Using Neural Networks to Predict Forex Prices

Vincent Wang

March 31, 2019

Contents

Part I

Analysis

The analysis section gives some background on forex trading as well as outlining the end user and the proposed solution. A specification has been created and fully justified.

Please note that modelling of the problem on a high level such as E-R, data flow diagrams and description of the prediction models used have been shown in the Design section instead of the Analysis section as it was felt this arrangement read more logically for this project. Apologies for any inconvenience caused by this.

1 Research

1.1 Forex Trading

The Forex (foreign exchange) trading market is huge. Every day, $5.3 trillion US dollars are traded on the market combined - fifty three times the volume that is traded on the New York Stock exchange [1].

Successful traders are often those that have lots of experience with markets. Over time they gain some intuition or "feel" for how the market will act. That being said, markets move randomly. Trading, especially forex trading, has been likened to gambling because of this - it's risky and very difficult to reliably predict [5]. Even when a correct prediction is made, margins in forex are very small as the markets do not move a large amount so turning a profit is difficult, especially when taking into account the broker's fees to carry out the trade. In order to make any significant profits, large investments need to be made, which carries large risk with it.

To an extent, a trader can try to predict long term forex trends through following current events. For example, if a country is going through a period of political instability or uncertainty, a trader might choose to sell (or short) that currency. For example, the result of the 2016 "Brexit" vote caused the pound to fall to a 31 year low [12] *(See Figure 1)*. This aspect of trading strategy presents challenges for algorithmic trading as it is difficult to inform a program about the political climate of a country.

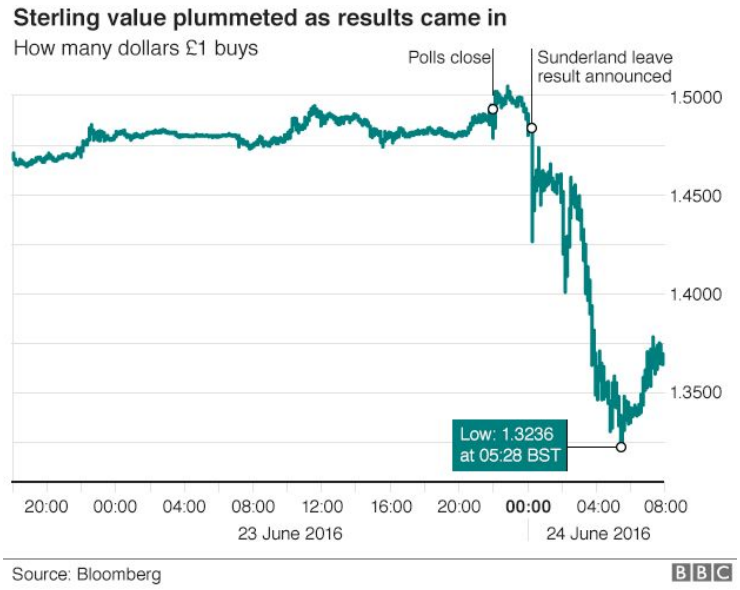

Figure 1: Pound against the dollar around the result of the Brexit Vote [12]

1.1.1 Mechanics of Trading

When trading forex, we talk about trading currency pairs [5]. A currency pair is represented in the form *base currency/quote currency* and its value is how much of the quote currency the base currency buys. For example, EUR/USD = 1.2500 means that the Euro buys 1.25 US dollars. When a pair is bought or sold, it involves buying ("going long") on one currency while simultaneously selling ("going short") on the other. E.g. putting a buy order on EUR/USD means going long on euros while going short on dollars.

Every time a currency pair is traded on the forex market, its price changes. If the price rises in a certain time frame the market described as bullish, if it falls it is described as bearish. The goal of a forex trader is to to try predict these changes and open buy or sell orders so that they can make a profit off the market movement when the order is closed. For example, if a trader thought that the price of EUR/USD was going to rise (the Euro strengthens against the dollar), they would buy EUR/USD. If the price of the Euro rises against the dollar, closing the trade at that point (selling the euros) would result in an overall profit - the of euros bought when closing the trade are more valuable than when the trade is opened, and so is worth more in whatever currency that the account is denominated in. If a downwards movement is predicted, then one can sell at the higher price and buy back at a lower price to make a profit.

For day traders (the group for which this project is aimed at) one needs to open

an account with a brokerage firm. An account is opened with a base currency - the currency with which is used to buy/sell assets and the currency which profits/losses are given in. To carry out trades, one deposits money into their account with the broker. Brokers can make money in a number of different ways, including putting commission on trades (which is discussed below) or buy offering services such as data analysis or advice for a monthly fee.

1.1.2 Buying vs Selling

In trading, one does not have to previously own an asset to sell it. Shorting a currency works by borrowing the specified amount of the currency from your broker, agreeing to buy it back in the future at the future price.

Because of this, shorting has an inherent difference to buying in forex. When buying, you are betting on the price of a currency pair to rise. The worst outcome of a trade is that the value of your trade goes to 0 as the price of the pair does i.e. you lose all of your initial investment. When shorting, there is theoretically no limit on how much you could lose. The price of a currency pair can keep rising, and with it the amount you need to repay when closing the trade does [13]. In practice, the value of a pair will not keep rising indefinitely, however this is still important to consider when shorting.

To protect against the danger of this, you can set a stop order when you start a trade.

1.1.3 Broker Fees

Broker fees can come in a number of different varieties. Disregarding fees a broker might charge for advice or other services, two common fee types are spread and commission [16].

Spread is the difference between the buy and sell price of a currency pair quoted by a broker. This difference is given in pips - the fourth decimal place of a quoted price.[1] Spreads can be fixed or variable depending on the broker. Variable spreads could depend on market volatility or trading volume for example - if a currency pair moves a large amount, a broker would prefer to set a larger spread.

Commission can come in fixed and variable forms as well. Fixed fee commissions tend to be very large, and targeted for people trading at high volumes. Variable fees are dependant on the volume traded, and so offer a good middle ground for all traders. Because of this, variable fees are growing in popularity [9].

[1] Imagine you are trading EUR/USD. The charts show a value of 1.2000 however your broker quotes two prices - a buy price of 1.2002 and a sell price of 1.2000. In this case we would say the spread is 2 pips. The small percentage on top of the actual value of the currency for the buy price is how the broker can make money off the spread.

1.1.4 Trading on Leverage

As discussed above, currencies on the forex market do not move large amounts, and so huge investments are needed to make any non insignificant profits. To help with this individuals can trade with leverage from their broker.

Trading on leverage is the act of borrowing money to boost the size of an investment. It acts as a multiplier on the original investment, increasing both the potential profits and losses from it. For example, if 100 worth of USD is bought with 50:1 leverage, the trade has a value of 5000. If the trade is closed when GBP/USD has moved up 20 pips, 10 is made instead of the 0.20 made if leverage is not used. [7]This also works the opposite way however - if GBP/USD moves down 20 pips, then 10 is lost. Again, we can use stop orders to help protect against the risks of this.

1.1.5 Stop Orders

Stop orders can be used by traders to decrease the chance of loss on a trade. Stop orders are instructions for a broker, telling them to close a certain trade when the market has reached a certain price. Stop orders can be used to both minimize losses and secure profits. e.g. if after buying a currency pair, the value starts to decrease, a stop order lower than the initial value can protect from the initial investment depreciating in value too much. On the other side, if a trader is buying a pair and is happy to "cash out" when they have made a certain amount off the market movement, they can set a stop order with a price higher than that bought, to protect from negative impact of potential future downward movements. Stop orders could be used for volatile markets, or on trades that won't be able to be appropriately monitored by the trader. [10].

On the other hand, buy orders can also be set depending on the broker, which allows users to automatically initiate trades when a price reaches a certain price.

