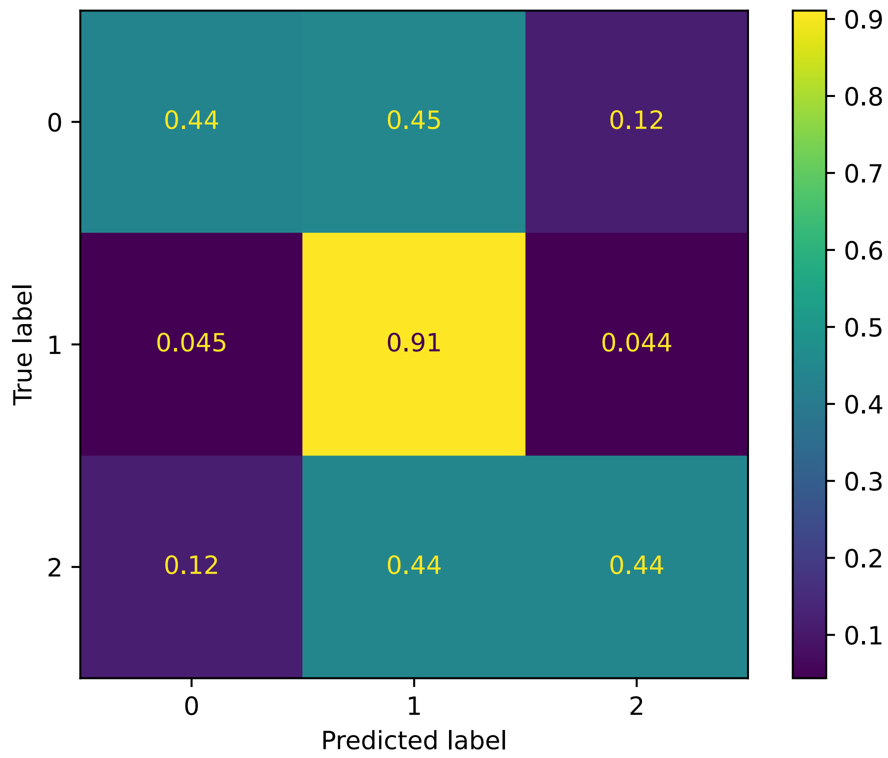

Figure 38: Confusion matrix for the CNN+LSTM model when k=20

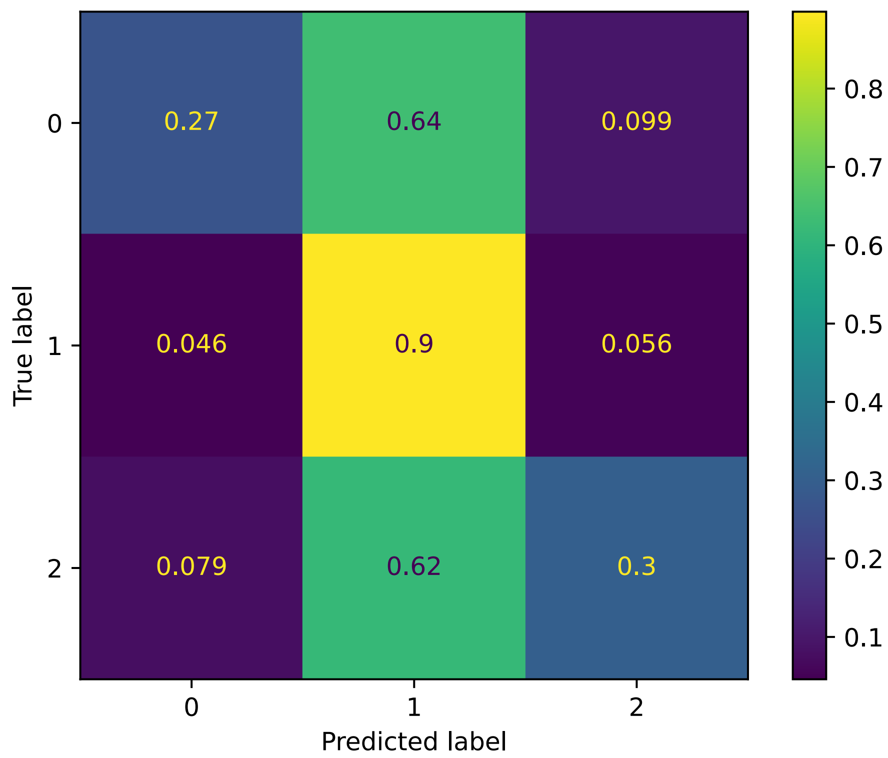

Figure 39: Confusion matrix for the Transformer model when k=20

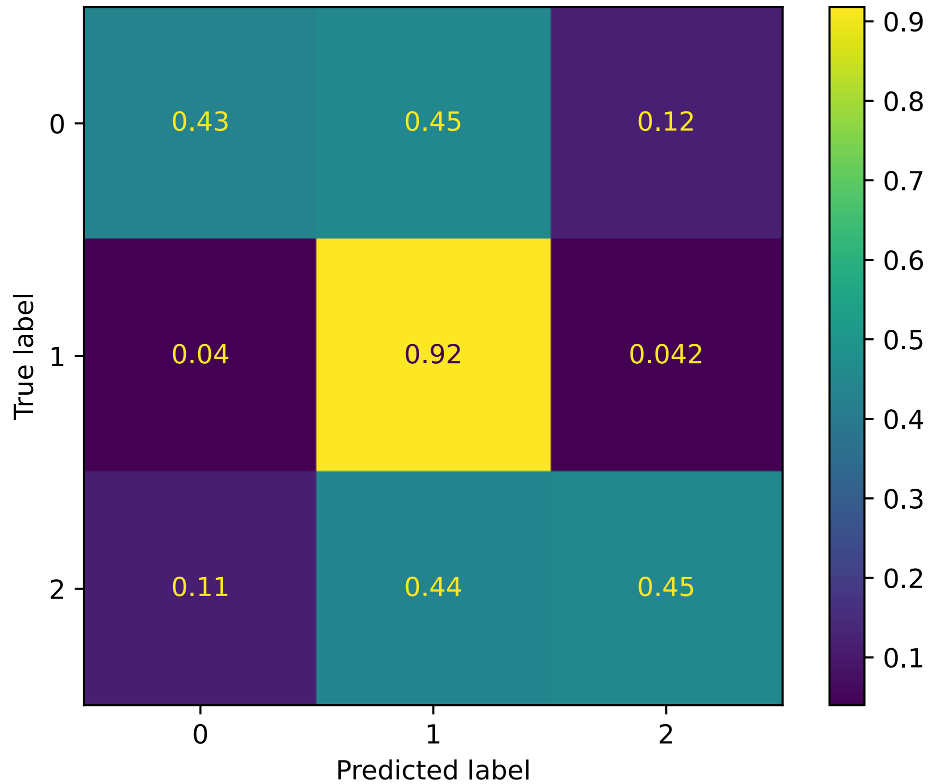

Figure 40: Confusion matrix for the CNN+Transformer model when k=20

5.4.3 Case k=50

The data distribution is shown in the following table.

