D.A.V. Public School Airoli

Computer Science Project Report

2021-2022

Project Stonks

Aditya Gandhi
Class: XII B
Board roll: 15607802

13 March 2022

Certificate

2021-2022

This is to certify that Master ***Aditya Jayesh Gandhi*** of XII, D.A.V. Public School, Airoli, has successfully completed the project entitled ***Project Stonks*** using Python during the academic year ***2021-2022 .***

It is further certified that the project is the genuine work of the student and has been done sincerely and satisfactorily.

Internal Examiner's Signature	External Examiner's Signature	Principal's Signature

Computer Science Project Report

Aditya Gandhi, Nisarg Jain

March 2022

Contents

Chapter 1

Synopsis

Project Stonks was initially pitched as a tool for demystifying the stock market and making it more accessible and generate interest for it in Gen Z. While the project failed to meet that aim, it still remains a great stock screening tool that someone with minimal knowledge of the stock market can play around with in order to test out things such as trends and the use of *Artificial Intelligence* and *Deep Learning* in the field of stock investments.

We did thorough research about both, the stock market and artificial intelligence before finalising upon this project and we understand very well that it is impossible to predict the stock market fluctuations using any kind of artificial intelligence or machine learning models. However we stuck to the idea solely for experimental purposes as we hoped to learn more about how deep learning models react to unpredictable datasets and situations such as stock market fluctuations.

1.1 The Backend

Like most applications, ours has two main parts, the frontend and the backend. The backend of the application is divided further into two more parts

- The deep learning model
- The data retrieval scripts

Both of these will be elaborated upon later in the report. The backend was primarily worked upon by Aditya Gandhi with some help from external parties.[1]

1.2 The Frontend

The frontend was mentioned in our project synopsis however the plans were altered quite a bit following the submission of the synopsis. the frontend no longer seeks to use React JS to deploy our project as web application instead it uses python's Kivy framework to deploy as a desktop application. Nisarg Jain was primarily responsible for the programming of the frontend with the designing aspect being the responsibility of Aditya Gandhi. The frontend will also be elaborated upon in the later parts of this report.

[1] check the special thanks section in the bibliography for details

Chapter 2

Backend

[1] The backend, as previously discussed, is divided into two main parts, this chapter will address both of them. The backend is written purely in python and uses several external libraries to accomplish its goals.

The libraries used for the backend are:

1. yfinance
2. os
3. datetime
4. Ipython
5. matplotlib
6. seaborn
7. numpy
8. pandas
9. tensorflow-keras
10. scikit learn
11. beautiful soup
12. requests

2.1 StonkNet

this was the first part of the project that entered development. StonkNet is comprised of two deep learning models, one is responsible for powering the daily prediction tool and one for the weekly predictions tool.

The *LSTM (Long Short Term Memory)* class of the *RNN (Recurrent Neural Network)* architecture was chosen for both of these models. The reason *LSTM* was chosen is because our initial research concluded that *LSTM* was the best choice for a sequential modelling problem such as ours.

2.1.1 RNN

[2] *Recurrent Neural Networks* are great for processing sequential data like natural language or in our case, Time series. the reason for that is because they have a short memory which allows them to consider all previous elements of a sequence when making predictions on a specific time step.

The way they do this is relatively simple, when an input vector x is passed into the *RNN* it builds a *'hidden state'* which is a function of the input vector and the previous hidden state ,this hidden state is given by:

$$h_t = \tanh(W_{hh}^T \; h_{t-1} + W_{xh}^T \; x_t) \tag{2.1}$$

where W_{hh}^T and W_{hx}^T are the respective weight matrices for h_{t-1} and x_t

and then the model predicts an output vector which is given by:

$$\hat{y}_t = W_{hy}^T \; h_t \tag{2.2}$$

where W_{hy}^T is the weight matrix for the output vector. this is the way an RNN cell updates per time step, as with all common deep learning models, a loss function is computed based off the predictions and ground truths. In our case the loss function was *Mean Squared Error(MSE)* and the accuracy metric was *Mean Absolute Error(MAE)*

While the *RNN* is a relatively good architecture for simple sequential problems, for problems such as ours where long term effects of the fluctuations in the time series at a given set of time steps must be taken into account, the *RNN* architecture begins to falter. This is because of the problem of *vanishing gradients*. To account for this problem we choose to use the popular *LSTM* architecture.

[1] This section of the report assumes some knowledge of statistical and Machine Learning terminology, however some information may be found in the Appendices

2.1.2 LSTM

I won't really go too deep into the theory for the *LSTM* architecture but they are relatively simple. Its basically a modified *RNN* where there's a second cell state c_t along with h_t and this c_t is comprised of these "gates" which can regulate the flow of information into the cell.

There's 3 main kinds of "gates" each responsible for a particular kind of regulation, these regulations can be labelled as:

- forgetting data selectively
- taking in data selectively
- outputting data selectively

They do so by some math that I wont go into but its very probabilistic and uses the Sigmoid function for it. These gated cells makes it so that the effects of fluctuations at a particular or set of time steps isn't lost even at another time step, which may be far ahead in the future

LSTMs also allow for uninterrupted gradient flow during backpropagation through time due to this property of selective cell state updates.

2.1.3 The Dataset

[3] Since this is a supervised machine learning problem we needed large amounts of data to achieve good model performance , the dataset we used comprised of 25 features across 5700+ time steps it covered around 15 years of daily stock data. We attained the daily stock data using the yfinance module

The features that we used are as follows:

- Opening prices of Goldman Sachs
- Closing prices of Goldman Sachs
- High prices of Goldman Sachs
- Volume of Goldman Sachs
- Low prices of Goldman Sachs
- Adj Close prices of Goldman Sachs
- Closing prices of Morgan Stanley and JP Morgan
- Closing prices of composite indices like NASDAQ, Bombay Stock Exchange, Nikkei225 and the New York Stock Exchange
- Closing prices of the Volatility Index
- Technical Indicators for Goldman Sachs such as 20 day moving average, 200 day moving average, 26 day exponential moving average, 12 day exponential moving average, Moving Average Convergence Divergence, Signal Line, Momentum, Bollinger bands.
- and lastly, Fourier Transform data with varying components.

here's a snippet of the training dataframe that was generated after normalization, pre processing and the creation of sliding windows over the dataset.

Open	High	Low	Close	Adj Close	Volume	close ms	close jpm	close ndaq	close nikkei	close bse	close nya	close vix
0.059601	0.061111	0.061662	0.049415	0.029433	0.194031	0.391336	0.387573	0.407861	0.391367	0.352739	0.016788	0.397617
0.043008	0.041265	0.051426	0.046054	0.026907	0.065146	0.390821	0.390668	0.410782	0.396127	0.356910	0.012875	0.397617
...	...	...	...	...	...	...	...	...	...	...	...	...

ma20	ma200	26ema	12ema	MACD	signal line	ema	momentum	BOLU	BOLD	long term	medium term
0	0	0.026839	0.028554	0.564450	0.563315	0.045883	0.049415	0.290221	0.294883	0.682203	0.625120
0	0	0.026571	0.028009	0.561962	0.562742	0.043333	0.046054	0.290221	0.294883	0.681346	0.622065
...	...	...	...	...	...	...	...	...	...	...	...

Table 2.1: First 2 rows of the pre processed training dataframe

2.1.4 The Models

The application's forecast tool is powered by two deep learning models. One for making daily predictions and one that does long-term predictions over two weeks at a time.

Both the models used the *Adam* optimization algorithm, however we experimented with *Adadelta* and *Adagrad* as well but concluded upon *Adam* as our choice

StonkNet-Daily

We used the very popular *TensorFlow* and *TensorfFlow-Keras* libraries for creating these deep learning models.To start off here's the initial model structure that we started off with, it was meant to make predictions one day into the future:

```
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True),
    tf.keras.layers.Dense(units=1)
])
```

This was a stateless[9] LSTM model with 32 units and a data set of batch size 32, it was trained on about 20 epochs with early stopping regularisation with a patience value of 2. It ended up over fitting by a large factor and these are its learning curves which are indicative of the same:

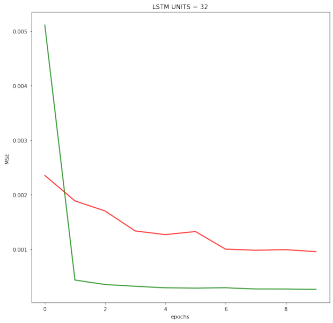


Figure 2.1: Overfit model learning curves

After some tuning, the final model structure was as follows:

```
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(16, return_sequences=True),
    tf.keras.layers.Dense(units=1)
])
```

This model has 16 units, an unchanged batch size for the data set, the patience for early stopping up scaled to 15 and the input and output time steps in a single sliding window equal 150. The model fits very well and performs relatively well against a baseline model. The baseline model is used as a measure of the minimum accuracy a good model should have, in our case, the baseline RMSE value comes out to be 0.01584 while the *LSTM*'s RMSE is a very close 0.01523.

Usually its desirable to have a larger difference between model performance and baseline but since we're only predicting one day into the future, this is acceptable.These are the learning curves for the final model:

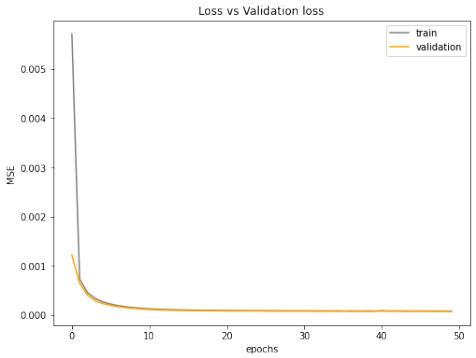
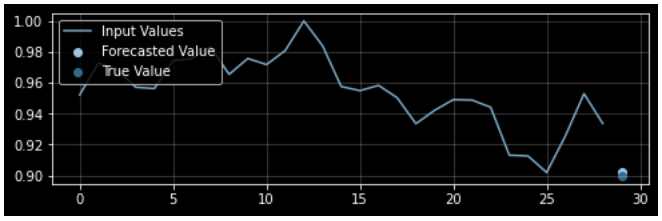
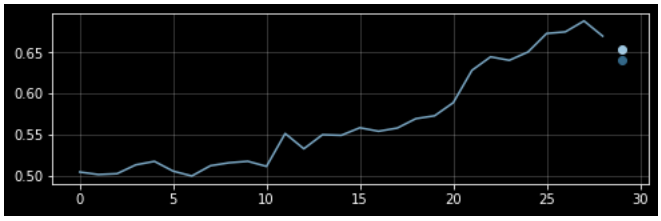


Figure 2.2: Final daily model learning curves

following are the results obtained upon testing the model on the test dataset:



(a) plot 1



(b) plot 2

Figure 2.3: Daily Model Test Results

StonkNet-Weekly

The two week model was significantly more difficult to train than the daily one. we ended up experimenting with *encoder-decoder type LSTMs*, *Stacked LSTMs* and several kinds of *regularisation* before we settled with a *vanilla* stateless[9] LSTM with substantially more neurons than the Daily model. here's the structure of the final two week model.

```
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(320, return_sequences=True),
    tf.keras.layers.Dense(units=1)
])
```

This model was trained on 100 epochs. This model also used the *MSE* score of the baseline as a direct argument for the early stopping regularisation with a patience value of 10. The feature and label sizes of the sliding window were set to 100. The learning curve of this model was turbulent but still did *not* indicate overfitting.