1.1.6 "2% Rule"

Stop orders can be used to help follow the 2% rule - a strategy used to balance risk and reward in which a trader risks no more than 2% of their available capital between all trades at any one time. When creating stop orders (and trading on leverage) one might use this to decide on the price to set the stop order at. [14]

1.1.7 Open High Low Close

When, trading it can be useful to look at other meta data in addition to the raw exchange rate to inform what action to take. For example, if there is an

indication that the market is very volatile at a point in time, one might wish to hold off on a trade as a prediction could be more likely to be false.

Open High Low Close (OHLC) data gives us some indication of the volatility of an asset during a particular time frame. Open/Close prices are the price of the asset at the start/end of the time period. High/Low are the greatest and smallest prices within a time frame. [15]

These charts are typically represented in two ways. The first is simple known as a "bar chart". It has two short horizontal dashes - one line pointing to the left (back in time) at the opening price and one line pointing to the right (forward in time) indicating closing price. The range (high/low) is given by a vertical line with one end at the highest price and another at the low [11]. *(See Figure 2)*

Figure 2: Example of an OHLC Bar Chart [11]

The other is known as a "Japanese Candlestick Chart" a thin vertical line represents the range, and a thicker vertical box represents the open and close price [2]. *(See Figure 3)*

Figure 3: Example of an OHLC Japanese Candlestick Chart [2]

Both of these graphs (especially Japanese Candlestick) are usually colour-coded to help distinguish between bullish and bearish movements.

1.1.8 Why Trade Forex?

Forex trading attracts people for different reasons. One thing that makes forex attractive is that because movements are small and leverages offered are much larger than those on the stock market. This allows people to start trading effectively with relatively small sums of money, making the market more accessible to the general public using day trading strategy. Unlike the stock market, forex is also open 24 hours a day (although retail brokers close on weekends - from 10pm Friday to 10pm Sunday UTC time).

Others however might not trade forex with the intention of directly making money. If a trader was trading US stocks, they might be worried about the potential decline/volatility of the dollar. To offset this - allowing them to still make money off their stock trades overall, they could short US dollars against the Euro [8]

2 Outline of the problem

As discussed above, forex trading is a very difficult and risky way of trying to make money. Not only does a user need to predict the movement of market correctly, but they need to make predictions good enough to cover the broker fees, whether they be monthly or per trade. Because of this, retail brokers show stats of up to 80% of accounts registered with them losing money overall [6].

For someone new to trading on the forex market, trading can be especially challenging. Some of the things that make forex so appealing can also be the main pitfalls. For example if trading on leverage is not used responsibly (e.g. used with stop orders and within the 2% rule) it can be very easy to end up losing money quickly. With the market being open 24/5, traders might be tempted to hold trades overnight, during which time the market could shift dramatically. In addition "tips" for minimizing risk such as the 2% rule of thumb are likely to not be practical to follow as the size of new user's portfolio is likely to be small.

2.1 Current solutions

Professional forex traders will often use a wide array of analytical tools to aid in carrying out trades. However, such tools are usually either too expensive to justify using for a new trader, or private and bespoke to a company. Either way, many home traders are unlikely to be able to access them.

3 End User

The target audience for the project is forex day traders using scalping/intraday strategies - individuals carrying out short term trades on the time scale of hours at home who want to get a larger picture and be better informed before carrying out trades.

As discussed such users might not have some of the more advanced analytical tools that a trader working in a hedge fund might have to help them make successful trades more reliably. Thus this could be a very useful tool for them.

4 Proposed Solution

The solution should be an aide to an amateur day trader, giving short term (intraday) predictions in 15 minute intervals. It will take in real-time 15 minute data from previous prices in the market, and give some form of prediction of price movements for EUR/USD for a number of different points in the near future (e.g. 30 mins, 1 hour, 2 hours etc.). Neural networks will be used to make these predictions.

Day trading was chosen for a number of reasons. Firstly from a technical perspective, longer term predictions were thought to be more difficult to carry out as they would likely need to consider more data. It would be difficult to achieve effective solutions for this as the hardware availiable was relatively limited. In addition longer term predictions are more impacted by factors such as political climate surrounding the two currencies in a pair. This is difficult to quantify and thus difficult to inform a network or any algorithm about. Given that prices should not move large amounts in a given day of trading the possible scope of the price predictions limited and thus easier to do.

From a user perspective, longer term trades also carry more risk so generally amateur/home traders (the target end user) will make shorter term trades. In addition, an end user will be able to see payoffs much faster, getting a sense of the usefulness of the solution within the day unlike with predictions made on the scale of days or months. Thus a client base is more likely to be built as the users are able to trust the solution faster.

The solution should include a web frontend that displays predictions along with measures of the accuracy of predictions in a graphical form to give users more information with which to make a judgement. This will include an API, with which a user should be able to get all data shown on the web page at a point in time in numerical form to be able to manipulate however they desire.

Originally it was thought the solution would be one which carries out trades directly, however this was thought to be both too difficult as well as too limited

- directly carrying out trades would require committing to a single api broker which could be problematic e.g. if the broker goes out of business. In addition, it was thought that there would not be anyone, particularly in the target client base. who would be willing to use it as it would mean entrusting one's assets to a service they don't have much control of.

5 Specification

1. **Predictions** - *see 6.5* for more details.
   (a) Predictions should be made for times at least 4 hours in the future.
   (b) The networks must be able to take in real-time 15 minute data from the EUR/USD market from 10pm on Friday to 10pm on Sunday (UTC time).
   (c) All predictions must be made within 30 seconds of receiving the real-time price data.
   (d) The recent accuracy of predictions should be able to be calculated in real time. The measure of accuracy again depends on the type of outputs the network gives and so this will be further elaborated on later in the document *(see 6.5)*
   (e) (Desirable but not essential) All trained networks should be compared against specific, measurable critea on unseen test data in order to be able to be used in the final implementation. These criteria cannot be spcified at this point in time as it will be based off the final type of network being used, in order to be used in the final implementation *see 6.5*.
2. **Webpage**
   (a) The webpage should display predictions of prices as well as previous price data graphically
   (b) The webpage should display the recent accuracies of each network's predictions graphically
   (c) The site should self-explanatory and relatively intuitive for a day trader to use, without too many specifics of its own. Any features and usage of the site should be explained in an "about page".
3. **API**
   (a) Using an approved, random API key, a user should be able to retrieve all data of predictions and their accuracies numerically in an existing structured format which is easy to work with.

(b) A user should be able to request a unique API key with little resistance using only their email. A value entered that has already been used or that is an invalid email addresses will not get an API key

(c) The user's email should never be stored or sent in plaintext to maintain user privacy.

(d) API requests that are made too frequently should not be serviced

(e) If a user is inactive (makes no requests) for a month, their record will be removed

(f) All data of user requests should be stored.

(g) Daily metrics such as number of active of active users, total number of requests, number of requests that weren't serviced should be displayed to the console.

5.1 Justification

Justification of the specification points above are as follows

1. **Predictions**

   (a) 4 hours was thought to be a useful timeframe for day traders - it would fit with common trading timeframes. [4]

   (b) Predictions should be shown for all open hours of the market (24/5). The EUR/USD currency pair was chosen as it is one of the most widely traded and thus the potential user base for the project is large. In addition, this makes sourcing real-time and historic data easier.

   (c) 30 seconds was thought to be an appropriate max processing time given the timescale of the predictions. This should also help ensure the server does not get overloaded during operation.

   (d) Given the random nature of the forex market, a network could provide better or worse predictions depending on the market behaviour at a certain point in time. Thus having some measure of recent accuracy could be useful to the user, and likely more so than the accuracy of the network on the test data used to assess the network using training.

   (e) (Desirable) Networks should be giving "useful" to an extent. However as the forex market is stochastic, it is thought that quantitative "usefulness" may not always be appropriate. e.g. A network predicting the price of the market at a point in time that is good at accurately determining the direction of the movement may not be as precise than another which mostly quotes the close price and so has

converged on a smaller "loss" [2] when training When determining if a network should be used in the solution, qualitative impressions of the network's ability to act as a helpful aide should be considered.

2. **Webpage** A website was chose for the frontend as it is more accessible being platform independent. Additionally most trading services such as brokers and sources of data use web pages so it would be a familiar format for users.