	labeled 0	labeled 1	labeled 2	Total
Train data	71,482 (35.1%)	61,826 (30.3%)	70,491 (34.6%)	203,799
Val data	16,123 (31.6%)	18,846 (37.0%)	15,980 (31.4%)	50,949
Test data	38,467 (27.6%)	66,006 (47.3%)	35,113 (25.1%)	139,586

Table 10: Data distribution for k=50

We show here the different variations of Training & Validation Accuracy and Training & Validation Loss of each of the 3 models and their evolution for a large number of epochs.

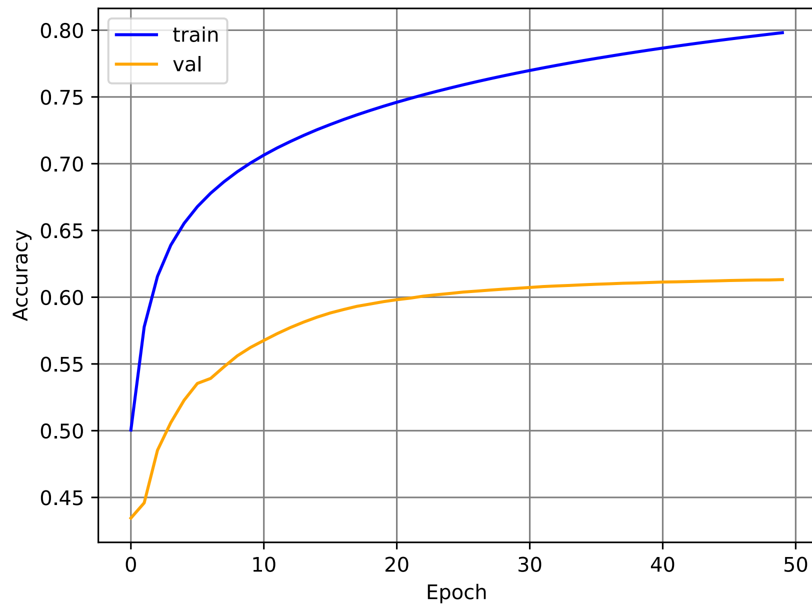

Figure 41: Train & Val Accuracy vs Epochs of the CNN+LSTM model when k=50 for 50 epochs

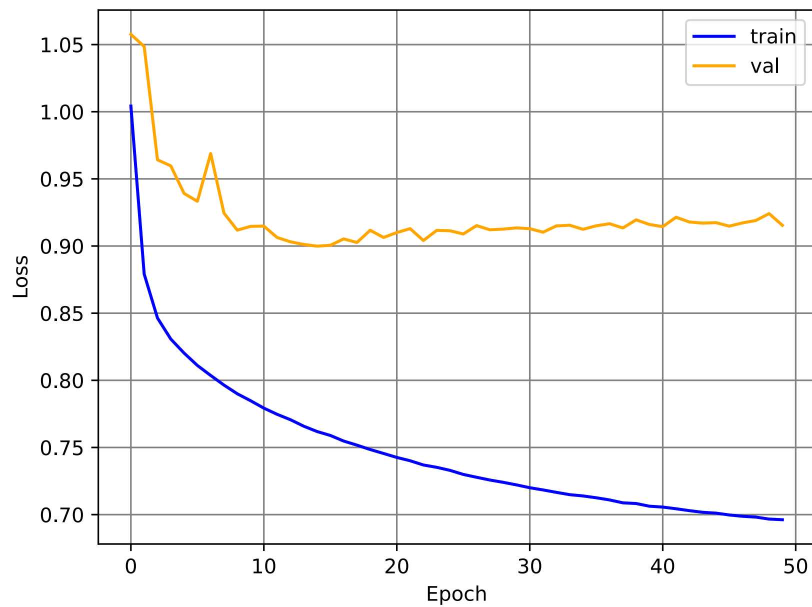

Figure 42: Train & Val Loss vs Epochs of the CNN+LSTM model when k=50 for 50 epochs

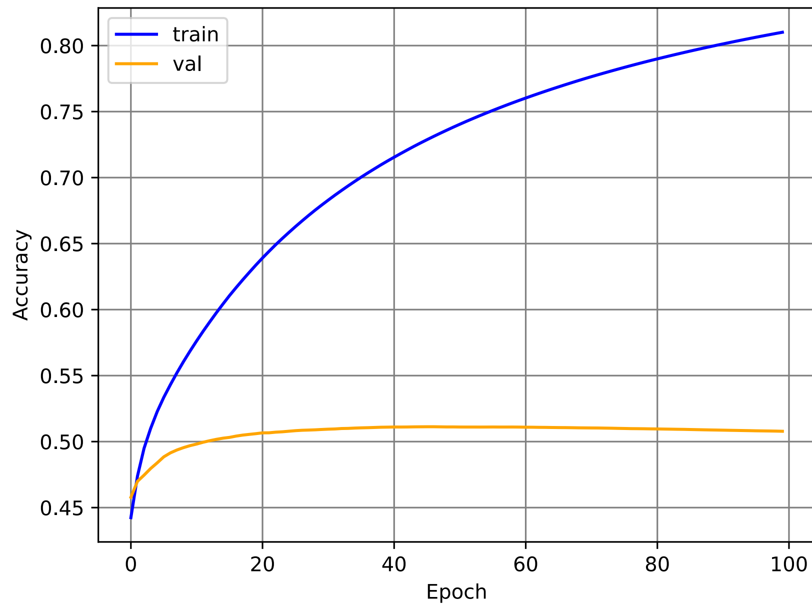

Figure 43: Train & Val Accuracy vs Epochs of the Transformer model when k=50 for 100 epochs