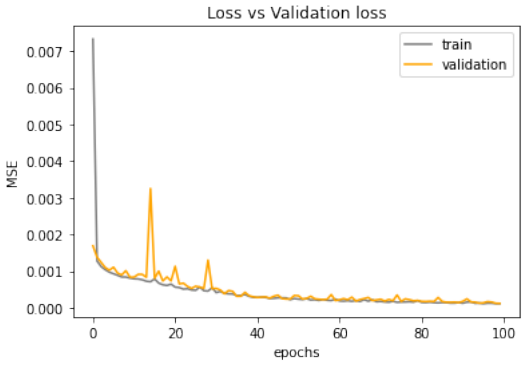
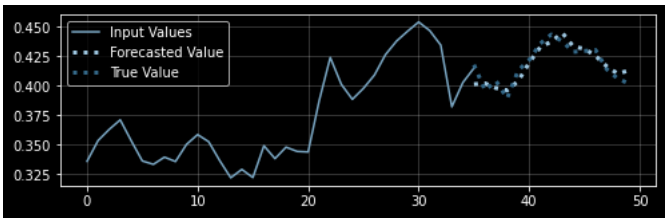
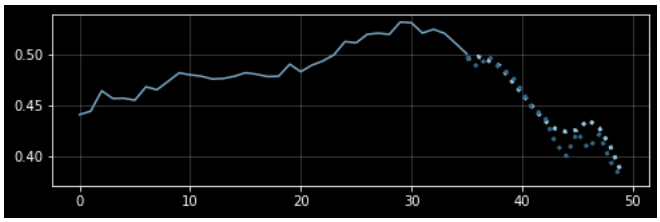


Figure 2.4: weekly model learning curves

This model performs significantly better against the baseline as compared to the daily model, probably because of the large offset. The baseline model's RMSE came to around 0.0529 while the LSTM scored an RMSE of 0.0125. This model was also tested on the test dataset like the daily model and provided satisfactory results:



(a) plot 1



(b) plot 2

Figure 2.5: Weekly Model Test Results

2.2 Prediction

Once we obtained the finish models the first task was to export them, since we weren't working on a local system, we first exported the model in TensorFlow's *SavedModel* format to google drive, and then download it locally.

```
from google.colab import drive
drive.mount('/content/gdrive')
path = f"/content/gdrive/My Drive/{model_save_name}"
tf.keras.models.save_model(lstm_model, filepath=path)
```

loading the model and making predictions is relatively simple with *TensorFlow-Keras* it was done using these functions:

```
def load_models(path):
    model = tf.keras.models.load_model(path)
    return model

def get_predictions(model, data):
    predictions = model.predict(data)
    predictions = predictions[0, :, 0]
    return predictions
```

The predictions however would be returned in a 3 dimensional array because this is a stateless model and it returns the output as 3 dimensional sequence in the same format as it was fed in, However the *get-predictions* functions takes care of this and converts the three dimensional tensor to a one dimensional tensor. The predictions will also still be scaled down between 0 and 1, however changing this turned out to be easy as the module that provides the scaler also provides an inverse transformation for it, the following function was created to apply it to any value. The function also takes in the index of the original dataframe, before it was normalized as a parameter and applies it back to the predictions and any other values with the appropriate increments.

```
def inverse_transform(values, dates, future=None, index=3, cols=24):
    df_shaper = pd.DataFrame(values)
    for i in range(cols):
        df_shaper[i + 1] = df_shaper[0]
    df_shaper_columns = df_shaper.columns
    unscaled_val_arr = scaler.inverse_transform(df_shaper)
    df_unscaled = pd.DataFrame(unscaled_val_arr, columns=df_shaper_columns)
    inverse_values_df = df_unscaled[index]
    if future:
        for date in dates:
            date += datetime.timedelta(days=future)
        dates = dates[:-1]
        inverse_values_df.index = dates
        inverse_values = inverse_values_df.to_numpy()
    else:
        dates = dates[:-1]
        inverse_values_df.index = dates
        inverse_values = inverse_values_df.to_numpy()
    return inverse_values, inverse_values_df
```

To actually obtain the daily or weekly forecast one most just splice the same number of elements as the value of the offset from the end of the array or dataframe of predictions for example, for obtaining the daily forecast one would just use $dataframe[-1:]$ furthermore we wrote some code in order to assess the model's performance with labelled data during training, the plots returned by that function are also used extensively in this report.[2]

2.3 Data Retrieval

The frontend is responsible for displaying copious amounts of data, this part of the backend is responsible for fetching all of that data.

The core principle behind all the data retrieval code lies behind web scraping, while this does slow down the program quite a bit, it does provide easy and accurate results. The yfinance module itself uses a large amount of web scraping, that coupled with the scrapers that we ourselves set up, adds quite a bit of runtime to the application.

Almost all the data displayed on the screen is either scraped manually or obtained from the yfinance module and then dumped into a *.json* file for easy implementation[3]

[2] check appendix D for greater clarity and scripts
[3] check appendix C for greater clarity and scripts

Chapter 3

Frontend

As stated in the progress report, *Stonks* was originally meant to be deployed as a web application via the React framework. However due to certain difficulties this plan was scrapped and Stonks was deployed as a desktop application.

The progress report also mentioned that *Stonks* would have a U.I. design catered specifically to the target demographic. This goal was fulfilled with relatively minor complications. The mock ups for the U.I. were designed over a relatively large time window and so you may see deprecated features still prevalent in some of them.

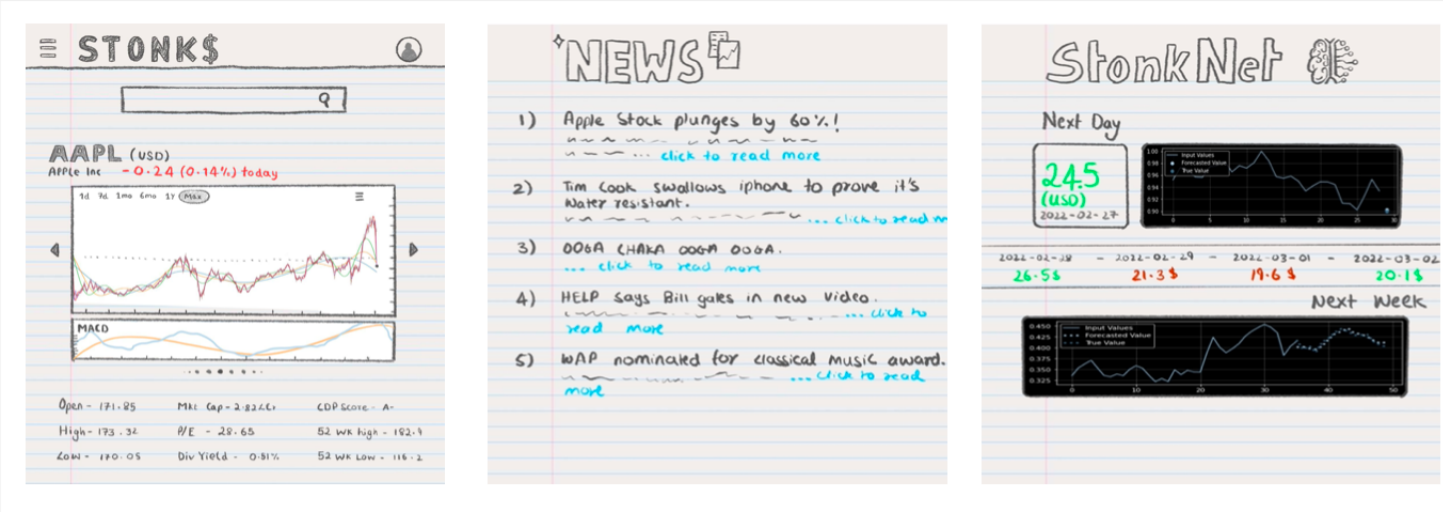


Figure 3.1: Mockups

To stay true to the "handwritten notebook" U.I. all the graphics on screen,including the font, were *hand made* using Procreate which is conventionally used for digital art and isn't the optimal choice for U.I. design.

3.1 Kivy

In order to deploy Stonks as a desktop application, the Kivy framework was used. Kivy makes U.I. development in python easy by separating the python code from the G.U.I. components, this is done via a *.kv* file that can be referenced separately in code.

The *Widget class* is the basis of all G.U.I. components in Kivy, it defines all the necessary behaviours required in order to create a fully functional user interface.

several kinds of instances of the widget class were used in this project, they're listed below:

Labels: Blocks of space that can be filled with texts, colours or images

Buttons: Perform a function on clicking

BoxLayout: Pre-defined yet highly customizable layout of arrangement of widgets, which can itself be treated as a widget and inserted in other BoxLayouts.

Text input: Takes input from the user and stores it for further use

Carousel: Creates multiple spaces that can be filled with widgets and switched by swiping from one space to another

Radio Buttons: A collection of parameters, out of which only one can be chosen

Graph: Can be created by using the *KivyCanvasFigureAgg()* of the Kivy module and can be treated as an image

Image: Images can be treated as widgets in Kivy

Screen manager: Used to handle multiple screens in the front end that can be accessed by pressing designated buttons.

Widget Properties

Every widget in Kivy can be assigned properties in order to influence their behaviour. the following properties were utilized in the creating of the Stonks GUI.