   (a) A graphical format is far easier and quicker to interpret that numerical data. This will allow users to get a feel for the predictions without getting bogged down in the specifics. Having the previous prices along with the predictions will also help users get a better sense of the predictions without having to change tab to look at price data separately.

   (b) Displaying accuracies will allow the user to make a better judgement about the predictions. They should be shown graphically for the same reasons as those stated above.

   (c) The barrier for entry for using the site specifically should be low in order not to deter users. The site should not require any technical knowledge beyond that needed for day trading.

3. **API**

   (a) Returning data with an API allows the user more flexibility in how they wish to view/handle the predictions. An existing format should be used so that users will have experience/can easily find documentation for how to process the data they receive.

   (b) Using an API key allows for authentication of a user - policing requests and ensuring the user is verified without much overhead.

   (c) It is thought that the users will not need to be contacted, thus emails need not be stored in plaintext. To keep user privacy, emails should not be sent in plaintext either to reduce the impact of MITM attacks.

   (d) Not servicing requests made too frequently will help ensure good user etiquette and minimize server load.

   (e) Removing inactive users will help reduce the amount of data stored making the API framework faster as querying the database for valid API keys will take as much time.

   (f) In keeping a log of all API requests, metrics regarding the use of the API will be able to be calculated to get some insight into how users use the service and what they might need from the service in the future, aiding future development.

[2] The "loss" or "cost" is a calculation made from comparing the network's outputs to the expected outputs

(g) Some of these simple metrics well help give a sense of the size of the client base and the usage over time.

Part II

Design

The design section outlines the design of the site and API on a high level, as well as giving details of the design and iteration of the networks during the training process.

A diagram of the general overall structure of the project is shown below: (See 4).

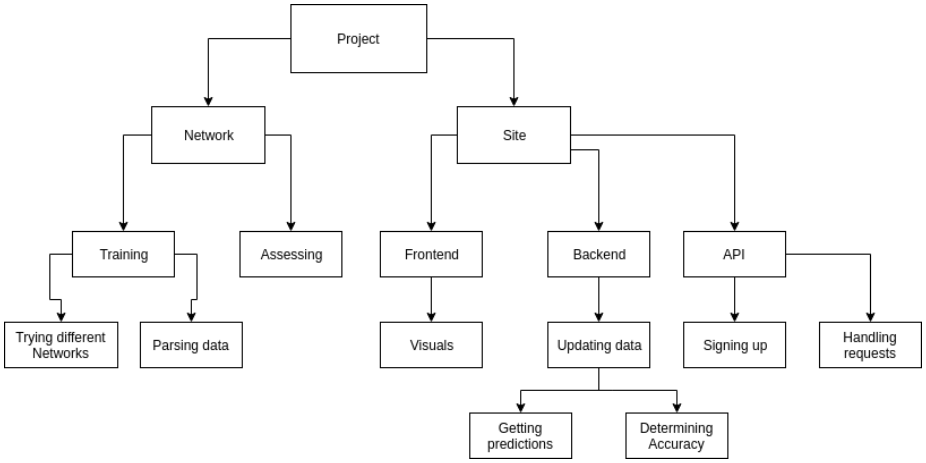

Figure 4: Overview of the structure of the project

6 Networks

6.1 Network Frameworks

There were two main contenders for neural network frameworks to use - Pytorch and Tensorflow. Both of these frameworks are written for python and are widely used, so very well documented.

In the end Pytorch was chosen for its simplicity, versatility and speed of training in comparison to other frameworks such as Keras. The more high-level features of Tensorflow to do with deployment or distributed training were not needed and thus it the larger initial cost of learning to use it was not worth it.

6.2 Data in/out

Each model should be able to work with 4 points of data at every time unit: open, high, low, close, or less. This is to fit the structure of incoming/real-world data provided by the AlphaVantage api (FX Intraday).

To train/test our agents, a dataset of open/high/low/close/volume from a six year period (2010 - 2016) on the EURUSD market with 15 min intervals will be used. We will start tests using an 80/20 split of training data to test data, as this is a good balance between ensuring that the agent learns enough during training and accurately assessing the agent's ability [17].

Neural networks work better when inputs are normalised i.e. take on values between -1, and 1. To achieve this (or at least a proxy for this), raw input values were changed to:

$$(value - meanClosePriceOfWindow) \times 100 \tag{1}$$

The window is the time range for which the network is given inputs. Even for large windows, tests showed that inputs to the network would very rarely exceed 1 or -1.

The initial plan was to let the network choose from one of three options - the price at a timestep will be within the spread of, or greater/lower than the spread of the most recent timestep. The spread was chosen for this as broker fees are often in the form of spread *(see 1.1.3)* - the price must "clear" the spread in order for a trade to return a profit and so seeing if a trade will go beyond the spread is useful. Each option would be assigned a confidence level for each new set of inputs - the percentage chance the network attributes to that outcome.

6.3 Proposed Networks

There are two problems to consider with the how inputs to the network. Firstly we need the network to be able to see many previous timesteps as this is what will likely allow it to predict prices correctly. Secondly, for each timestep, we have 4 data points - open, high, low, close, all of which should be considered.

A variety of input arrangements were proposed for the networks.

- **Feed-forward Networks:** All the feed-forward networks below would have a 1D convolutional layer as the first input layer, convolving each of the four inputs for a timestep (open, high, low, close) and producing one output for the next layer
  - **Normal deep learning network:** The network would be made up of standard feed-forward connections in which every ouput in one layer would be an input to every node in the next layer. While straightforward to implement, this requires many neurons to train
  - **Casually dilated convolutions:** This technique was inspired by Deepmind's wavenet [18]. It has the advantage of being able to look back on many previous timesteps while having a relatively small number of neurons to train. The structure of the network has the shape of a binary tree, where each layer has half the neurons of the last and each neuron takes two inputs from the previous layer. This allows the network to have a large lookback period without requiring many neurons.
  - **Exponentially increasing merged timesteps:** This setup is motivated by the assumption that the further back in the past the data is, the less relevant it is. For this, we will take current set of 15 min data (0 to -15), the set of data from -15 to -30, then -30 to -60, -60 to -120, -120 to -240 and so on (if needed). To merge timesteps, we take the open value of the timestep furthest back in the past for the window we are looking at, the close from the most recent, and the maximum high value and minimum low value throughout the range.
- **Recurrent Networks:** Recurrent networks are good at solving problems that have a sequential nature, such as time series problems. Because of this, they are very appropriate for the problem at hand, however they take longer to train [18].
  - **LSTMs**: LSTM (Lon Short-Term Memory) networks are especially good at solving sequenced problems as each LSTM unit in the network chooses to remember of forget certain values depending on how important the network deems the value to be.

It was decided that a feed-forward network with "normal inputs" and a recurrent network should be tested before the other feed-forward networks as they would

be more difficult to implement.

6.4 Training Process

6.4.1 First Network

The initial model was a normal deep learning network. N previous sets of OHLC data were fed to the network and 3 outputs were given, representing the probability the network gave to the price moving up, down and staying within the typical spread price.

As discussed above, the first layer was a 1D convolution that created one output for each set of OHLC data. For the other hidden layers, a number of setups were tested, with usually a starting layer of 64 neurons. The output layer had three neurons with a softmax [3] applied so the network outputs represented the probability the network assigned to the price moving up, down, or staying within the spread. The network was initially setup with the SGD optimiser, with a small learning rate of 0.01 as is customary for deep supervised learning.