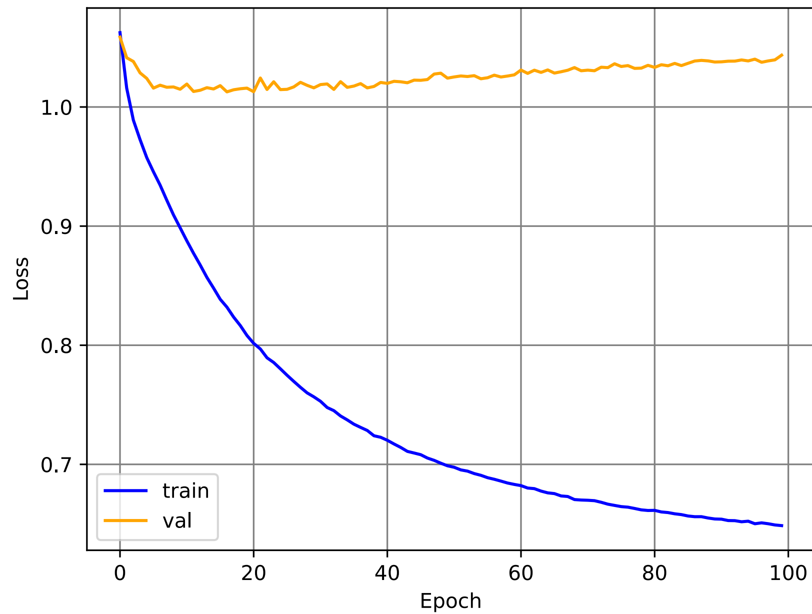

Figure 44: Train & Val Loss vs Epochs of the Transformer model when k=50 for 100 epochs

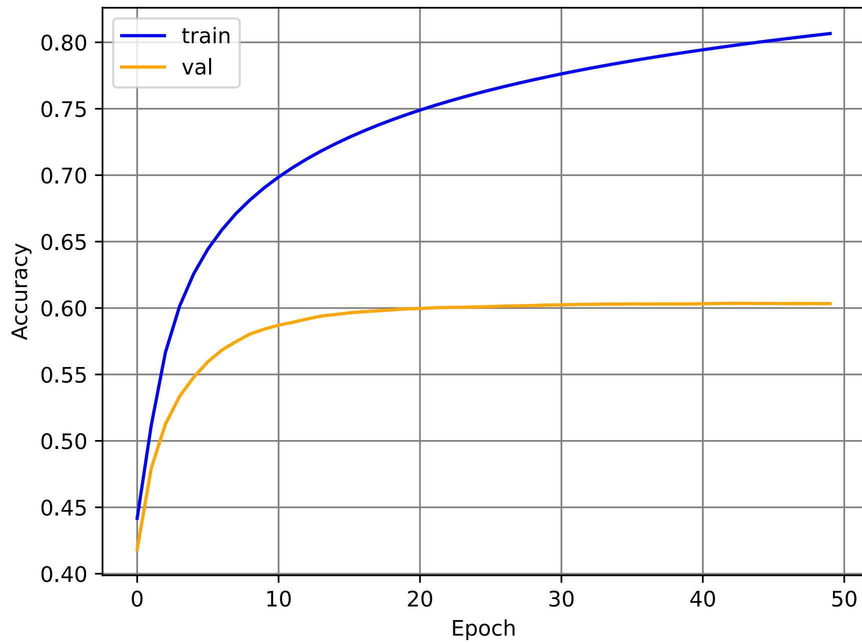

Figure 45: Train & Val Accuracy vs Epochs of the CNN+Trans model when k=50 for 50 epochs

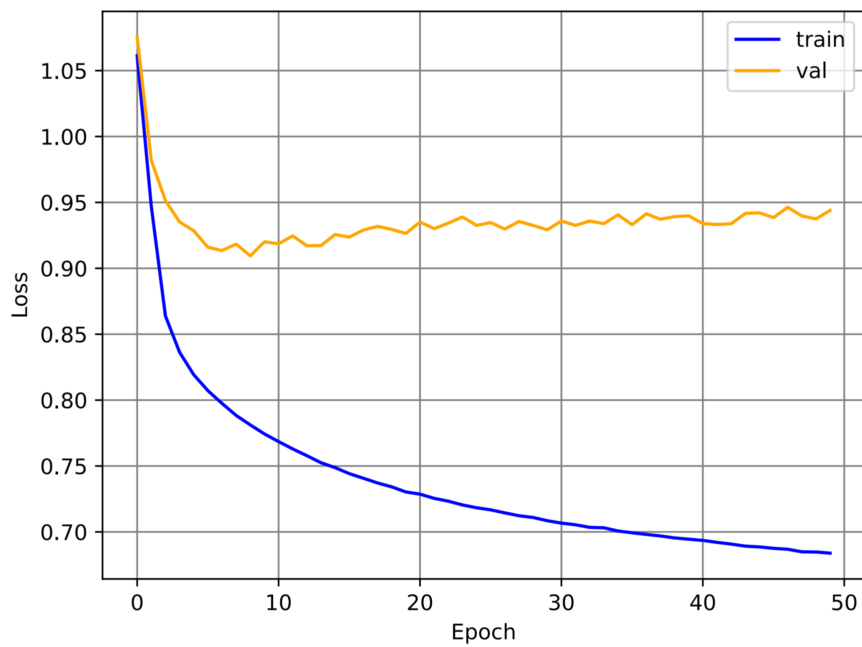

Figure 46: Train & Val Loss vs Epochs of the CNN+Trans model when k=50 for 50 epochs

All the models were trained again up until the point where they reached the minimum value of the Validation Loss. The results are shown below.