1. **size-hint:** takes size of the widget relative to the environment around it as a tuple of form (length, width). Can be assigned to all widgets.
2. **id :** Creates a variable that can be accessed from the python code. Can be assigned to all widgets.
3. **pos:** Takes position of the widget relative the the environment around it as a tuple of form (x, y). Can be assigned to all widgets.
4. **color:** Takes RGBA values as a tuple of form (R, G, B, A). Can be assigned to labels, the background, buttons, text and text input
5. **font-size:** Used to control the size of the font. Can be assigned to all widgets that accept fonts.
6. **font-name:** Can be used to customise the fonts used in the entire project or one widget at a time.
7. **padding, spacing:** Padding creates an envelope of empty space outside of a widget and spacing creates vacancy between widgets inside the same layout
8. **source:** Takes absolute/relative path of an image. Can be assigned to buttons, labels, images, text inputs etc.
9. **orientation:** Can be set to 'horizontal' or 'vertical' for BoxLayout and determines the arrangements of widgets inside it.
10. **text**: Text can be assigned to input boxes, labels, radio buttons, and buttons
11. **on-press:** Used to reference functions defined in the python script inside the .kv file to specific buttons.
12. **on-active:** Used to reference functions defined in the python script inside the .kv file to radio buttons.

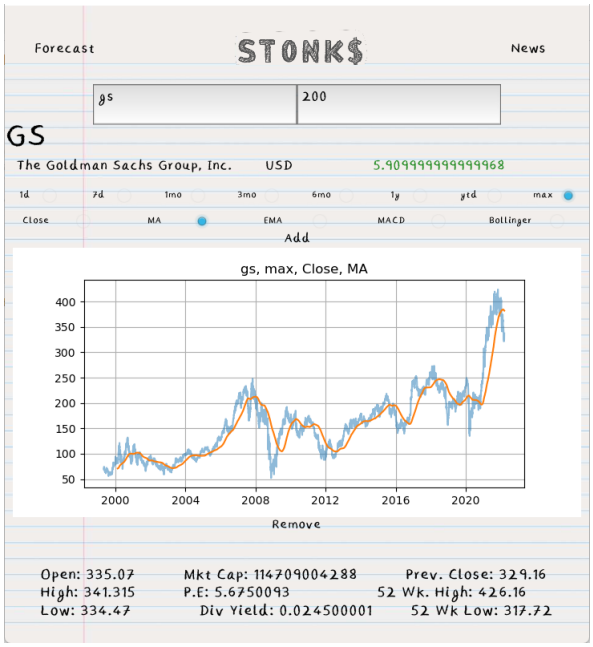
Chapter 4

Conclusion

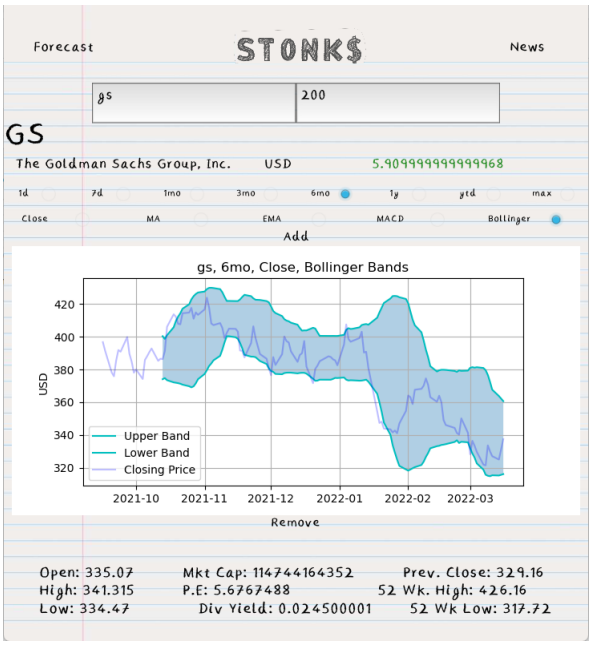
To conclude, while the application does not meet the goals, it still fulfills the job of being a good stock screening application. This segment will conclude upon the project and display the final outcome and talk about some important final steps.

4.1 Finalized Application

The following screenshots are of the final application. the application runs non erroneously and performs as expected.



(a) Home Page with MA indicator displayed



(b) Home Page with Bollinger bands displayed

Figure 4.1: Home Page

Figure 4.2: Financial News Page

4.2 Finalized Models

The models were finalised with the previously discussed architecture and trained on 100% of the dataset as opposed to the 70% (training set), this step is called finalising a model.

On 100% data the weekly model took around 1:15:45 to train while the daily model took around 00:08:45. The daily model was found to have a total and trainable parameter count of 2705 with 2688 of those being those of the LSTM layer and rest belonging to the Dense layer. The weekly model was found to have a total and trainable parameter count of $443,201$ with $442,880$ of those belonging to the LSTM layer and the rest to the Dense layer.

Both the models contain 0 non trainable parameters. Following the training, the models were exported to *TensorFlow's SavedModel* datatype and put to use on the prediction tools.

4.3 License

We're strong supporters of the Open Source Software Movement and as such have chosen to license this project accordingly. Both the neural networks used in this project are licensed under the *Mozilla Public License v2.0* and are available on this GitHub repository[1]. The *Mozilla Public License 2.0* is a weak copyleft license and under it contributors are free to do the following

- commercial use
- distribution
- modification
- patent use
- use

as long as the following conditions are agreed upon:

- Source code must be made available when licensed material is distributed
- A copy of the license and copyright notice must be made available with the licensed material
- Modifications of the existing files must be released with the same license when distributing the licensed material, in some cases a similar or related license may be used.

The full license and its descriptions may be found on the previously mentioned GitHub repository along with the *SavedModel* files and two Python notebooks which contain all the code for the neural networks along with markdown cells which go slightly deeper into the workings of the network.

[1] for anyone viewing a hard copy of this document, the link is: https://github.com/Batsy24/Stonks

References

[1] Jason Brownlee (PhD). *Time Series Forecasting as Supervised Learning.* 2016. URL: https://machinelearningmastery.com/time-series-forecasting-supervised-learning/.

[2] Alexander Amini. *MIT 6.S191: Recurrent Neural Networks.* 2021. URL: https://www.youtube.com/watch?v=qjrad0V0uJE.

[3] Boris Banushev. *Using the latest advancements in deep learning to predict stock price movements.* 2019. URL: https://towardsdatascience.com/aifortrading-2edd6fac689d#c237.

[4] codebasics. *Vanishing and exploding gradients — Deep Learning Tutorial 35 (Tensorflow, Keras Python).* 2021. URL: https://www.youtube.com/watch?v=qowp6SQ9_Oo.

[5] G Sanderson. *But what is the Fourier Transform? A visual introduction.* URL: https://www.youtube.com/watch?v=spUNpyF58BY.

[6] S Talebi. *The Fast Fourier Transform (FFT). With a teaspoon of intuition — by Shawhin Talebi.* URL: https://medium.com/swlh/the-fast-fourier-transform-fft-5e96cf637c38.

[7] S Talebi. *Time Series, Signals, the Fourier Transform — by Shawhin Talebi — Towards Data Science.* URL: https://towardsdatascience.com/time-series-signals-the-fourier-transform-f68e8a97c1c2.

[8] *Time series forecasting — TensorFlow Core. (n.d.)* 2021. URL: https://www.tensorflow.org/tutorials/structured_data/time_series.

[9] Anita Yadav, CK Jha, and Aditi Sharan. "Optimizing LSTM for time series prediction in Indian stock market". In: *Procedia Computer Science* 167 (2020), pp. 2091–2100.

[10] Zijing Zhu. *Taking Seasonality into Consideration for Time Series Analysis — by Zijing Zhu — Towards Data Science.* URL: https://towardsdatascience.com/taking-seasonality-into-consideration-for-time-series-analysis-4e1f4fbb768.

Special thanks

we would like to thank Shloka Shetty and Tanay Nalawde of XII A for their help with this project. We would also like to thank Rachna Ranade, Jeff Holmes. The following webites also played a crucial role in providing us with the information necessary for this process: Stackoverflow, Investopedia, YouTube, Machine Learning Mastery

Appendix A

Training Procedure and Hyperparameters

A.1 Backpropagation through time and vanishing gradients

[2]

In feed forward neural networks, we first feed forward the data through each layer where each neuron performs a weighted sum of all the activations and then applies a non linear activation function such as sigmoid or ReLU. Once the final prediction is made, the loss function is calculated to determine how well the model performed. Then a derivative is taken of the cost with respect to every parameter such as the weights and the bias and using that the weights are shifted in the network going backward, this is backpropogation.for example assume a neural network of just two neurons L and $(L-1)$

in such a network the cost of one training example can be given by the difference between the predicted value and the actual value, squared:$C_0 = (a^L - y)^2$ where y is just the label value and a^L is last activation or the predicted value and is given by:

$$a^L = \sigma(w^L a^{L-1} + b^L) \tag{A.1}$$

$$z^L = w^L a^{L-1} + b^L \tag{A.2}$$

where w^L is the weight, $a^{L-1}$is the activation of the last neuron and b^L is the bias value. σ is the activation function. we can compute the gradient necessary for the gradient descent algorithm by applying chain rule of derivatives

$$\frac{\partial C_0}{\partial w_L} = \frac{\partial z_L}{\partial w_L} \cdot \frac{\partial a_L}{\partial z_L} \cdot \frac{\partial C_0}{delta_L} = a_{L-1} \,.\, \sigma'(z_L) \,.\, 2(a_L - y) \tag{A.3}$$

we would have to average this for every training example.

[4]now when you backpropagate like this through the network it is possible that your gradients towards the end of the network are very small, i.e. the neurons near the end don't contribute a lot to the final prediction. In this case when you backpropagate further if your initial weights near the beginning of the network are also small and you try to adjust them using:

$$\frac{\partial(cost)}{\partial w^N} = \frac{\partial(activation)}{\partial w^N} \cdot \frac{\partial(cost)}{\partial activation} \tag{A.4}$$

you'll get a very small value since the both derivatives will end up being small. so the more the layers you add, the more of these derivatives you'll multiply and so the gradient becomes very small near the earlier layers and in turn the changes to the weights become very small and so the training process slows down. this is the problem of *vanishing gradients*.

This same backpropagation algorithm is applied across the time steps instead of across the layers of neurons in RNNs and that's called *Backpropagation Through Time* and the same problem of *vanishing gradients* arises there which is why we choose to use gated cells such as LSTM and GRU.