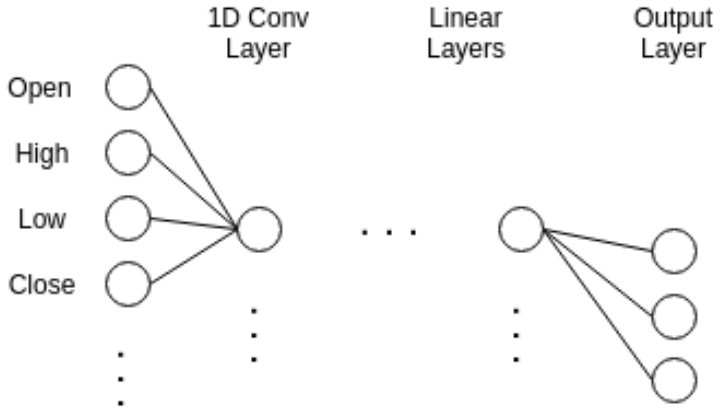

Figure 5: Feed-forward network Diagram

As a preliminary test, it was decided the network should try to overfit [4] on a small batch first. However, the network greatly struggled with this. On every run the network would either converge on a solution that assigns equal probability to each outcome, or one that always assigns 100% to the price increasing or decreasing.

[3] A softmax is a function that normalises all the outputs of a layer. This is often used to get probabilities from an output layer in prediction-based models.

[4] Overfitting is where a network learns the test data and corresponding outputs it is given instead of being able to generalise and learn which features of the inputs cause certain outputs, thus performing very badly on unseen data. This usually happens when the size of the test data is too small

Eventually, using the Adam optimiser instead of SGD, the network managed to overfit on the small batch of 20.[5]

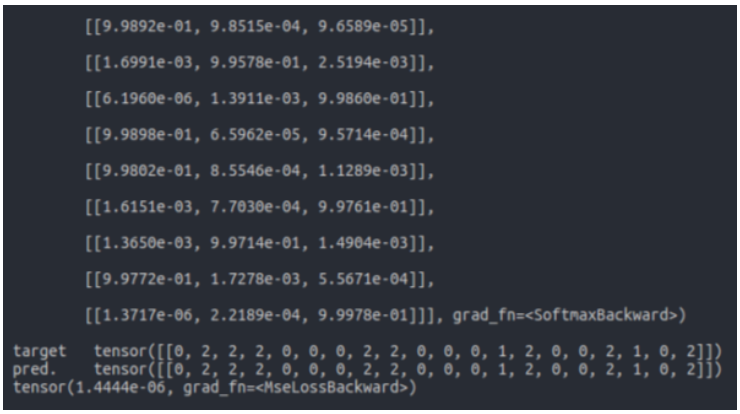

Figure 6: Output from running the overfitting test.

It was thought that the reason Adam worked was because the problem is quite complex and thus SGD (steepest gradient descent - which takes steps only in the direction of the steepest gradient) was less likely to have found the global minimum of the cost function, whereas Adam, which is stochastic, is better at exploring the landscape of the cost function and so more likely to find the global minimum.

When attempting the actual training (using the entire training dataset), all runs gave unsatisfactory results. At best networks gave the correct prediction around 49% of the time. Given that at the 15 minute interval the price moves up/down around 45% of the time and stays within the spread the remaining 10%, the network was not giving desirable predictions (the network was barely doing better than random guesses).

The approach that was taken to this problem of mapping the inputs onto one of three discrete outputs was influenced by classic classification problems such as recognising images of handwritten digits in which there is a clear "correct" mapping between the inputs onto one output. However forex prices are stochastic - it is possible to determine with certainty given a set up of inputs what the "correct" output should be so it was thought that training a network using the three discrete outputs (price up, down, same) represented as a [1, 0, 0], was causing problems during optimisation.

It was decided that a network with a different structure should be tested.

[5]In Fig. 6 the rows above show the output tensors from the network for the last few inputs. Below the expected outputs vs. the network ouput with the largest value (the network prediction)

6.4.2 LSTM Network

It was decided that the next test should be an LSTM network that predicted the price itself. N individual sets of OHLC values would be fed one by one into the network with one LSTM layer and one single output neuron. The Adam optimiser was used with learning rates from 0.05-0.4.

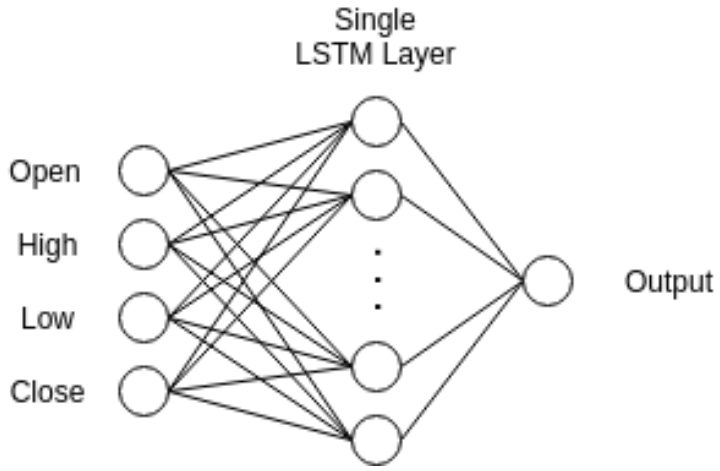

Figure 7: LSTM Network Diagram

The network produced what were thought to be far more desirable results, with initial tests able to predict the price in 15 mins correctly within the spread (taken as 0.7 pips) roughly 60-80% of the time with a limited number (200-300) training iterations. This was initially surprising as it was thought that predicting the price itself was a more difficult problem than just predicting the discrete movement of the price.

When graphing the prediction values against the actual prices at that timestep however, there was a concern that the networks were only achieving these results as it seemed to be just quoting the most recent close price for as the prediction for future timesteps.

In Fig. 8, the predictions series looks almost exactly the same as the actual price series, just displaced 2 timesteps.

There were a few ideas about why this might have been the case and how to improve on it.

- **Too few neurons** The network was not large enough and so couldn't learn the more complex behaviours required.
- **Window size was too large** Too much data was being fed to the network. With so many inputs it was difficult for the network to converge on a valid solution other than quoting the latest close price. Additionally,

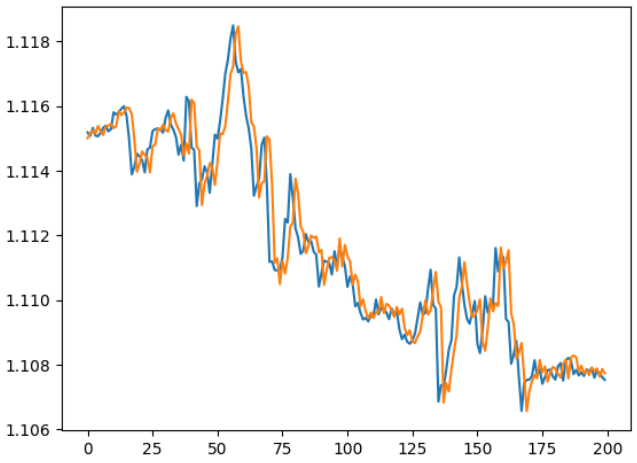

Figure 8: Sample prices (blue) against the prediction (orange) made for the price at that timestep 30 minutes (2 timesteps) beforehand

network outputs were given as movements from the mean of the window so the larger the window, the greater the variance of the mean in relation to the final close price, which could be affecting the validity of the predictions.

- **Prediction time-scale too small given the data** The tests were first being done on 15 and 30 minute predictions. Given the data being fed to the network was in 15 minute intervals, it was thought that these predictions could have been too close in the future for the network to converge on effective solutions as the price would not have moved too far/any movements had too much associated noise. It was thought that over longer time-scales general trends would be able to be found and thus more effective predictions could be made.

It was found that adding more neurons could often result in a worse solution, likely because the network was too complex and the number of training iterations was too small to be able to properly explore the landscape of the fitness function. When decreasing the window size, it was found that the network could converge on smaller losses, and when doing tests on the predictions further in the future, predicted values were thought to be better.