Model	Accuracy	Precision	Recall	F1
R Regression	0.44	0.44	0.43	0.43
CNN+LSTM	0.75	0.75	0.75	0.75
Transformer	0.60	0.60	0.60	0.59
CNN+Transformer	0.73	0.73	073	0.73

Table 11: Results of the predictions for k=50 [11]

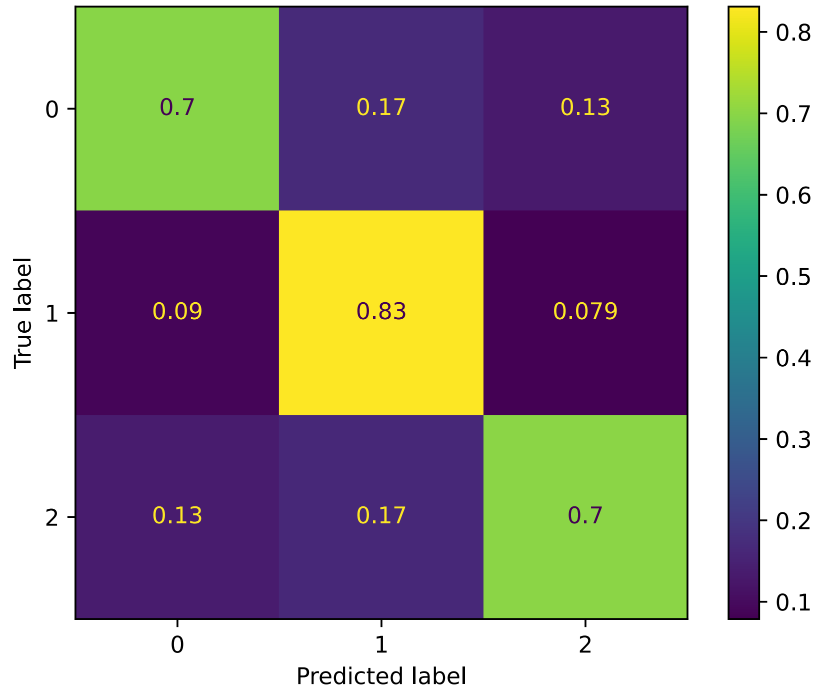

Figure 47: Confusion matrix for the CNN+LSTM model when k=50

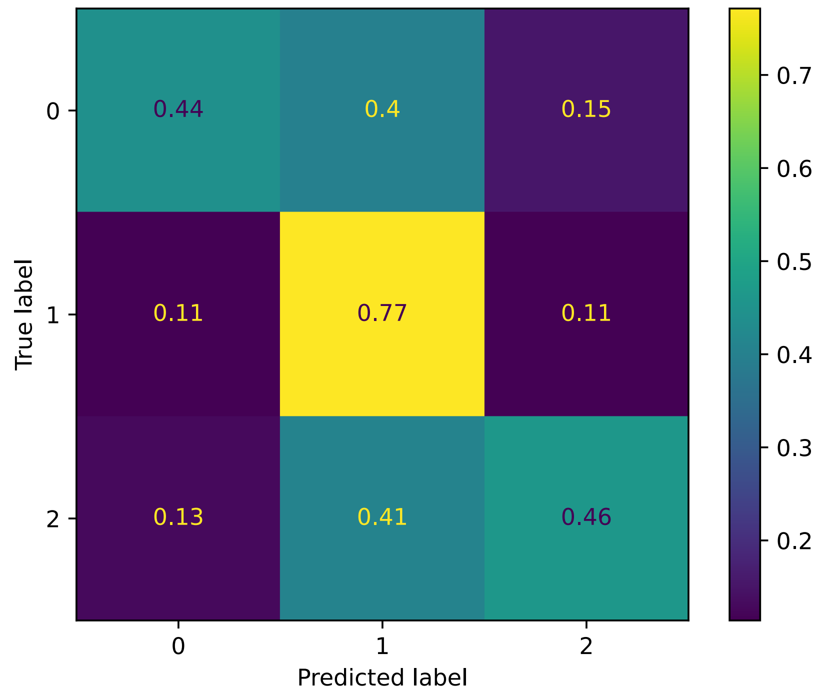

Figure 48: Confusion matrix for the Transformer model when k=50

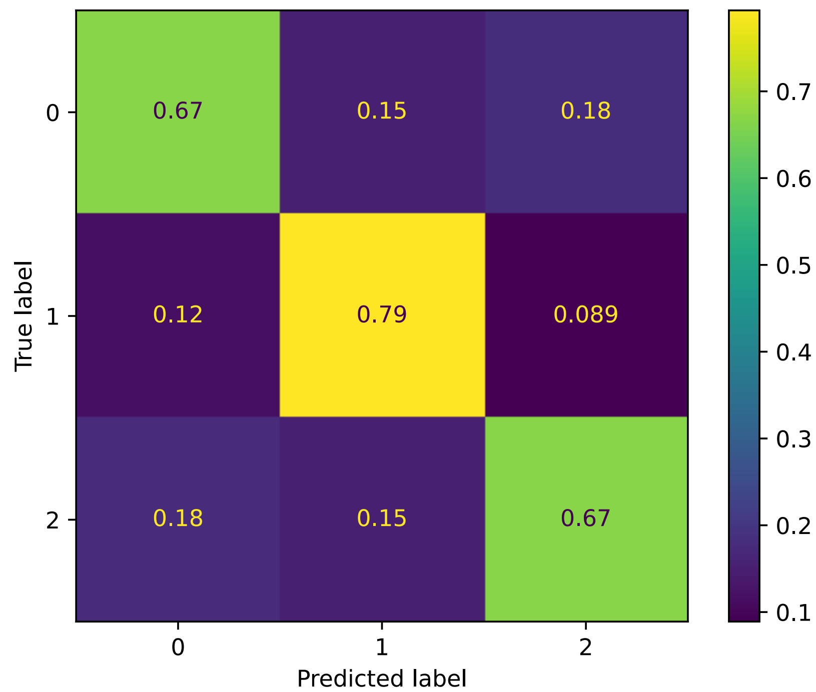

Figure 49: Confusion matrix for the CNN+Trans model when k=50

5.4.4 Case k=100

The data distribution is shown in the following table.

	labeled 0	labeled 1	labeled 2	Total
Train data	84,486 (41.5%)	36,221 (17.8%)	83,092 (40.7%)	203,799
Val data	18,981 (37.2%)	13,214 (26.0%)	18,754 (36.8%)	50,949
Test data	47,976 (34.4%)	48,059 (34.4%)	43,551 (31.2%)	139,586

Table 12: Data distribution for k=100

We show here the different variations of Training & Validation Accuracy and Training & Validation Loss of each of the 3 models and their evolution for a large number of epochs.