A.2 Hyperparameter tuning

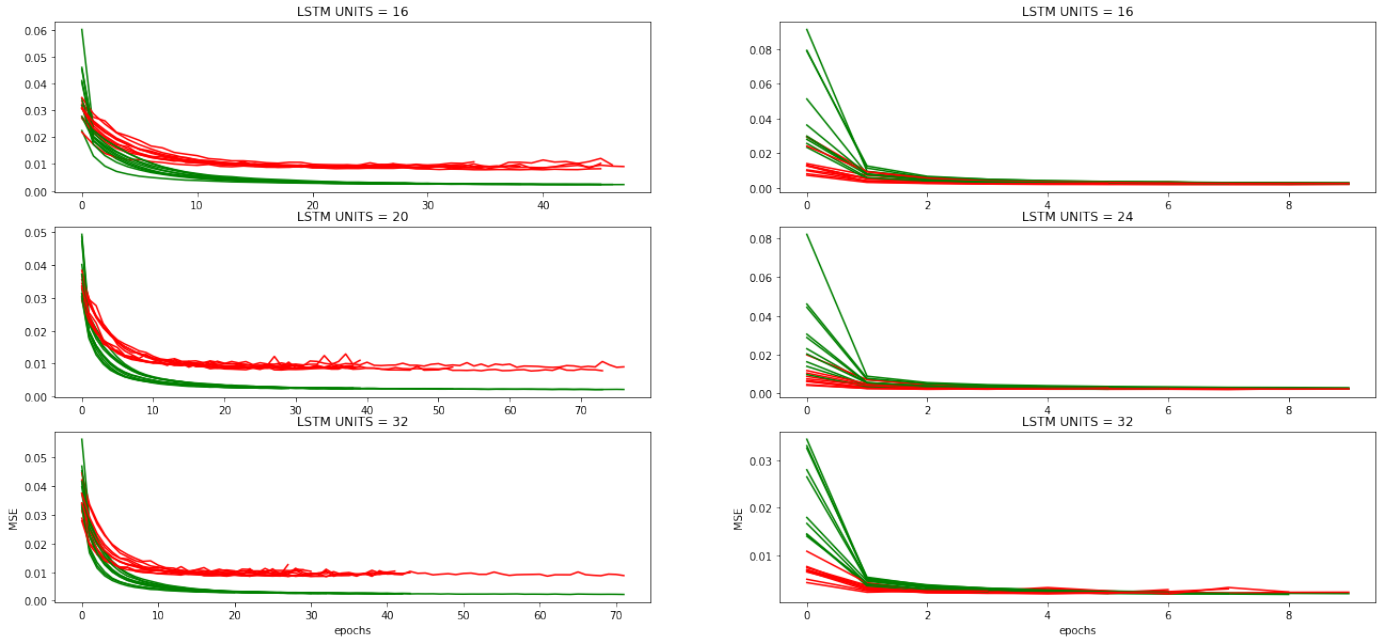
Hyperparameter tuning is tricky with deep neural networks, there aren't a lot of concrete and defined ways to do it and so often time it comes down to trial and error, however this process can still be sped up with the right kind of automation. For *StonkNet* we wrote a single function that with a little tweaking could help me tune every parameter that we were considering, i.e.

- Batch size
- Epochs
- Regularization
- number of layers
- number of *LSTM* units

The following function was written to help tune these hyperparameters by training multiple neural networks and shifting the hyperparameters by a fixed value and then averaging the performance over a fixed number to account for the stochastic behaviour of the algorithm:

```
def compare_model_performance(tests=10, units=[16, 20, 32]):
  plt.figure(figsize=(10,10))
  for z in range(len(units)):
    for i in range(tests):

      lstm_model = tf.keras.models.Sequential([
        tf.keras.layers.LSTM(units[z], return_sequences=True,
                             kernel_regularizer=tf.keras.regularizers.l1(l1=0.001)),
        tf.keras.layers.Dense(units=1)])

      history = compile_and_fit(lstm_model, wide_window)
      IPython.display.clear_output()
      loss = history.history['loss']

      plt.subplot(len(units),1,z+1)
      plt.plot(loss, label='Train', c='g')
      plt.plot(history.history['val_loss'], label='Validation', c='r')
      plt.title(f'LSTM UNITS = {units[z]}')

  plt.xlabel('epochs')
  plt.ylabel('MSE')
  plt.show()
```

Here's an example output for this function, it was taken during the training process of both the models and shows the learning curves for repeated training of 2 neural networks with different values of lstm units, the red lines are the validation loss values and the green ones are the training loss values



(a) weekly model

(b) daily model

Figure A.1: Diagnostics

A.3 Baseline

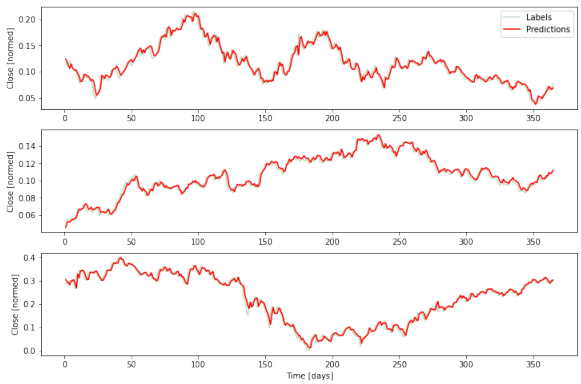
Baseline is a naive model, meaning there's no real logic or learning to how it makes predictions, its usually easy to create, simple to replicate and used as a minimum measure of accuracy for the final model, meaning the final model can only be regarded acceptable if it performs better than the baseline. In my case the choice of accuracy metric was RMSE as it goes well with time series forecasting problems. The following code was written to obtain the predictions from whats called a *persistence model*, its a popular choice of algorithm for a baseline model and it basically returns the input back as a prediction for the next time step.

```
def persistence(f):
  return f

test_data = next(iter(persistence_window.test))

inputs = test_data[0]
input_datapoints = inputs[0, :, indices]
input_datapoints = input_datapoints.numpy()

labels = test_data[1]
label_datapoints = labels[0, :, 0]
label_datapoints = label_datapoints.numpy()

preds = list()
for datapoint in input_datapoints:
  y_hat = persistence(datapoint)
  preds.append(y_hat)

preds = np.array(preds)
persistence_score = np.sqrt(mean_squared_error(label_datapoints, preds))
print(f'Persistence Model Score: {persistence_score}')

model_preds = lstm_model.predict(inputs)
model_preds = model_preds[0, :, 0]
model_score = np.sqrt(mean_squared_error(label_datapoints, model_preds))
print(f'LSTM Model Score: {model_score}')
```

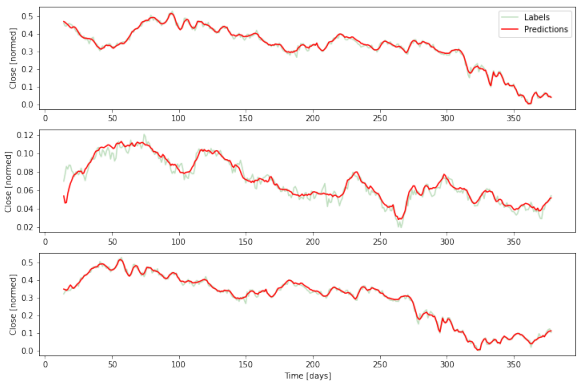
The final Persistence vs Model results for the daily and weekly models are as follows:

```
# Daily Model
Persistence Model score: 0.01584206521511078
LSTM Model Score: 0.015296120196580887

# Weekly Model
Persistence Model Score: 0.05295164883136749
LSTM Model Score: 0.012525191530585289
```

Finally here's the outputs for both the models' performance with predictions on a 365 day time frame, in practice such a time frame is *highly* unrealistic, however here it was just a matter of evaluating performance with context to labels.



(a) Daily Model



(b) two week model

Figure A.2: Performance over 365 days

A.4 Plotting Learning Curves

The following code was written for plotting the learning curves that were used extensively while tuning and training the model.

```
loss = history.history['loss']
val_loss = history.history['val_loss']
mae = history.history['mean_absolute_error']
val_mae = history.history['val_mean_absolute_error']
plt.plot(loss, label='train', c='gray')
plt.plot(val_loss, label='validation', c='orange')
plt.title('Loss vs Validation loss')
plt.xlabel('epochs'); plt.ylabel('MSE')
plt.legend(loc='upper right')
plt.show()
```

A.5 Compiling the model

We used four main kinds of regularisation when tuning and testing the model, Dropout; which drops neurons based on a predefined probability, Early stopping, and lasso and ridge regularizers, both of which penalise overfitting in their own way. lasso regularization was also shortly used as a substitute to feature selection by us.

$$ridge\ regularizer(l2) = \frac{1}{n}\sum_{i=1}^{n}(y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2 \tag{A.5}$$

$$lasso\ regularizer(l1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} abs(\theta_i) \tag{A.6}$$

Compiling and fitting the model as a whole is incredibly easy with the *TensorFlow-Keras* library, the following function was used to do so:

```
epochs = 100

def compile_and_fit(model, window, patience=10):
  early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                                    patience=patience,
                                                    mode='min',
                                                    baseline=0.0035914969630539417,
                                                    restore_best_weights=True)

  model.compile(loss=tf.losses.MeanSquaredError(),
                optimizer=tf.optimizers.Adam(),
                metrics=[tf.metrics.MeanAbsoluteError()])

  history = model.fit(window.final, epochs=epochs,
                      validation_data=window.val,
                      callbacks=[early_stopping])

  return history
```

This function was changed slightly over and over during tuning and it also differs slightly for the training of the daily model.

A.6 Accuracy and Loss metrics

For our project both the models used the *Mean Squared Error (MSE)* loss function and the *Mean Absolute Error(MAE)* as the accuracy metric, and the *Root Mean Squared Error(RMSE)* metric for comparing baseline performance with model performance. *MSE* is often used for regression problems such as ours and both *MAE* and *RMSE* punish false forecasts in Time series problems.

$$Mean\ squared\ error:\ J(W) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{A.7}$$

$$Mean\ Absolute\ Error := \frac{\sum_{i=1}^{n} abs(\hat{y}_i - y_i)}{n} \tag{A.8}$$

Appendix B

Preprocessing

Preprocessing refers to the processing of data before passing it to a machine learning model. This is one of the most important steps in the machine learning development pipeline.