In Fig. 9 predictions for four hours in the future are shown. While it does seem appear at first glance that the network mostly seems to be quoting the most recent close price, at many points we can see that it is both over and undershooting peaks and troughs - showing that it is observing trends and

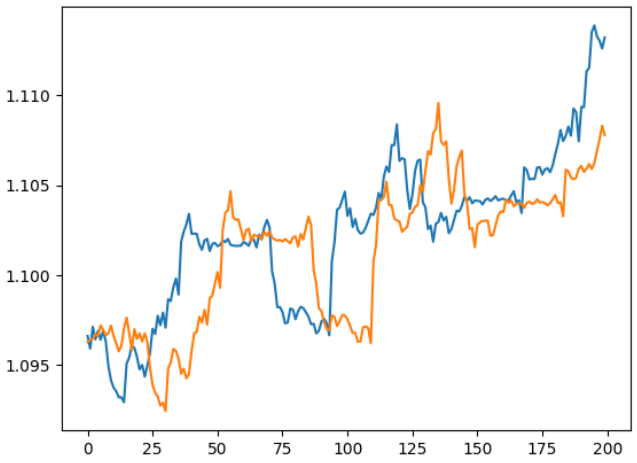

Figure 9: Sample prices (blue) against the prediction (orange) made for that timestep 4 hours (16 timesteps) beforehand

making decisions based on data from more than just the last timestep.

It was decided that this structure should be used in the final solution.

6.5 Measuring Success

As the format of the output of the networks had been decided, the specifics as to how a network should be assessed could now be determined. A network was measured by two criteria - the mean squared error between the predictions and the target values, and the percentage of predictions made that fell within the high and low prices of the relevant timestep.

As discussed above, it seemed as though the network could have been just quoting the most recent close price as the prediction. To test this - ensure the network was making "valid" predictions, a baseline was created by finding the MSE and percentage accuracy of a solution quotes the most recent close price as the prediction for a future timestep.

It was decided that that networks MSE would be evaluated against the baseline's MSE and that their prediction accuracy should be above 50% on test data.

6.6 Saving/Loading Networks

To be able to use the models, the trained parameters need to be saved. Thankfully, this is quite straightforward to do in pytorch. The current parameters of a network can be written to a file in a custom pytorch format to save a trained network. To load in a network from this file, when an instance of the original network class is initialised, the file can be parsed to the network.

Thus for each network being used, both the trained parameters and data about the network structure needs to be stored.

7 User Interface

The site should have three pages - the "main" page, which displays the predictions and accuracies, an "API" page where a user can get enter an API key as well as see a sample request/sample returned datafile and and "About" page in which the functionality of the site is explained.

When a user enters a valid email that is not currently in the database, they will be redirected to a separate page that displays their API key and some messages about good usage.

Using the above, the structure of pages of the site is shown in Fig. 10

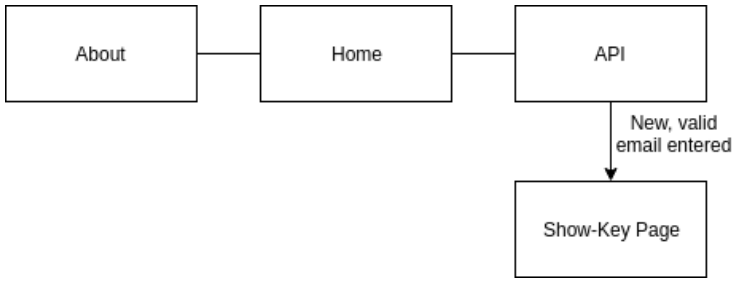

Figure 10: Diagram of the layout of the site

On all pages of the site there should be a clear navigation bar which allows the user to easily move between pages. There should be some indication of the current page the user is on.

7.1 Main Page

The predictions should be shown in a concise way graphically so that users viewing predictions via the webpage can more easily interpret the data. All data should be shown on one page.

It was thought that previous market data, predictions as well as some measure of the precision of the predictions such as standard deviation of error should be displayed on one graph. If possible, users should be able to choose how much historic data/how many predictions are being displayed at one time using sliders. The idea behind this is that it would help users interpret the data better - when all relevant data is displayed on the page the graph it was thought that the graph might appear to be too "noisy". On the same page, there should be a chart displaying the recent accuracy of predictions (how often the network correctly predicts the price within the actual high and low of the timestep).

On the page the current UTC time and the time of the data being shown should be displayed.

As the market is closed on weekends/AlphaVantage doesn't update data over the weekend, there should be a message on the home page to tell users that prices aren't being updated if it is between Friday 10pm and Sunday 10pm.

A mock-up of the main page is shown in Fig. 11

8 API

8.1 Data Returned

When an API call is made, the server will check if the API key used is valid (is in the list of API keys being used). If it is, then data will be returned, else an error message will be returned.

Data returned should be in JSON format and include all data that is shown on the webpage in numerical form. JSON was chosen as it is relatively compact, well supported and also allows for more logical representation and ordering of data with the use of hierarchy.

A request should not be served if requests from the user are being made too frequently.

8.2 Signing Up

On the webpage, there should be a text field where a user can enter their email to get an API key. If the email is valid, a random API key will be generated and sent to the user. If the email is invalid an error message should appear. If the email has already been used to sign up, the API key should be shown.

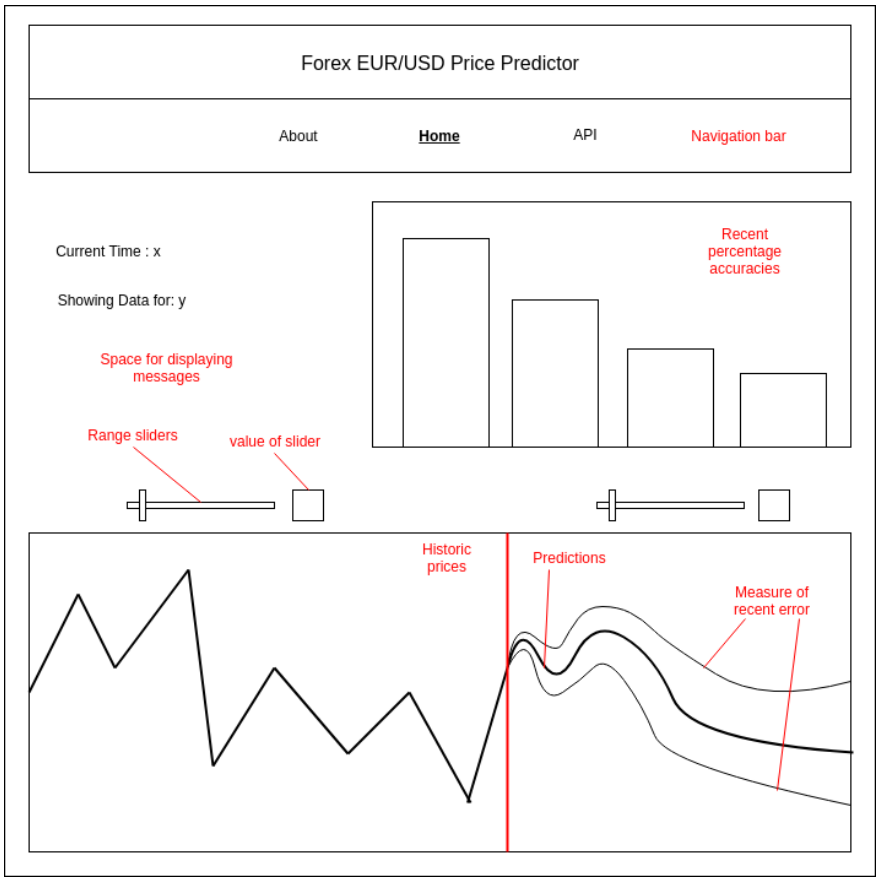

Figure 11: Mock-up of the main page. Labels are shown in red

8.3 Email Hashing

It is thought that the users would never need to be directly contacted and thus the user emails should never be stored or sent in plaintext to ensure user privacy. Were emails sent in plaintext, a "man in the middle" would be able to see the email.

To ensure this, when a user signs up for the service their email should be hashed on the client side with a random salt before being sent to the server. The user will then be shown their API key on the webpage. The hash algorithm used should not be too complex so as to ensure that there's not too much time between a user entering their email and receiving a response. The size of the hashed value should be large enough that there is a negligible chance of collisions occurring given the number of users who are expected to use service.