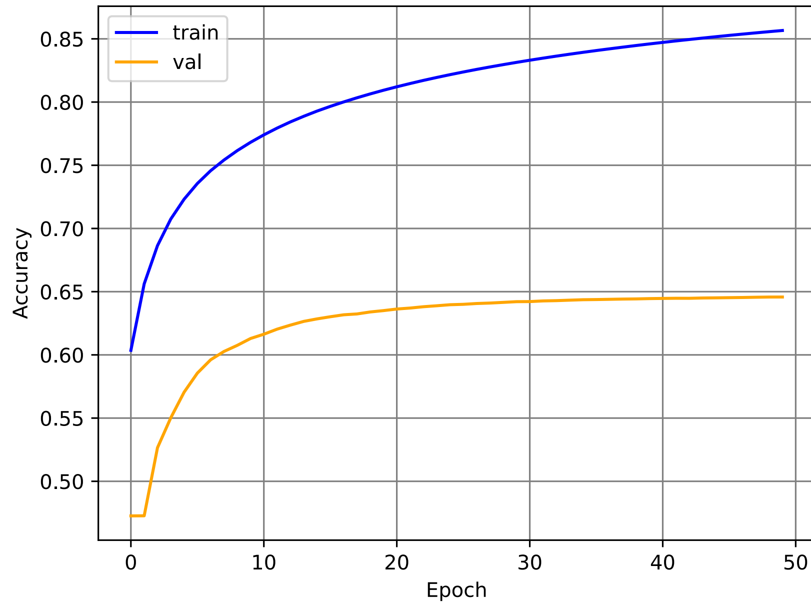

Figure 50: Train & Val Accuracy vs Epochs of the CNN+LSTM model when k=100 for 50 epochs

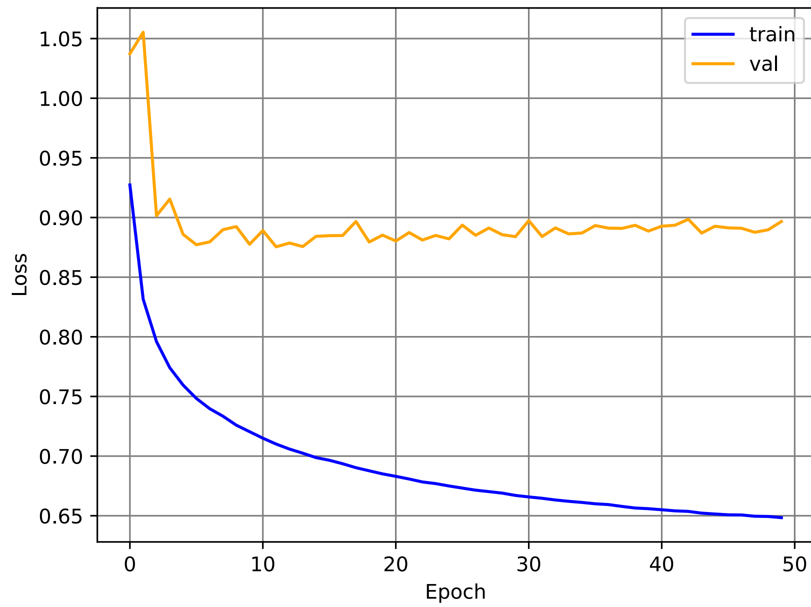

Figure 51: Train & Val Loss vs Epochs of the CNN+LSTM model when k=100 for 50 epochs

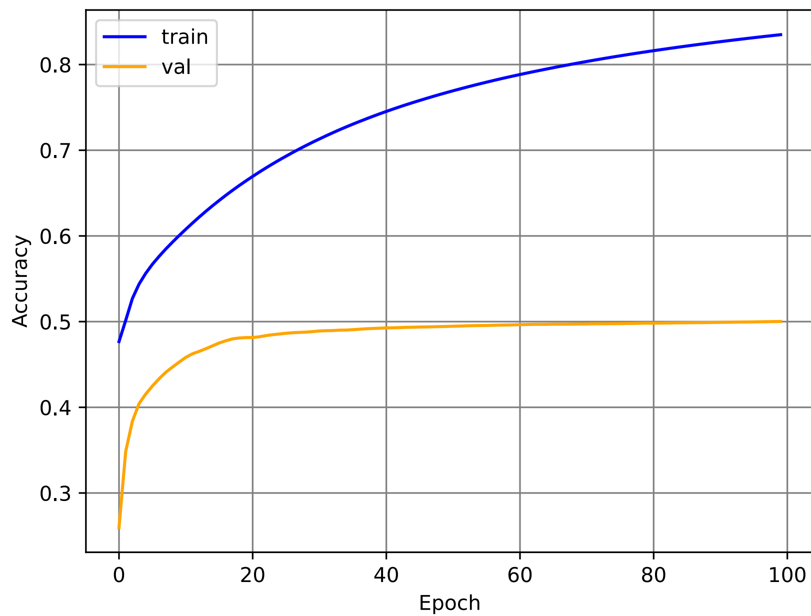

Figure 52: Train & Val Accuracy vs Epochs of the Transformer model when k=100 for 100 epochs

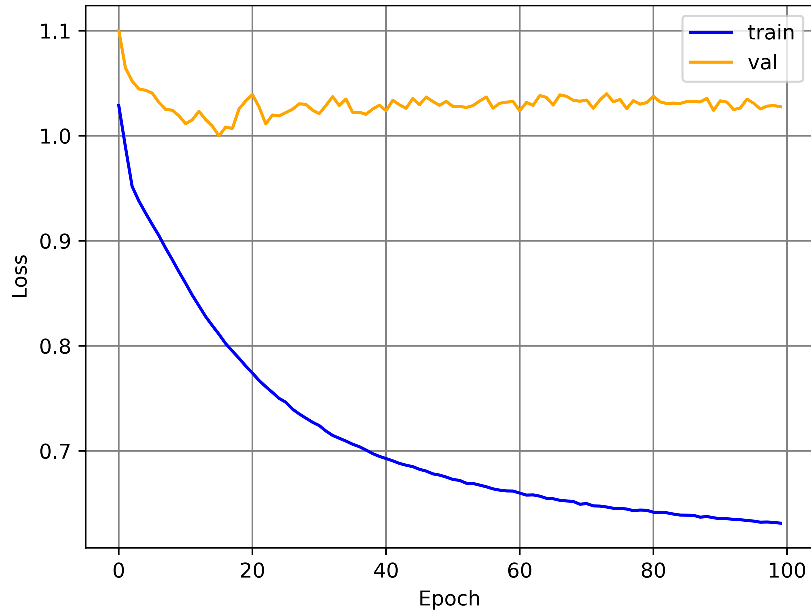

Figure 53: Train & Val Loss vs Epochs of the Transformer model when k=100 for 100 epochs