For us the process of pre processing started with *Normalization*. *Normalization* refers to the process of scaling a set of values to a uniform scale. This is a very important step as deep neural networks are very sensitive to scale. There are several ways of Normalizing data, however we decided to go with the MinMaxScaler since we had experience with it in the past. We also tried experimenting with the StandardScaler but gave up quickly due to time constraints. MinMaxScaler basically scales down the data to a predefined range, in the default case that is 0 and 1, we chose to leave it like that.

The equation that governs this transformation is given by:

$$y = std\ .(max - min) + min \tag{B.1}$$

following the min max transformation we split the total data into 3 sets of fractions $0.7, 0.2$ and 0.1 i.e. 70%, 20% and 10% of the total data, this is called a train-val-test split. The 70% of the data is called *training data* and is used to *train* the model, at each epoch the model is then shown the 20% of data which is called *validation data*, its data that the model hasn't seen during training and is used to check if the model is overfitting or not and finally the *test data* or the last 10% of the data is used to evaluate the model once its done training entirely.

The following code was written in order to carry out normalization and the data split

```
# df is the dataframe with all the data
dates = df.index.to_list()
df_index = df.index
columns = df.columns.tolist()

scaler = MinMaxScaler()

df_scaled = scaler.fit_transform(df)
df_scaled_pd = pd.DataFrame(df_scaled, index=df_index, columns=columns)


n = len(df_scaled_pd)
train_df = df_scaled_pd[0:int(n*0.7)]
val_df = df_scaled_pd[int(n*0.7):int(n*0.9)]
test_df = df_scaled_pd[int(n*0.9):]
```

B.1 Sliding Windows

[1][8] This is a step unique to time series forecasting problems and also a very crucial one. Time series data by itself is just a series of daily data with no actual defined "relation" for the model to process. A supervised machine learning problem, by definition has a dataset which has feature-label pairs, creating *Sliding Windows* over our entire dataset can help us transform it into a supervised learning problem. A simple *sliding window* may be defined as such: for time step t let the label be $t + 1$. Rearranging the data like this tells the model what it should be using to predict what and how many days into the future because as in the case of our two week model one may also define a sliding window where for a feature of time step t the label is $t + 14$.

In both the models we consider more than just one time step as the input and we consider the number of features to be equal to the number of labels. the amount of days its predicting in the future is given by a variable value called the *offset*.

The following code defines a class called WindowGenerator which we utilised to create such *Sliding Windows* over our entire dataset, the class also defines some useful methods and attributes that were used during evaluation of the models. It also reshapes the data into a 3 dimensional tensor of format (samples, time steps, features) as is required by LSTM models

```
class WindowGenerator():
  def __init__(self, input_width: int, label_width: int, offset: int,
               train_df=train_df, val_df=val_df, test_df=test_df,
               final_df=df_scaled_pd,
               label_columns=None):

    # assertions
    assert input_width > 0, "Input width must be a positive number!"
    assert label_width > 0, "Label width must be a positive number!"
    assert offset > 0, "Offset value must be a positive number!"


    # DATAFRAMES
    self.train_df = train_df
    self.val_df = val_df
    self.test_df = test_df
    self.final_df = final_df

    # giving indices to every column in label_columns (in this model there'll only be one label_column)
    # and to every column in the dataset
    self.label_columns = label_columns
    if label_columns is not None:
       self.label_column_indices = {name: i for i, name in enumerate(label_columns)}

    self.column_indices = {name: i for i, name in enumerate(train_df.columns)}

    # WINDOW PARAMETERS.

      # input parameters
    self.input_width = input_width
    self.label_width = label_width
    self.offset = offset
    self.total_window_width = input_width + offset

    self.input_slice = slice(0, input_width)
    self.input_indices = np.arange(self.total_window_width)[self.input_slice]

       # label parameters
    self.label_start = self.total_window_width - self.label_width
    self.label_slice = slice(self.label_start, None)
    self.label_indices = np.arange(self.total_window_width)[self.label_slice]


  def split_window(self, features):
    #(3, 7, 23)
    inputs = features[:, self.input_slice, :]
    labels = features[:, self.label_slice, :]
    if self.label_columns is not None:
      labels = tf.stack([labels[:,:, self.column_indices[name]]for name in
                         self.label_columns], axis=-1)

    inputs.set_shape([None, self.input_width, None])
    labels.set_shape([None, self.label_width, None])

    return inputs, labels

  def __repr__(self):
    return '\n'.join([
        f'Total window size: {self.total_window_width}',
        f'Input indices: {self.input_indices}',
        f'Label indices: {self.label_indices}',
        f'Label column name(s): {self.label_columns}',
        f'{self.label_column_indices}'])
def plot(self, model=None, plot_col='Close', max_subplots=3, scatter=False):

  inputs, labels = self.example
  plt.figure(figsize=(12, 8))
  plot_col_index = self.column_indices[plot_col] # pulls index for col from dict
  num_plots = min(max_subplots, len(inputs))
  for n in range(num_plots): # assume n=3
    plt.subplot(num_plots, 1, n+1)
    plt.ylabel(f'{plot_col} [normed]') # (batch, time,  features)
```

```
    plt.plot(self.input_indices, inputs[n, :, plot_col_index],
             label='Inputs', zorder=-10)

    if self.label_columns:
      label_col_index = self.label_column_indices.get(plot_col, None)
    else:
      label_col_index = plot_col_index # never runs for me lmao

    if label_col_index is None:
      continue

    if scatter:
      plt.scatter(self.label_indices, labels[n, :, label_col_index],
                  edgecolors='k', label='Labels', c='#2ca02c', s=64)
    else:
      plt.plot(self.label_indices, labels[n, :, label_col_index],
                  label='Labels', c='g', alpha=0.25)

    if model:
      predictions = model(inputs)
      if scatter:
        plt.scatter(self.label_indices, predictions[n, :, label_col_index],
                    marker='X', edgecolors='k', label='Predictions',
                    c='#ff7f0e', s=64)
      else:
        plt.plot(self.label_indices, predictions[n, :, label_col_index],
                     label='Predictions',
                     c='r')

    if n == 0:
      plt.legend()
  plt.xlabel('Time [days]')
WindowGenerator.plot = plot

def make_dataset(self, data):
  data = np.array(data, dtype=np.float32)
  ds = tf.keras.utils.timeseries_dataset_from_array(
      data=data,
      targets=None,
      sequence_length=self.total_window_width,
      sequence_stride=1,
      shuffle=True,
      batch_size=32,) # this value changes depending on the model
  ds = ds.map(self.split_window)
  return ds

WindowGenerator.make_dataset = make_dataset
@property
def train(self):
  return self.make_dataset(self.train_df)
@property
def val(self):
  return self.make_dataset(self.val_df)
@property
def test(self):
  return self.make_dataset(self.test_df)
@property
def final(self):
  return self.make_dataset(self.final_df)
@property
def example(self):
  #Get and cache an example batch of `inputs, labels` for plotting.
  result = getattr(self, '_example', None)
  if result is None:
    result = next(iter(self.train))
    self._example = result
  return result
WindowGenerator.train = train
WindowGenerator.val = val
WindowGenerator.test = test
WindowGenerator.final = final
WindowGenerator.example = example
```

Appendix C

Data Acquisition

Data acquisition is, as the name suggests, the process of acquiring the data necessary for training a supervised machine learning model. As stated earlier our dataset spanned across 25 features across 5700+ time stamps. The yfinance module was used to acquire all the daily prices and the technical indicators and Fourier transform data was calculated.

I'm using the Goldman Sachs stock data as my primary stock data. The reason for picking GS is that it shows no obvious trends over large time periods and also reflects large changes in the economy, such as the 2008 crisis very well. However using only that much is not enough. I'm adding some more features that I think will help the network identify patterns more easily and make more accurate predictions.

To start off, I'm using the closing prices of companies similar to Goldman Sachs (GS), i.e. other investment banks, I've left out some companies such as *Deutsche* intentionally in order to reduce dimensionality and also for convenience.

- JP Morgan Chase
- Morgan Stanley

Furthermore, I'm using composite indices since they reflect large scale fluctuations in the market very well, also I'm using the VIX index which is maintained by the *Chicago board options exchange (CBOE)* in order to monitor the volatility of the market.

- NASDAQ
- NYSE (New York Stock Exchange)
- BSE (Bombay Stock Average)
- Nikkei225(Nikkei Stock Average, Japan)
- VIX (Volatility Index)

Furthermore, stock analysts use two main types of analyses to study and predict stock patterns, Fundamental Analysis and Technical Analysis.

Fundamental analysis is a method of assessing the intrinsic value of a stock, this can be accomplished by looking at external influence, events, industry trends etc. Unfortunately due to time constraints I was unable to learn the logic and code behind what I wanted to implement for fundamental analysis and so it was omitted entirely from the dataset.[1]

Technical analysis is a method of identifying patterns and price trends in order to find trading opportunities using statistical means. There's many kinds of Technical Indicators used to accomplish this and I've included some of the common indicators:

- Momentum
- Moving averages over specified time periods
- Moving averages convergence/divergence (MACD)
- Exponential moving averages
- bollinger bands

The following code describes the usage of the yfinance module in acquiring data.