When the hashed email is sent the server on signup, the table of current users will be queried to find if the API key is already being used. If it is, then the user is returned their API key with a message for clarity.

9 Back-End

It was decided that Flask should be used as the web framework. Much like Pytorch, it is lightweight, versatile, easy to use and the advantages that other similar frameworks such as Django provide were not thought to be necessary for the end product.

9.1 Dataflow

There should only be one copy of historic data and predictions to ensure data is consistent in all places and updating these is easier. Because of this, the same datafile that can be accessed via the API should be used as the source of data for the webpage. This means that all data would be kept in a JSON format as prices retrieved from AlphaVantage are in JSON format. Thus it was decided that predictions returned from the site API would also be JSON.

9.1.1 Updating data

Every 15 minutes, the server should check for new data from AlphaVantage. However, data from the API does not update every 15 minutes on the dot, so the server should keep making API calls until the time of the data changes. AlphaVantage stops fulfilling requests if they exceed more than 5 requests a minute or more than 500 requests a day. Given that new data for each 15 minute timestep seemed to appear approximately 5 minutes late, a 45 second delay between requests was thought to be appropriate.

Once new data has come in from AlphaVantage, it should be processed and fed to all the models to produce a new set of predictions as well as compared with previous predictions to determine the recent performance of the networks

Fig. 12 shows a data flow diagram for the updating process.

Note above that "Price Data" (data from AlphaVantage) and "Predictions + Accuracy" (data sent by API) are represented as temporary data stores while "Predictions" are a permanent data store. For each new set of 15 minute price data coming in, the JSON files for the price data and the predictions + the accuracy are overwritten with the new values for the current timestep. It was thought that price data would not need to be stored as it will likely be readily available elsewhere if needed. Thus in only storing all predictions, data for predictions is not duplicatied, reducing data store and by using historic price

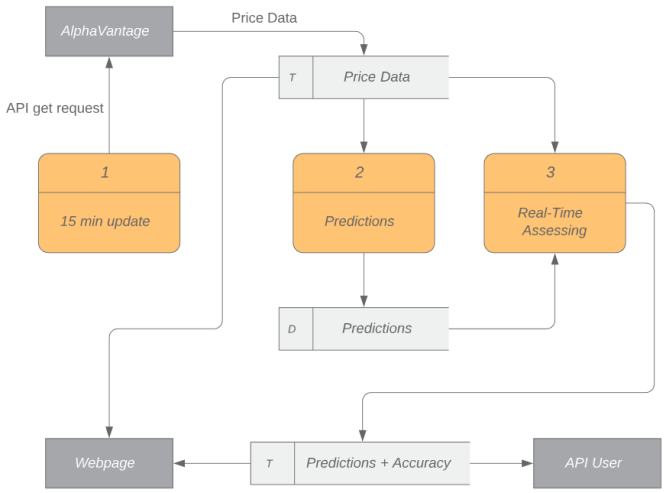

Figure 12: Data flow diagram for the update process

data any new desired statistics about prediction accuracy can still be calculated.

9.2 Database

To allow for the API functionality outlined, each user's email hash and the corresponding API key needed to be stored as well as the timestamp of the most recent API request to ensure the user is not making requests too frequently. It was thought that keeping a record of all requests could be beneficial as it would allow user metrics such as total and individual activity to be calculated, allowing for an insight into user behaviour *see 5.1*. Storing this data would mean that the timestamp of each user's last request need not be stored in the user table as it could be found by querying the list of all requests.

The predictions made at each timestep also needed to be stored to allow for real-time assessing of the networks' ability.

Thus it was decided that the tables required would be as follows

- **User:** Stores Email hash, API key, Date of signing up.
- **Request:** Stores the time of the request, the user ID and if each request was served.
- **Predictions:** Stores the timestep that the predictions were made (i.e. some multiple of 15 minutes less than or equal the time at which predictions were made instead of the time at which the predictions were made

themselves) and all the price predictions.

9.2.1 Normalisation

The database should be normalised in order to make querying and managing tables easier. It was decided that normalising up to third normal form would be appropriate as forms beyond this are rarely used in practise [3]. With the proposed structure, the database will be designed so that it passes all the forms below.

- **1st Normal Form:** Values in each field must be atomic i.e. each first stores one value. The only field for which this needed to be considered was the predictions table - each prediction should be in their own column.
- **2nd Normal Form:** Every non-key attribute of a table with a composite key, should depend on the whole key. The request field is the only table that is relevant to this as it has a relation to the user table. To help avoid this, the request table will just user a standard numeric primary key.
- **3rd Normal Form:** Fields have no dependencies on other non-key fields. The database is quite small overall - there are no problems with non-key dependencies.

9.2.2 Final Database Structure

Following the discussion above, it was decided the final database structure should be as shown in Fig. **??**. It satisfies 3rd normal form.

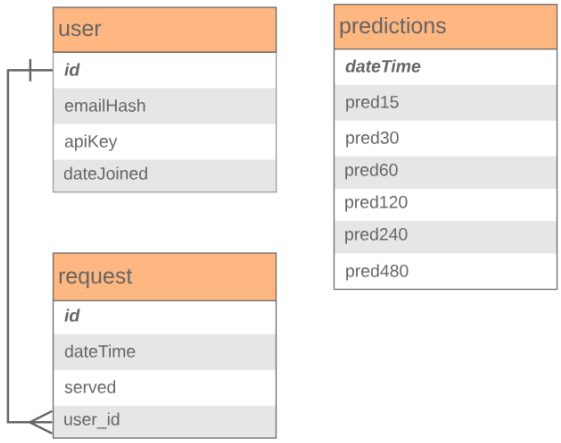

Figure 13: Entity relationship diagram

The user table is defined as follows:

user		
Field	Type	Description
id	INT	Primary key - uniquely defines each user
emailHash	VARCHAR (32)	Hashed hex value of user's email+salt
apiKey	VARCHAR (16)	Random alphanumeric API key used for requests
dateJoined	DATETIME	Date that the user signed up

The apiRequest table is defined as follows: An integer is being used as a logical boolean as SQLite does not have a boolean type.

apiRequest		
Field	Type	Description
id	INT	Primary key - uniquely defines each request
dateTime	DATETIME	The datetime at which that the request was made
served	INT (logical BOOL)	Whether or not the request was served (1 is yes, 0 if no)
user_id	INT	Foreign key to the user table

The prediction table is defined as follows:

predictions		
Field	Type	Description
dateTime	DATETIME	Primary key - the time at which a prediction is made is unique to each set of predictions
pred15	FLOAT	The predicted price for 15 mins in the future
pred30	FLOAT	The predicted price for 30 mins in the future
pred60	FLOAT	The predicted price for 60 mins in the future
pred120	FLOAT	The predicted price for 120 mins in the future
pred240	FLOAT	The predicted price for 240 mins in the future
pred480	FLOAT	The predicted price for 480 mins in the future

Part III

Final Design/Implementation

This section shows the final state of the solution. This includes screenshots of the final site, an outline of data structures and examples of algorithms used as well as other details of the implementation.

10 Front-End

The item on the navigation bar corresponding to the current page goes bold to give the user an indication of the page they are currently viewing. This helps make navigation of the site easier as discussed in 7.

10.0.1 Main Page

The final layout of the main page is shown in Fig. 14. The charts were all created with chart.js. Predictions and historic data is shown on the same graph. When the sliders are moved, the chart is redrawn and the values displayed beside the sliders change accordingly. The page meets all the requirments outlined in 7.1.