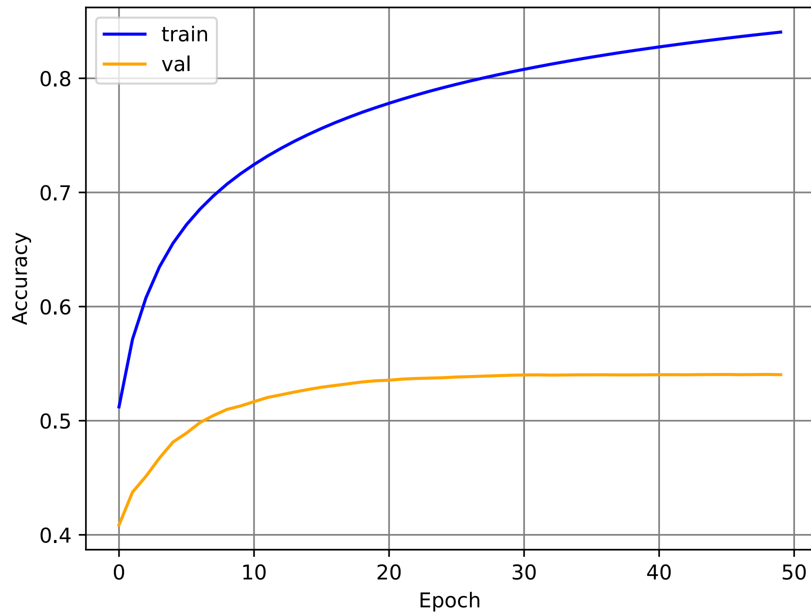

Figure 54: Train & Val Accuracy vs Epochs of the CNN+Trans model when k=100 for 50 epochs

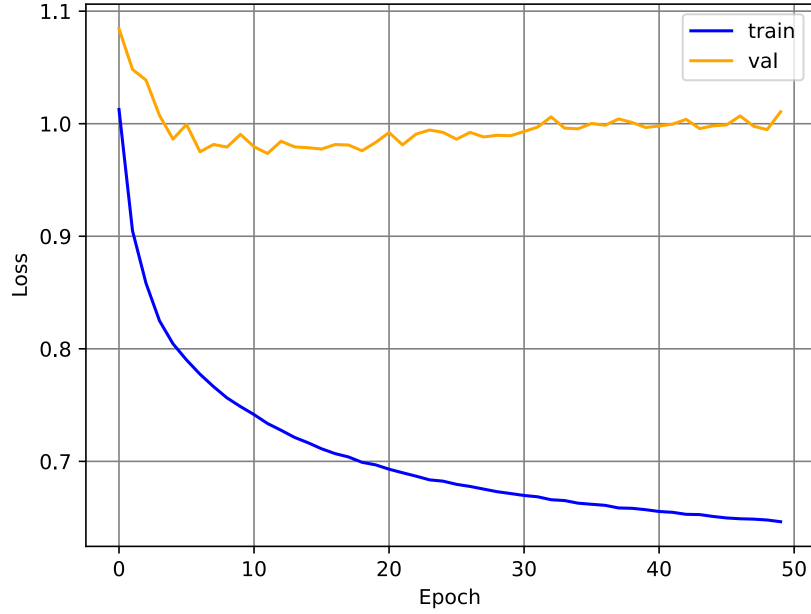

Figure 55: Train & Val Loss vs Epochs of the CNN+Trans model when k=100 for 50 epochs

All the models were trained again up until the point where they reached the minimum value of the Validation Loss. The results are shown below.

Model	Accuracy	Precision	Recall	F1
R Regression	0.43	0.43	0.43	0.42
CNN+LSTM	0.74	0.74	0.74	0.74
Transformer	0.56	0.57	0.56	0.56
CNN+Transformer	0.72	0.72	0.72	0.72

Table 13: Results of the predictions for k=100 [11]