[1] To elaborate a little, I originally intended to perform sentiment analysis on financial news using Google's pre trained NLP models like *BERT*

```
gs = yf.download('GS', period='max', progress=False)
```

Similar code was written 6 more times in order to acquire the data for the other listed indices.

the following code returns a Pandas dataframe for each of the mentioned stocks, of which each dataframe contains 5 features

- close
- open
- adj close
- high
- low

Furthermore, to obtain volatility index prices, we use the data provided directly by CBOE who maintain the volatility index.

```
url = 'https://cdn.cboe.com/api/global/us_indices/daily_prices/VIX_History.csv'
vix = pd.read_csv(url, names=['Date', 'Open', 'High', 'Low', 'Close'])
vix = vix.iloc[1: , :]
```

However we wish to only use closing prices from every index and as such we slice every dataframe and create one large dataframe, the following was written to do exactly that

```
df=gs.copy()
def get_assets(dataframe=df):
  global vix
  dataframe['close ms'] = ms['Close']
  dataframe['close jpm'] = jpm['Close']
  dataframe['close ndaq'] = ndaq['Close']
  dataframe['close nikkei'] = nkk['Close']
  dataframe['close bse'] = bse['Close']
  dataframe['close nya'] = nya['Close']

  vix = vix.set_index(vix['Date'])
  vix = vix.drop('Date', axis=1)
  vix.index = pd.to_datetime(vix.index)
  dataframe.index = pd.to_datetime(dataframe.index)

  dataframe['close vix'] = vix['Close']
  dataframe[['close vix']] = dataframe[['close vix']].apply(pd.to_numeric)

  return dataframe
```

C.1 Technical Indicators

[3] As discussed before, the technical Indicators are statistical tools used to assess and predict stock movements, the following function was written to calculate the necessary indicators for the dataset.

```
def get_technical_indicators(dataset=df):

  dataset['ma20'] = dataset['Close'].rolling(window=20).mean()
  dataset['ma200'] = dataset['Close'].rolling(window=200).mean()

  dataset['26ema'] = dataset['Close'].ewm(span=26, adjust=False).mean()
  dataset['12ema'] = dataset['Close'].ewm(span=12, adjust=False).mean()
  dataset['MACD'] = (dataset['12ema']-dataset['26ema']) # MACD
  dataset['signal_line'] = dataset['MACD'].ewm(span=9, adjust=False).mean()

  dataset['ema'] = dataset['Close'].ewm(com=0.5).mean() # EMA

  dataset['momentum'] = dataset['Close']-1 # Momentum

  dataset['TP'] = (dataset['Close'] + dataset['Low'] + dataset['High'])/3
  dataset['std'] = dataset['TP'].rolling(20).std(ddof=0)
  dataset['MA-TP'] = dataset['TP'].rolling(20).mean()
  dataset['BOLU'] = dataset['MA-TP'] + 2*dataset['std']
  dataset['BOLD'] = dataset['MA-TP'] - 2*dataset['std']

  dataset = dataset.drop('TP', axis=1)
  dataset = dataset.drop('MA-TP', axis=1)
  dataset = dataset.drop('std', axis=1)

```

```
  return dataset
```

C.2 Cleaning up the data

Before I calculated the Fourier transform, I had to clean up the data in order to avoid errors and discrepancies, this involved doing things such as eliminating or replacing NaN values wherever necessary. The following functions were written in order to do so.

```
mean_nkk = df['close nikkei'].mean()
mean_bse = df['close bse'].mean()
mean_vix = df['close vix'].mean()
mean_bolu = df['BOLU'].mean()
mean_bold = df['BOLD'].mean()

mean_list = [mean_nkk, mean_bse, mean_vix, mean_bolu, mean_bold]
incomplete_cols = ['close nikkei', 'close bse', 'close vix', 'BOLU', 'BOLD']
n = 0

for mean in mean_list:
  df[incomplete_cols[n]].fillna(value=mean, inplace=True)
  n+=1

df['ma200'] = df['ma200'].fillna(0)
df['ma20'] = df['ma20'].fillna(0)
```

C.3 Fourier Transform

[5][7][6][10]

Any wave varying with time or any wave in the *time domain* is called a time signal. any such signal can be represented as a linear combination of sines and cosines of varying frequencies f_n and amplitudes A_n and B_n:

$$y(t) = \sum_n (A_n \sin(2\pi f_n t) + B_n \cos(2\pi f_n t)) \tag{C.1}$$

The Fourier transform, given by:

$$\hat{g}(f) = \int_{-\infty}^{\infty} \mathrm{e}^{-2\pi i f t}\, \mathrm{dt} \tag{C.2}$$

lets you see the signal in the *frequency domain* where each frequency that the original signal was comprised of is represented by a spike in the plot. The FFT or The Fast Fourier Transform does the same thing but it is more computing efficient. in this case, since there's no seasonality or periodicity in stocks, I'll be using the FFT and inverse FFT to perform noise reduction on the data and get a combination of sines and cosines as the resulting graph. we can use this graph to analyse long term trends in the data over specific windows.

The code below allows us to plot and obtain the values of the necessary Fourier Transform data

```
# Fourier Transfrom LETSGOOOOO whooooo
df['Date'] = df.index
data = {'Date': df['Date'].tolist(),
        'Prices': df['Close'].tolist()}
df_ = pd.DataFrame(data)
df_.Date = pd.to_datetime(df_.Date)
fft = tf.signal.fft(np.asarray(df_['Prices'].tolist()))
fft_df = pd.DataFrame({'fft':fft})
fft_df['abs'] = np.abs(fft_df['fft'])
df = df.drop('Date', axis=1)
fft_df.head()


def plot_trends_fft(dataframe=fft_df):
  global df_
  plt.figure(figsize=(20, 9))
```

```
fft_list = np.asarray(dataframe['fft'].tolist())
for num_ in [3, 6, 9, 50, 100]:
    fft_list_= np.copy(fft_list)
    fft_list_[num_:-num_] = 0
    inverse_fft = np.abs(tf.signal.ifft(fft_list_))
    plt.plot(inverse_fft, label=f'Fourier transform with {num_} components')

    # inverse_fft_arr = inverse_fft.numpy()
    if num_ == 3:
        df['long term'] = inverse_fft.tolist()
    elif num_ == 9:
        df['medium term'] = inverse_fft.tolist()

plt.plot(df_.Prices, label='Real', alpha=0.5)
plt.xlabel('Days')
plt.ylabel('Price')
plt.title('Fourier transforms and GS (Closing) prices')
plt.legend()
plt.show()
```

The following plot is generated, you can observe how the plot gets closer to the ground truth as the number of components are increased

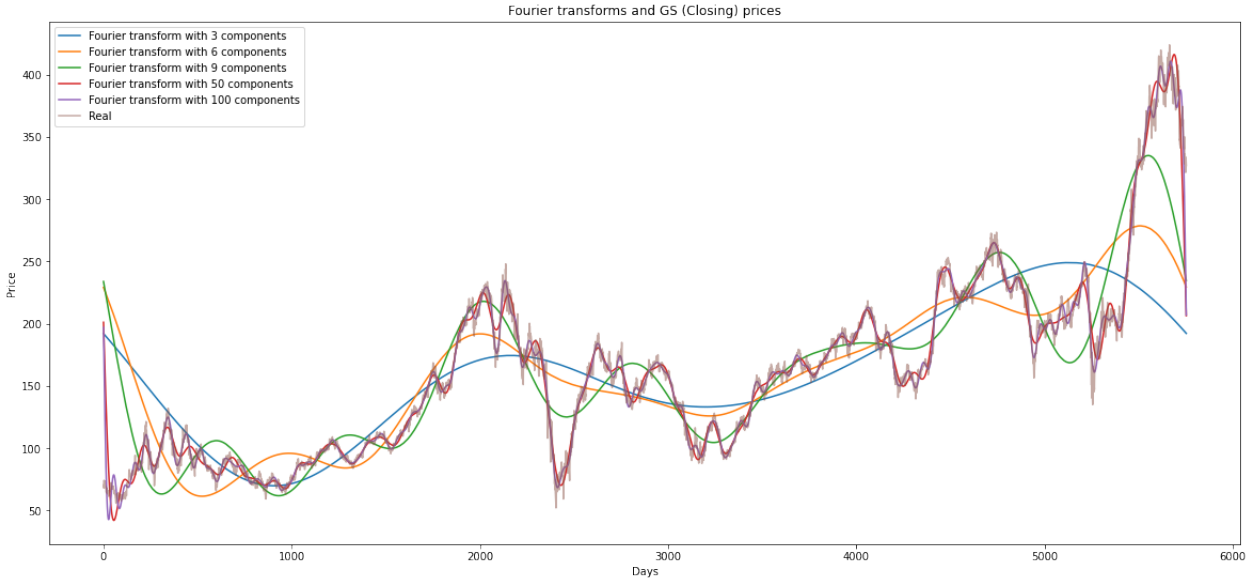


Figure C.1: Inverse Fourier Transform with varying components

Appendix D

Code Snippets

This section of the report contains all the miscellaneous code and its respective output that couldn't be placed anywhere due to irrelevancy.

This section primarily exists to fulfill the requirement of having the entire code in the report.

D.1 Data Exploration

Data Exploration is a crucial step in the ML development pipeline, it involves using visualisation methods in order to familiarise oneself with the dataset. In our case I plotted the technical indicators and conducted a few other tests in order to familiarise myself. This proved to immensely useful later when debugging. The following code snippet depicts the functions I created in order to visualise some parts of my data.

```
def plot_technical_indicators(dataset=df, last_days=5700):
    plt.figure(figsize=(16, 10), dpi=100)
    shape_0 = dataset.shape[0]
    xmacd_ = shape_0-last_days

    dataset = dataset.iloc[-last_days:, :]
    x_ = range(3, dataset.shape[0])
    x_ =list(dataset.index)

    plt.subplot(2, 1, 1)
    plt.plot(dataset['ma200'],label='MA 200', color='r',linestyle='--')
    plt.plot(dataset['ma20'],label='MA 21', color='b',linestyle='--')

    plt.plot(dataset['BOLU'],label='Upper Band', color='c')
    plt.plot(dataset['BOLD'],label='Lower Band', color='c')
    plt.fill_between(x_, dataset['BOLD'], dataset['BOLU'], alpha=0.35)

    plt.plot(dataset['Close'],label='Closing Price', color='b', alpha=0.25)
    plt.title(f'Technical indicators for Goldman Sachs - last {last_days} days.')
    plt.ylabel('USD')
    plt.legend()

    plt.subplot(2, 1, 2)
    plt.plot(dataset['close vix'], label='VIX', color='b', alpha=0.5)
    plt.plot(dataset['Close'], label='Closing price', color='r')
    plt.title('Volatility index against prices')
    plt.legend()

    plt.show()


def plot_MACD_signal_line(dataset, last_days):
    shape_0 = dataset.shape[0]
    xmacd_ = shape_0-last_days

    dataset = dataset.iloc[-last_days:, :]
    x_ = range(3, dataset.shape[0])
    x_ =list(dataset.index)
    dataset[['MACD', 'signal_line']].plot(figsize=(16, 8))
    dataset['Close'].plot(label='Closing Price', alpha=0.25, secondary_y=True)
    plt.title('MACD and Signal')
```

The following plots are generated upon calling these functions with different parameters:

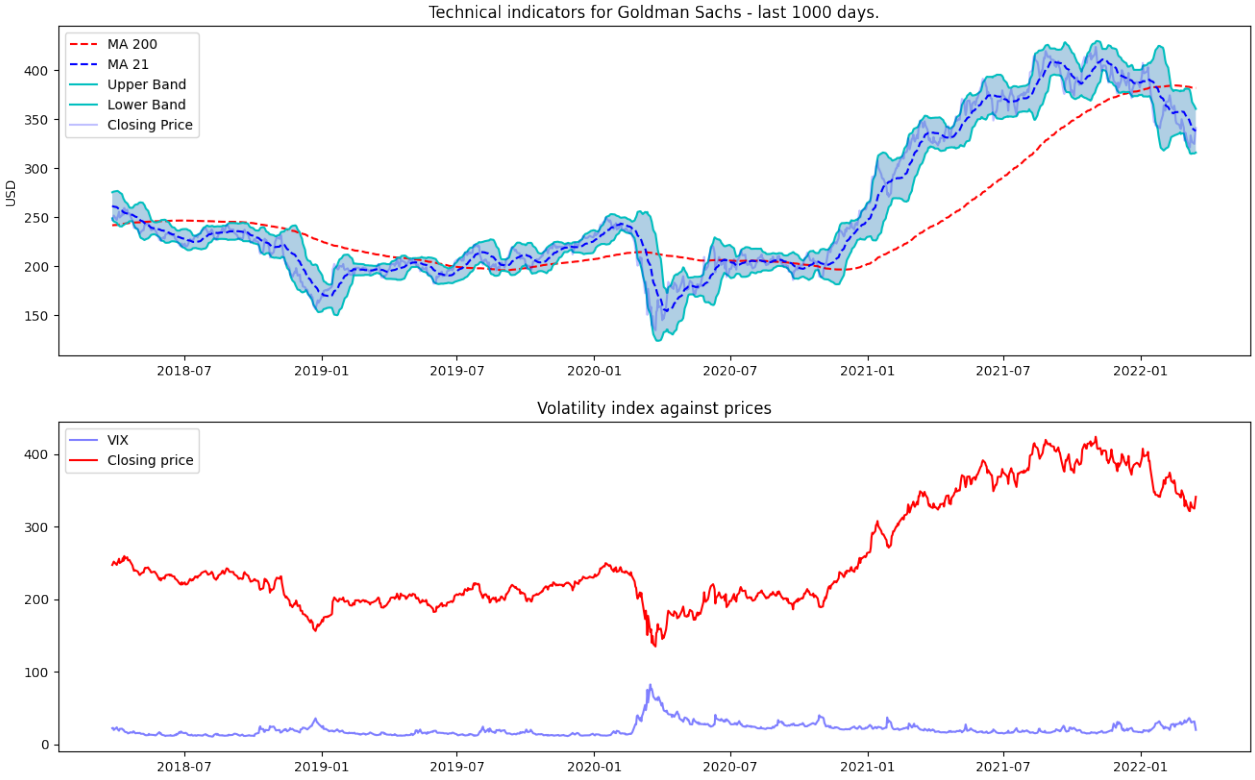


Figure D.1: Technical Indicators plotted I

The plots are as expected, There's a downward spike in the stocks at the beginning of the COVID pandemic and further research showed similar characteristics denoting the 2008 financial crisis, it is interesting to note the working of the Volatility index and the technical indicators.

Accompanying these there's also a 30 day plot of the MACD indicator

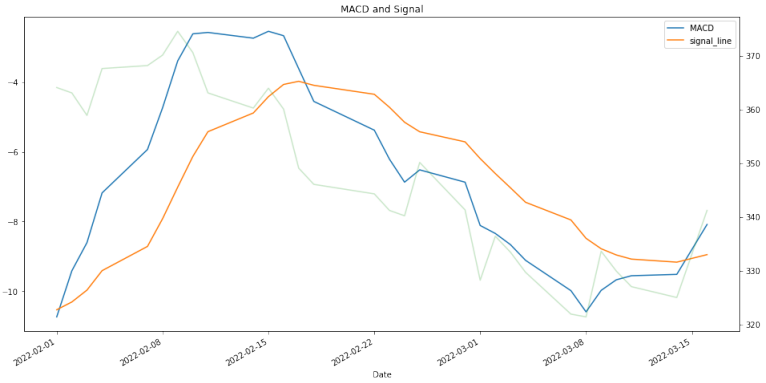


Figure D.2: MACD vs Signal Line

D.2 Data Retrieval

As mentioned in the footnotes, this section contains the scripts regarding the retrieval of data.

D.2.1 Homepage data scraper

This script is responsible for scraping, cleaning and dumping statistical data into a *json* file in order to implement it with the UI.

```
def get_news(news_list=news_):
    link_list = list()
    news_keys = list()
    for i in news_list:
        link = i['link']
        link_list.append(link)
        title = i['title']
        news_keys.append(title)

    return link_list, news_keys

def get_news_sample(link_list):
    samples_list = list()
    for link in link_list:
        data = requests.get(link).text
        div_caas_body = SoupStrainer('div', {"class": "caas-body"})
        scraper = BeautifulSoup(data, 'lxml', parse_only=div_caas_body)
        try:
            for i in scraper:
                txt = scraper.find('p').text
            samples_list.append(txt)
        except:
            placeholder = ''
            samples_list.append(placeholder)

    return samples_list
def get_trimmed_news(sample_list):
    new_samples = list()
    str_slice = slice(0, 120)
    for sample in sample_list:
        trimmed_sample = sample[str_slice]
        new_sample = trimmed_sample + '... click to read more'
        new_samples.append(new_sample)
    return new_samples
# noinspection PyBroadException
def get_data(news_titles, news_list):
    stats = dict()
    news = list()
    stat_keys = ['sector', 'longBusinessSummary', 'website', 'industry', 'currentPrice', '
    financialCurrency',
                 'longName', 'symbol', 'previousClose', 'open', 'trailingPE', 'marketCap', 'dayLow', '
    fiftyTwoWeekHigh',
                 'fiftyTwoWeekLow', 'dividendYield', 'dayHigh', 'logo_url']
    info = stock.info
    stats.fromkeys(stat_keys, None)
    for key in stat_keys:
        stats[key] = info[key]
    for i in range(len(news_titles)):
        news_pair = (news_titles[i], news_list[i])
        news.append(news_pair)

    return stats, news
links, news_key = tuple(get_news())
samples = tuple(get_news_sample(links))
news_samples = get_trimmed_news(samples)
try:
    stock_stats, stock_news = get_data(news_key, news_samples)
    json_dat = [stock_stats, stock_news]
    with open('data.json', 'w', encoding='utf-8') as file:
        json.dump(json_dat, file, ensure_ascii=False, indent=4)
except:
    print('invalid ticker, please try again')

```

D.2.2 Chart Data Retriever

similar to the last script, this script deals with retrieving data for the graphs that are displayed on the home page. The full script also contains extra code that's basically reusing the code from the neural network's *data acquisition* segement and as such I've chosen to not include it here again.

```
if time_period_input is not '1d':
    chart_data = yf.download(ticker_input, period=time_period_input, progress=False)
else:
    chart_data = yf.download(ticker_input, period=time_period_input, interval='1h', progress=False)
```

D.3 Plotting labelled data

This the is code used to plot labelled data during training and tuning in order to check model predictions on unseen data

```
legend = True
remove = -1*offset
with plt.style.context(('dark_background')):
  num = 3

  for i in range(num):
    plt.figure(figsize=(8,8))
    inputs, labels, predictions = gen_data()
    forecast = predictions[remove-1:]
    past = inputs[:remove]
    true_value = labels[remove-1:]

    plt.subplot(num, 1, i+1)
    plt.plot(x[:remove], past, label = 'Input Values', color= '#7aa6c2')

    plt.plot(x[remove-1:],forecast, label = 'Forecasted Value', color='#9dc6e0',
             linestyle='dotted', linewidth=3)
    plt.plot(x[remove-1:],true_value, label ='True Value', color='#346888',
                linestyle='dotted', linewidth=3)

    if legend:
      plt.legend(loc=2)
      legend = False

    plt.grid(alpha=0.25)
    plt.show()
```

Outputs like this are obtained upon running this code:

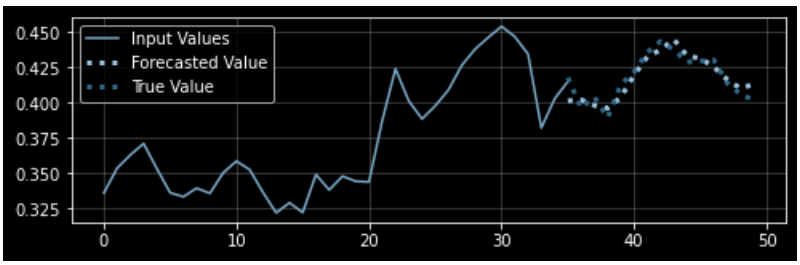


Figure D.3: prediction plot output

this code differs slightly for the daily prediction model, the only difference is that a scatter plot of the predictions and labels is displayed instead of a dotted line plot for better readability.

the gen-data function mentioned in the above code is as follows:

```
def gen_data(window=wide_window, model=lstm_model):
  dat = next(iter(window.test))
  input = dat[0]
  label = dat[1]
  predictions = model.predict(input)

  print(f'input shape = {input.shape}, label shape = {label.shape}, prediction shape = {predictions.shape}
    ')

  index = window.column_indices['Close']

  input_col = input[n, :, index]
  label = label[n, :, 0]
  predicted = predictions[n, :, 0]

  return input_col, label, predicted
```

D.4 Kivy UI

this section contains the script used to create the user interface in python using Kivy and the .kv file.