10.1 API

The final layout of the API page is shown in Fig. 15

If a user enters an email whose hash is not found in the database, they will be redirected to the "show key" page, shown in Fig. 16

If the email hash is found in the database, the page will reload and a message will be displayed above the text field. Fig. 17

If an invalid email is entered, a javascript alert will be shown and the page reloaded. Fig. 18

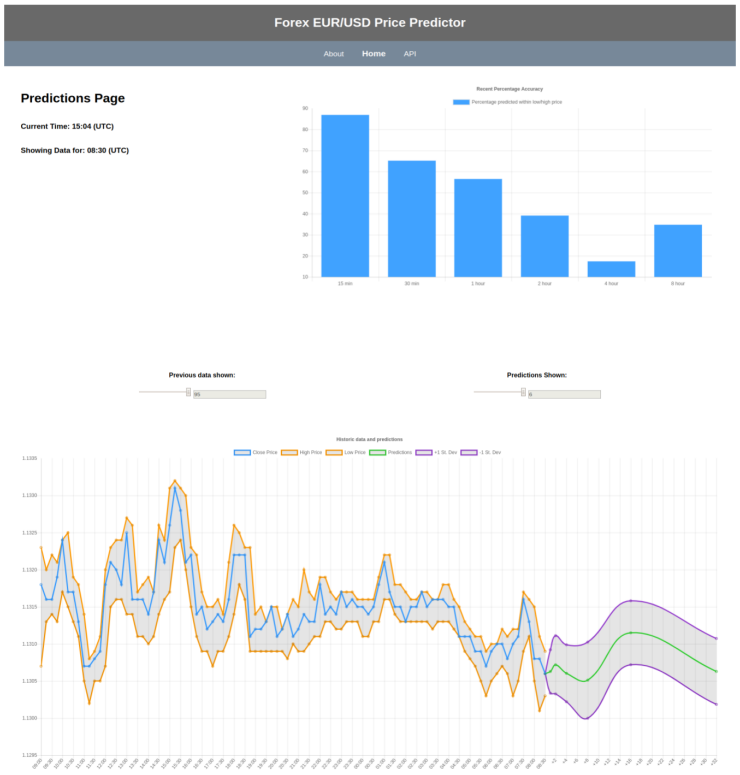

Figure 14: Layout of main page

Figure 15: Layout of api page

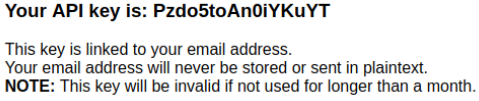

Figure 16: Show key page

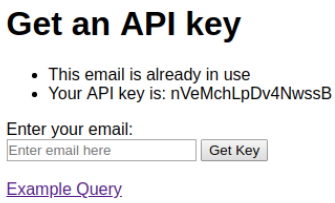

Figure 17: Used email message

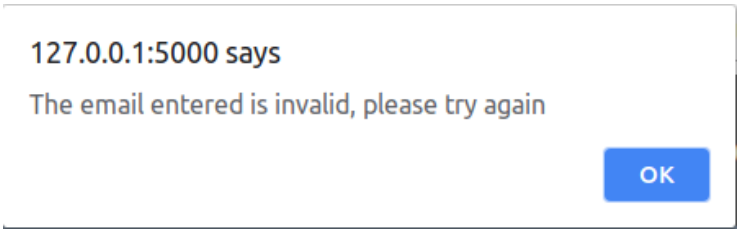

Figure 18: Javascript alert

10.1.1 Hashing

As discussed in 8.3 hashing should be done client-side to maintain users' privacy. It was decided that md5 would be a good hashing algorithm to use as it is (relatively) secure and quick, so hashing the emails this way would not add a noticeable amount of time to the time it takes for a request to be processed.

The algorithm for verifying and hashing an email is outlined below:

- *on the submission of a form*
  - *get value entered to the form*
  - *if a match is found when the value is tested against an email regex pattern*
    - *apply the salt to the email and send a post request to the server*
  - *else*
    - *show a javascript alert to the user, asking them to enter a valid email*

10.2 About Page

The final layout of the About page is shown in Fig. 19. The page includes links to the dataset used to train/test the networks, chart.js and alphavantage.

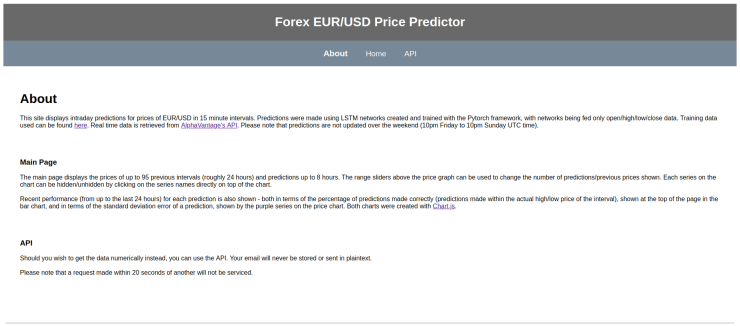

Figure 19: Layout of about page

11 Back-End

The Site's source code file tree is shown below.

```
site
|___static
|   |___json                //all json files used
|   |   |___data.json           //AlphaVantage data file
|   |   |___error.json          //file returned by API if error
|   |   |___invalidGet.json     //file returned by API if bad
↪  request
|   |   |___predictions.json    //file of predictions/accuracies
|   |   |___sample.json         //example of data in format of
↪  predictions.json
|   |___nets                //trained networks
|       |___  ...
|___templates           //html files returned by app routes
|   |___about.html          //about page
|   |___api.html            //api page
|   |___home.html           //home page
|   |___layout.html         //base html structure that the other
↪  pages extend
|   |___showKey.html        //page showing user their API key
|
|___config.py           //site configurations
|___database.py         //init the database
|___models.py           //SQLAlchemy database table models
|___app.py              //main
|___api.py              //all logic and routes for the API page
↪  and serving API requests
|___update.py           //price and prediction updates
```

11.1 Networks

When training the LSTM, network there still seemed to be a few issues with the validity of predictions - the networks still appeared to be converging on solutions that would more or less quote the most recent close price, and would end up performing worse than the baseline both in terms of percentage accuracy and MSE.

While the network didn't appear to be overfitting on training data, it was thought that early stopping could be useful to help make better predictions. By not allowing the network to fully converge on a solution, it was thought that it may be better at predicting trends instead and potentially perform better overall.

The final networks used had structures and performances as follows. For each timestep, the baseline stats are shown directly above the trained network. All final networks used only one layer of LSTM cells. Batch size used ranged from 10-5% and learning rate was altered from 0.1 to 0.01 depending on the batch size and if the network had already undergone training.

Timestep	Window Size	Layer number	Percentage Performance	Mean Squared Error
15 (15*1)	- 5	- 20	- 92.586	0.0047292 0.0047001
30 (15*2)	- 10	- 20	- 91.979	0.0091900 0.0091604
60 (15*4)	- 20	- 30	- 87.311	0.018476 0.018470
120 (15*8)	- 40	- 30	- 88.181	0.037538 0.037942
240 (15*16)	- 65	- 30	- 81.197	0.0047292 0.076780
480 (15*32)	- 90	- 30	- 69.199	0.14996 0.15115

All networks produced a percentage accuracy over 50% however the networks predicting the 2 and 8 hour price did not pass the baseline. It was thought that at least part of this was due to the hardware limitations of the machine being using to train the data. When attempting to train the networks with larger windows i.e. those predicting prices further in the future, the batch size has to be decreased significantly to the point where network predicting the 8 hour price was trained with a batch size [6] of 3%.

As the networks performed comparably to the baseline, it was still used in the final solution. Despite not passing the baseline however, the predictions made by these networks were thought to be very useful *(see 12.2)*.

Given the above, using a machine that has a graphics card with more memory, it is likely that "useful" predictions (predictions that have a lower MSE than the baseline) could have been made. However, there were limitations due to the amount of data AlphaVantage returned (100 timesteps reliably)and so a different real time price source/saving of historic prices would be needed.

When processing the historic data to feed to the network, a lot of care had to be put in to do this properly. For a given number of timesteps n, there are (n-

[6] During training, it is not practical to feed all the test data a network during each iteration (before trying to optimise to the network parameters). Because of this, a percentage or number of data is chosen randomly to be fed, giving an approximation of the networks ability. This means the "step" taken by the optimiser will likely not be the most appropriate however overall this saves time during training.

timeStepToPredict-windowSize) pairs of test data/targets as there is "padding" either side of every "current" price (the last price being fed to the network). For a given number of targets m however, there are m+windowSize datapoints that need to be considered. Because of the inconsistencies in the length of target and training data, care needs to be take when splitting data into training/testing data.