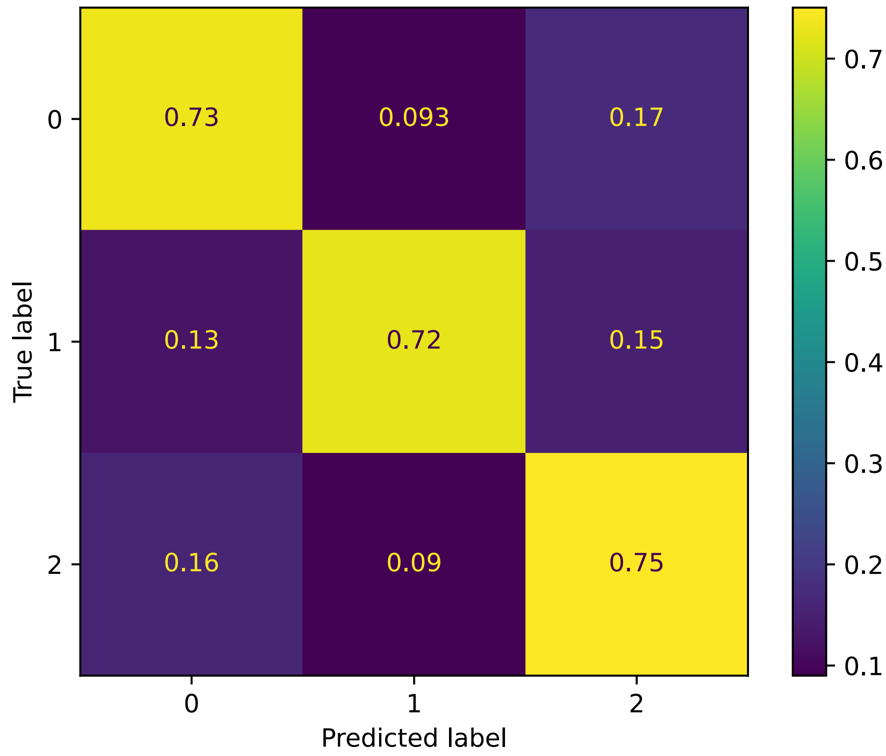

Figure 56: Confusion matrix for the CNN+LSTM model when k=100

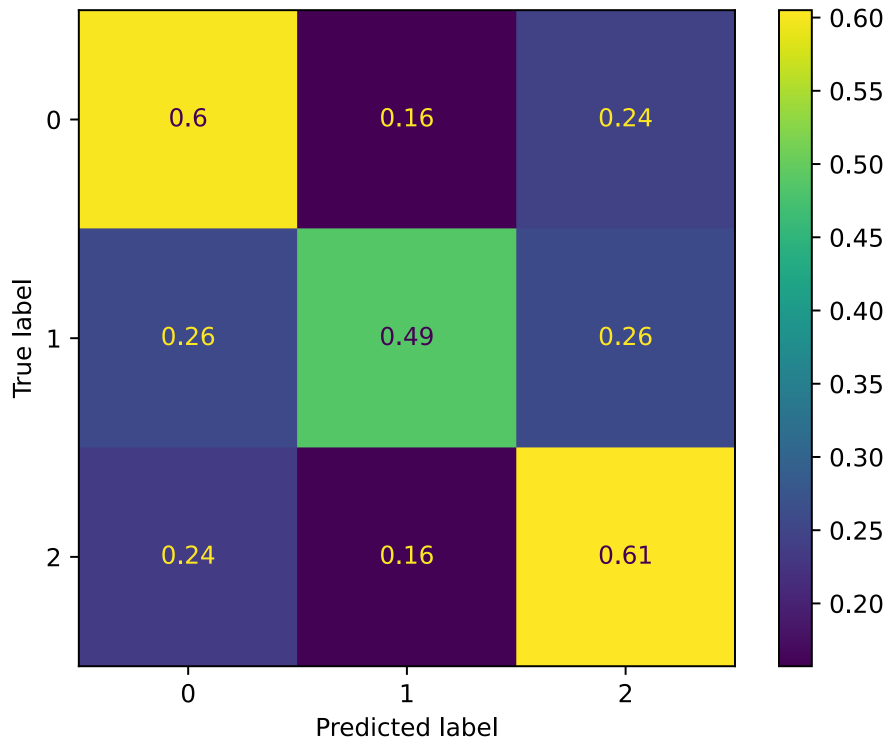

Figure 57: Confusion matrix for the Transformer model when k=100

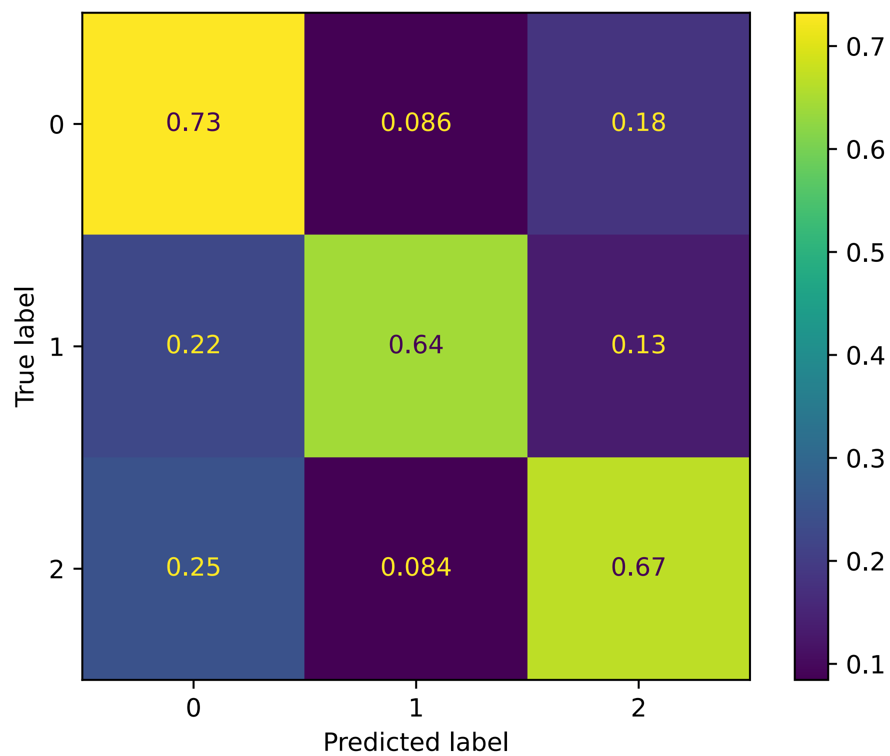

Figure 58: Confusion matrix for the CNN+Transformer model when k=100

5.5 Interpretation of Results

We will go case by case in the interpretation of results. Firstly, we must notice that all three Deep Learning models clearly and vastly out-perform the Ridge Regression. Also, in order to help with the interpretation and analysis, we present the following 2 figures, which display the evolution of the accuracy (Figure 59) and the evolution of the F1 score (Figure 60) through the horizons.

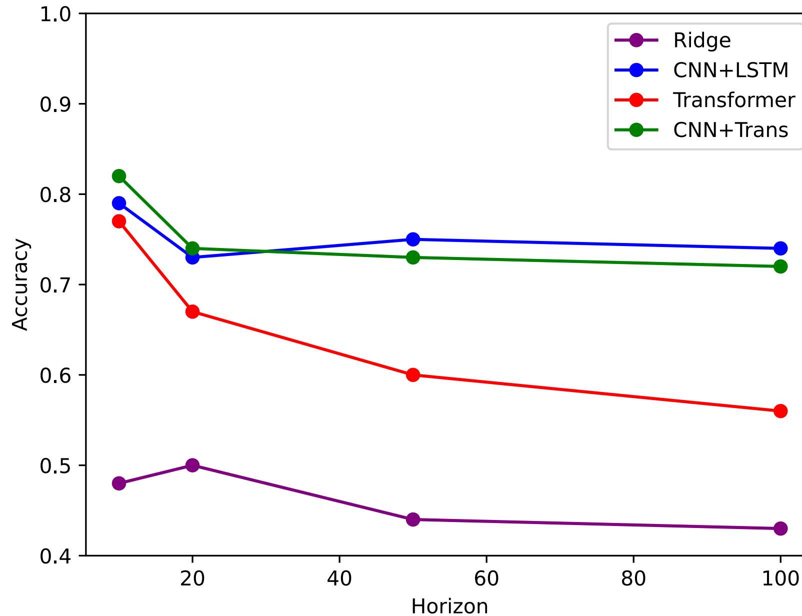

Figure 59: Accuracy vs horizon for each of the models presented

In the case $k = 10$, we see that the three Deep Learning models (CNN+LSTM, Transformer and CNN+Transformer) show very similar results in all measures. The CNN+Transformer architecture shows the highest accuracy of them all. The behavior of the networks is very similar as well, we can see in their Confusion Matrices in Figures 29, 31 and 30 that they usually predict "1" (steady mid-price) but they confidence in the prediction is low. They predict "0" and "2" less often, but, when they do, it is extremely likely that the prediction will be correct, especially with the Transformer model, as the other models miss when predicting "2".