The following class was defined in order to handle the behaviours of the Home page.

```
class HomePage(Screen):
    def __init__(self, **kwargs):
        super(HomePage, self).__init__(**kwargs)
    def addInCarousel1(self):
        global ticker
        global car1num
        global time
        global column
        global infovals1
        global window
        global stock_news
        window_period = self.ids.input_window.text
        if window_period == '' or window_period == 'Enter Period':
            self.ids.input_window.text = '200'
            window = 200
        else:
            window = int(window_period)
        ticker = self.ids.input_text.text
        if ticker == '' or ticker == 'Enter Input Ticker':
            self.ids.input_text.text = 'Error'
        elif car1num < 1 and (ticker != '' and ticker != 'Error'):
            fig, ax = plt.subplots()
            dataset_main = returndata.give_chart_data(time, ticker)
            if column == 'Close':
                dataset = dataset_main[column]
                ax.plot(dataset)
                ax.set_title(f'{ticker}, {time}, Close')
            elif column == 'MA':
                dataset1 = dataset_main['Close']
                dataset2 = returndata.MA(window, dataset_main)
                ax.plot(dataset1, alpha=0.5)
                ax.plot(dataset2)
                ax.set_title(f'{ticker}, {time}, Close, MA')
            elif column == 'EMA':
                dataset1 = dataset_main['Close']
                dataset2 = returndata.EMA(window, dataset_main)
                ax.plot(dataset1, alpha=0.5)
                ax.plot(dataset2)
                ax.set_title(f'{ticker}, {time}, Close, EMA')
            elif column == 'MACD':
                macd_data = returndata.MACD(dataset_main)
                shape_0 = macd_data.shape[0]
                xmacd_ = shape_0 - len(macd_data)
                macd_data = macd_data.iloc[-len(macd_data):, :]
                ax.plot(macd_data['Close'], label='Closing Price')
                ax.set_title(f'{ticker}, {time}, Close, MACD')
                ax.grid()
                self.graph1 = fcka(plt.gcf())    self.ids.graph_box1.add_widget(self.graph1)
                self.graph1.size_hint = (1, 0.8)
                fig1, ax1 = plt.subplots()
                ax1.plot(macd_data['MACD'])
                ax1.plot(macd_data['signal_line'])
                self.macd_graph = fcka(plt.gcf())                self.ids.graph_box1.add_widget(self.
    macd_graph)
                self.macd_graph.size_hint = (1, 0.2)
                car1num += 2
            elif column == 'Bollinger':
                dataset = returndata.Bollinger(dataset_main)
                last_days = len(dataset)
                fig.dpi = 100
                shape_0 = dataset.shape[0]
                xmacd_ = shape_0 - last_days
                dataset = dataset.iloc[-last_days:, :]
                x_ = range(3, dataset.shape[0])
                x_ = list(dataset.index)
                ax.plot(dataset['BOLU'], label='Upper Band', color='c')
                ax.plot(dataset['BOLD'], label='Lower Band', color='c')
                ax.fill_between(x_, dataset['BOLD'], dataset['BOLU'], alpha=0.35)
                ax.plot(dataset['Close'], label='Closing Price', color='b', alpha=0.25)
                ax.set_title(f'{ticker}, {time}, Close, Bollinger Bands')
                ax.set_ylabel('USD')
                ax.legend()
            if column != 'MACD':
                ax.grid()
                self.graph1 = fcka(plt.gcf())  self.ids.graph_box1.add_widget(self.graph1)
                #self.ids.input_text.text = ''
```

```
                car1num += 1
            news_object = sdr.return_news_object(ticker)
            links, news_key = tuple(sdr.get_news(news_object))
            samples = tuple(sdr.get_news_sample(links))
            news_samples = sdr.get_trimmed_news(samples)

            infovals1, stock_news = sdr.get_data(news_key, news_samples, ticker)
            print(stock_news)
            infostr1 = f'''\r
Open: {infovals1['open']}          Mkt Cap: {infovals1['marketCap']}          Prev. Close: {infovals1['previousClose']}
High: {infovals1['dayHigh']}          P.E: {infovals1['trailingPE']}                  52 Wk. High: {infovals1['fiftyTwoWeekHigh']}
Low: {infovals1['dayLow']}              Div Yield: {infovals1['dividendYield']}        52 Wk Low: {infovals1['fiftyTwoWeekLow']}'''
            self.ids.graph_label1.text = infostr1
            self.ids.heading_label.text = f"{ticker.upper()}"
            self.ids.longName.text = f'{infovals1["longName"]}       {infovals1["financialCurrency"]}'
            difference = infovals1['open'] - infovals1['previousClose']
            if difference >= 0:
                self.ids.difference.color = (34/255, 139/255, 34/255, 1)
            else:
                self.ids.difference.color = (1, 0, 0, 1)
            self.ids.difference.text = str(abs(difference))
    def removeFromCarousel1(self):
        global car1num
        global column
        if car1num ==1:
            self.ids.graph_box1.remove_widget(self.graph1)
            self.ids.heading_label.text = ''
            self.ids.longName.text = ''
            self.ids.difference.text = ''
            car1num -= 1
        elif car1num == 2 and column == 'MACD':
            self.ids.graph_box1.remove_widget(self.graph1)
            self.ids.graph_box1.remove_widget(self.macd_graph)
            self.ids.heading_label.text = ''
            self.ids.longName.text = ''
            self.ids.difference.text = ''

            car1num -= 2
    def set_time1(self, instance, boolean, curtime):
        global time
        if boolean:
            time = curtime
    def set_col1(self, instance, boolean, curcolumn):
        global column
        if boolean:
            column = curcolumn
```

The following Classes handle the behaviour for the other pages of the application:

```
class ForecastPage(Screen):
    def __init__(self, **kwargs):
        super(ForecastPage, self).__init__(**kwargs)
class NewsPage(Screen):
    def __init__(self, **kwargs):
        super(NewsPage, self).__init__(**kwargs)
    def News(self):
        for i in range(len(stock_news) - 1):
            self.ids.news_label.text += f'{i + 1}.'
            for item in stock_news[i]:

                self.ids.news_label.text += item + '\n\n'
```

D.5 Kivy design file

Earlier in the report all the necessary Widgets and their respective properties were defined, this section will go over the *Kivy* design file or the *.kv* file that contains a large part of these widgets and their respective properties.

D.5.1 Box Layout

To start off, the following snippet defines a BoxLayout for some elements of the Home page. A BoxLayout defines a widget that is itself a collection of other widgets and can be customised extensively in order to meet the needs of the application.

```
<HomePage>:
  canvas.before:
    Rectangle:
      pos: self.pos
      size: self.size
      source: 'Images/background.jpg'
  BoxLayout: #MAIN BOX
    orientation: "vertical"              #Menu, BIG LABEL, Profile
    size: root.width, root.height
    padding: 10
    BoxLayout:
      size_hint: (1, 0.07)
      pos: (0, 0)
      Button: #Menu button
        size_hint: (0.11, 1)
        font_size: 22
        text: "Forecast"
        on_press: root.manager.current = "ForecastPage"
      Image:
        size_hint: (0.376, 1)
        source: 'Images/Big heading cropped.png' #STONK$
      Button:
        size_hint: (0.11, 1)
        font_size: 22
        text: 'News'
        on_press: root.manager.current = 'NewsPage'


    BoxLayout:                           # Search box and search button
      size_hint: (1, 0.04)
      padding: (100, 0)
      TextInput: #Search box
        size_hint: (0.8, 1)
        id: input_text
        text: 'Enter Input Ticker'
        font_name: "D:\Study_stuff\Compuper_science\Kivy\Fonts\Handwritten-Regular.ttf"
        font_size: 22
        multiline: False
      TextInput: #Search button
        id: input_window
        text: 'Enter Period'
        font_name: "D:\Study_stuff\Compuper_science\Kivy\Fonts\Handwritten-Regular.ttf"
        font_size: 22
        size_hint: (0.8, 1)
    BoxLayout:
      orientation: 'vertical'
      size_hint: (1, 0.055)
      Label: #Heading Label
        id: heading_label
        text_size: root.size
        halign: 'left'
        valign: 'center'

        font_size: 50
      BoxLayout:
        Label:
          id: longName
          font_size: 22
        Label:
          id: difference
          font_size: 22


```

D.5.2 Radio Buttons

Following this several *Radio Buttons* were defined in order to create the buttons that the user would interact with in order to curate the kind of data that would be displayed on the home page. This mainly includes things like technical indicators and time periods.

the following is the code for one such button that would be responsible for displaying stock data across a singular day. Similar iterations of this code were created for every available time interval and technical indicator

```
id: radio_layout
      size_hint: (1, 0.03)
      spacing: 30
      BoxLayout:

        Label:
          text: '1d'
        CheckBox:
          group: 'time_periods'
          on_active: root.set_time1(self, self.active, '1d')
```

D.5.3 Carousel

The graphs in the application have the functionality of being swipe-able meaning, several stocks can be monitored at the same time along with their respective statistics and indicators.

The *Carousel* widget was utilised to implement this functionality. There are a total of 5 iterations of the widget, one for each stock that can be displayed. here is one such example of the widget.

```
Carousel: #GraphStuff
      size_hint: (1, 0.4)
      BoxLayout: #Carousel Item 1
        orientation: "vertical"
        id: car1

        Button:
          size_hint: (1, 0.05)
          font_size: 20
          text: 'Add'
          on_press: root.addInCarousel1()
        BoxLayout:
          orientation: "vertical"
          id: graph_box1
          size_hint: (1, 0.8)
        Button:
          size_hint: (1, 0.05)
          text: 'Remove'
          font_size: 20
          on_press: root.removeFromCarousel1()
        Label:
          font_size: 24
          size_hint: (1, 0.3)
          id: graph_label1
```

D.6 Forecast page

Following the Homepage was the Forecast page, since there is very minimal user interaction for this page, very minimal GUI components had to be defined in order to create it.

The following script defines the necessary behaviours.

```
    Rectangle:
      pos: self.pos
      size: self.size
      source: 'Images/background.jpg'
  BoxLayout:
    orientation: 'vertical'
    BoxLayout:
      size_hint: (1, 0.05)
      Button:
        text: 'Home'
        on_press: root.manager.current = 'HomePage'
      Button:
        text: 'News'
        on_press: root.manager.current = 'NewsPage'
    Image:
      source: 'Images/ForecastHeading.png'
    Label:
      size_hint: (1, 0.8)
```

D.7 News

Following the Forecast and the Home page was the final page of the application, The News page. The News page is responsible for displaying financial news related to the selected stock and allow the user to visit the source of the news if necessary. This page too is very minimal in terms of user interaction and thus has very few defined behaviours.

```
BoxLayout:
    orientation: 'vertical'
    spacing: 20
    BoxLayout:
      size_hint: (1, 0.05)
      Button:
        size_hint: (0.1, 1)
        text: 'Home'
        on_press: root.manager.current = 'HomePage'
      Label:
        size_hint: (0.8, 1)
        text: ''
      Button:
        size_hint: (0.1, 1)
        text: 'Forecast'
        on_press: root.manager.current = 'ForecastPage'

    Label:
      size_hint: (1, 0.05)
      text: ''
    Button:
      size_hint: (1, 0.02)
      padding: (0, 40)
      text: 'UpdateNews'
      on_press: root.News()

    Label:
      size_hint: (1, 0.8)
      id: news_label
      font_size: 20
      text: ""
```