In the case $k = 20$, we can see that, again, the CNN+Transformer model reaches the highest accuracy and F1 score, followed closely by the CNN+LSTM model. The Transformer model doesn't achieve the same accuracy score as with the previous horizon, similarly with the F1 score. Nonetheless, we see a similar behavior in the Confusion Matrices as with the previous horizon. As we can see in Figures 38, 39 and 40, all the models predict most of the time the stability of the mid-price, and then fewer times its movement. We can see, however, that when the Transformer model predicts movement, it is very likely that the prediction is correct, while the other models mis-predict "0" and "2" more often.

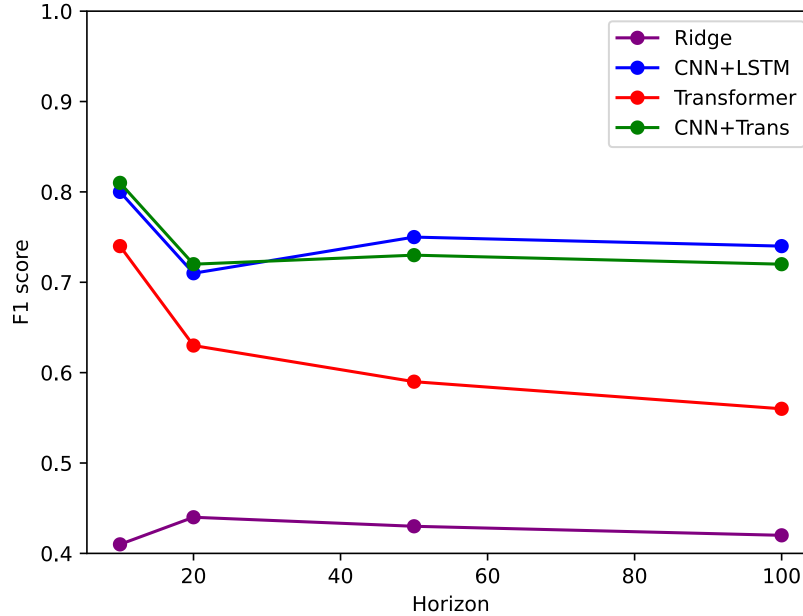

Figure 60: F1 score vs horizon for each of the models presented

Moving to the case $k = 50$, the CNN+LSTM model achieves the highest accuracy of the three Deep Learning models, only two percentage points ahead of the CNN+Transformer model. Here, we can not say the same thing as with the previous horizons, as the Transformer model doesn't display any superiority in either Accuracy or predicting mid-price movement. Here, from Figures 47, 48 and 49, we see that the CNN+LSTM model just predicts each outcome with higher reliability, slightly better than the CNN+Trans model.

Finally, the case $k = 100$. Here, the CNN+LSTM model shows the highest accuracy and F1 score, slightly better than the CNN+Transformer model and ahead of the Transformer model. From Figures 56, 57 and 58, The CNN+LSTM model better predicts each output, while the CNN+Transformer is slightly worse, while the Transformer model does not come close (its accuracy at predicting "1" is less than 0.5).

6 Conclusions and further work

In this paper, we have first introduced the Transformer architecture from the Natural Language Processing perspective, so as to understand the architecture from its original objective.

Then, we have used the Transformer encoder architecture to attempt to predict the hourly closing price of Bitcoin. The attempt, while unsuccessful, does not invalidate the Transformer's strength as a Deep Learning architecture. Rather, it shows that the price of Bitcoin can not be successfully predicted.

Finally, we have used the Transformer encoder architecture to attempt to predict the movement of the virtual mid-price in a Limit Order Book, both by itself and combined with a CNN, showing that the CNN+Transformer outperforms the CNN+LSTM on some horizons.

Thus, we see that, while transformer can be used in HFT, though more research is required on the topic. As future work, we may apply a different approach to the prediction of the closing price of Bitcoin from a different perspective: as the Transformer is excellent at NLP, we could use it to understand tweets and news by influential people in the Cryptos Market to try to predict when the price is going to move. Also, we may apply modifications to the transformer (for example, on the activation functions or the similarity measures) to make it better suited for high-noise data.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. “Attention is all you need”, In *Advances in neural information processing systems*, pages 5998-6008, 2017. arXiv:1706.03762

[2] G. Bebis and M. Georgiopoulos. “Feed-forward neural networks”, In *IEEE Potentials*, vol. 13, no. 4, pp. 27-31, Oct.-Nov. 1994, doi: 10.1109/45.329294.

[3] Li, B., Wang, Z., Liu, H., Du, Q., Xiao, T., Zhang, C., & Zhu, J. “Learning Light-Weight Translation Models from Deep Transformer”, In *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(15), 13217-13225, 2021. https://doi.org/10.1609/aaai.v35i15.17561

[4] M. Phuong, M. Hutter. “Formal Algorithms for Transformers”, 2022. arXiv:2207.09238

[5] Schmitt, Thilo and Chetalova, Desislava and Schäfer, Rudi and Guhr, Thomas. ”Non-stationarity in financial time series: Generic features and tail behavior”, In *EPL (Europhysics Letters)*, 2013. doi: 10.1209/0295-5075/103/58003

[6] Cont, R. “Volatility Clustering in Financial Markets: Empirical Facts and Agent-Based Models”, In *Long Memory in Economics. Springer, Berlin, Heidelberg*, 2007. https://doi.org/10.1007/978-3-540-34625-810

[7] Bradley, Brendan and Taqqu, Murad. “Financial Risk and Heavy Tails”, In *Handbook of Heavy Tailed Distributions in Finance*, 2003. doi: 10.1016/B978-044450896-6.50004-2

[8] https://www.cryptodatadownload.com/data/gemini/

[9] M. D. Gould, M. A. Porter, S. Williams, M. McDonald, D. J. Fenn, and S. D. Howison, “Limit order books,” *Quantitative Finance*, Nov. 2013.

[10] https://etsin.fairdata.fi/dataset/73eb48d7-4dbc-4a10-a52a-da745b47a649

[11] A. Ntakaris, M. Magris, J. Kanniainen, M. Gabbouj, and A. Iosifidis, “Benchmark dataset for mid-price forecasting of limit order book data with machine learning methods”, 2017. doi: 10.1002/for.2543.

[12] Z. Zhang, S. Zohren, and S. Roberts, “DeepLOB: Deep convolutional neural networks for limit order books”, 2018. doi: 10.1109/TSP.2019.2907260.

[13] J. Wallbridge. “Transformers for Limit Order Books”. arXiv:2003.00130