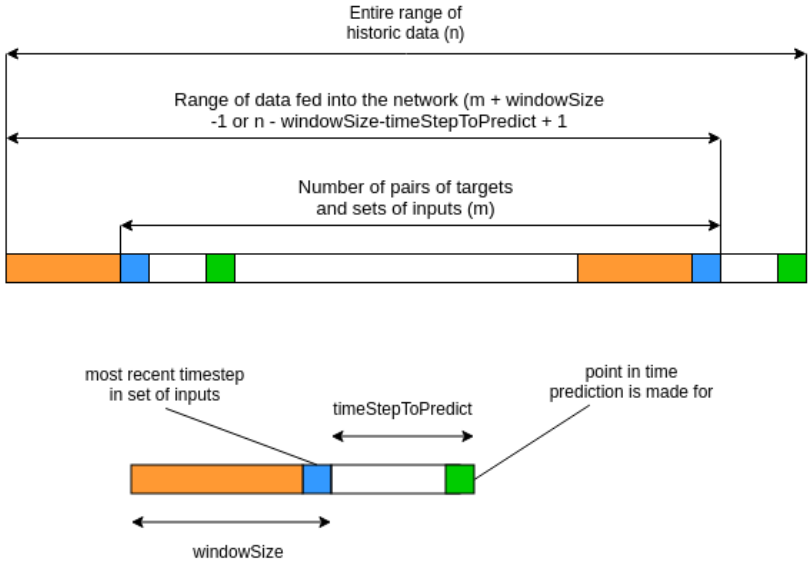

Figure 20: Diagram illustrating how raw data is arranged

To make sure that all the data is used, splitting of input data should be done as shown in **??**

The algorithm used to get all the data needed for testing/training was as follows:

- *retrieve data as list of records from csv*
- *convert the first (length of the returned records - the timeStepToPredict) records to pytorch tensors, take as "inputs"*
- *take the close price of the last (timeStepToPredict + windowsize) records as a pytorch tensor, take as "targets"*
- *get the mean for each window that can be made from the inputs*
- *split means, inputs and targets into training/testing data*

When training, each batch, was obtained like so:

- *determine the size of the batch*

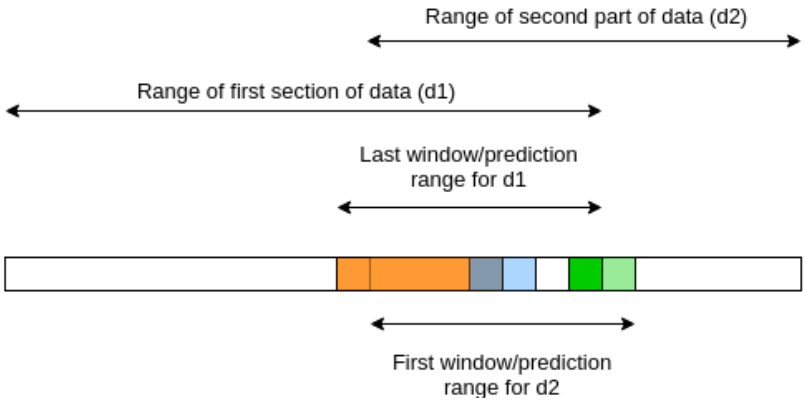

Figure 21: Diagram illustrating how a set of input data is split

- *get a random start index for the batch and find the end index*
- *split the targets/means at these indices, split the test data at the indices + windowSize*
- *for each potential window (length of test data - windowSize)*
  - *normalise the values in the window and the target using the mean of the window*
  - *add the normalised window to a list*

Each training iteration followed the process below:

- *get a random batch*
- *clear the hidden states in the LSTM cells*
- *clear the values in the optimiser*
- *feed the tensor of normalised batches to the network*
- *compute the MSE between the network outputs and the targets*
- *back-propagate and update the network parameters*

11.2 Updating data

The final updating process is shown in Fig. 22. The main path of the flowchart is shown in bold. If the json returned from the API request to AlphaVantage has the same timecode as the current values stored or if it isn't in the expected format, the current values are kept.

Figure 22: Flowchart of updating process

An example of some of the json data retrieved from AlphaVantage is shown below.

```
{
    "Meta Data": {
        "1. Information": "FX Intraday (15min) Time Series",
        "2. From Symbol": "EUR",
        "3. To Symbol": "USD",
        "4. Last Refreshed": "2019-03-28 01:15:00",
        "5. Interval": "15min",
```

```
        "6. Output Size": "Compact",
        "7. Time Zone": "UTC"
    },
    "Time Series FX (15min)": {
        "2019-03-28 01:15:00": {
            "1. open": "1.1252",
            "2. high": "1.1253",
            "3. low": "1.1246",
            "4. close": "1.1251"
        },
        "2019-03-28 01:00:00": {
            "1. open": "1.1248",
            "2. high": "1.1252",
            "3. low": "1.1245",
            "4. close": "1.1252"
        },
        "2019-03-28 00:45:00": {
            "1. open": "1.1250",
            "2. high": "1.1252",
            "3. low": "1.1246",
            "4. close": "1.1247"
        },
        ...
    }
}
```

11.2.1 Getting new Predictions

To make the design more flexible and easier to update, a general pytorch LSTM network class was defined that could take in all networks of the proposed structure. Each network was saved to a file whose name corresponded to the structure of the network it stored. The name was in the format "priceToPredict-windowSize-LSTMCellNo-LSTMLayerNo.pth" - e.g. "15-10-20-1.pth" is a network that gives predictions for 15 minutes in the future, taking in data from 10 previous timesteps and feeding them to 1 layer of 20 LSTM units.

Using the above, the algorithm for getting new predictions is as follows:

- *for each network file path*
  - *split network file path by '-'*
  - *create instance of network with correct structure using the split values*
  - *parse trained parameters at the file path to the network*

- *get data according to the window size specified in the file path and normalise the values*
- *feed the data to the network*
- *"un-normalise" the output of the network to get a prediction for the price.*

From a list of all the file paths of networks to be used, every 15 minutes, each file name was used to get the structure of the network, an instance of the LSTM network was initialised using this structure and the trained parameters of the network were loaded into the class instance. Incoming data was then normalised and fed to the network. Once all predictions were made, predictions are saved to the database and the performance of recent predictions were assessed. Finally the json file of predictions and accuracies with the new values.

11.2.2 Determining Recent Accuracy

The amount of historic data available at any one time is quite small (previous 100 timesteps) so potentially will not be many recent predictions that can be assessed at any one time e.g. when the site is first run after a period of time offline or on Monday morning where there will not not be any predictions from the past 24 hours to assess. Because of this, an "unbiased estimate" for standard deviation is calculated instead of the raw standard deviation.

$$\sigma \approx \sqrt{\frac{n}{n-1}\left(\frac{\sum x^2}{n} - (\frac{\sum x}{n})^2\right)} \tag{2}$$

The formula used is shown above, where x is the absolute error of each prediction and n is the number of predictions in the sample.

The algorithm for assessing predictions is as follows:

- *for each prediction timestep - i.e. for each of the 15min, 30min, 1h etc predictions*
  - *initialise variables totalCorrect, totalError, totalSquaredError*
  - *determine the time of prediction furthest back in the past for which there is actual price data*
  - *determine the most recent time of prediction for which there is actual price data*
  - *query the prediction table for predictions made between the two times calculated above*
  - *for each prediction record returned*

- *get the relevant prediction*
- *determine the time the prediction was made for*
- *get the actual OHLC prices for that time*
- *if the prediction was within the high/low price*
  - *add 1 to totalCorrect*
- *get the absolute error of the prediction-the actual close price*
- *add this to totalError, add the square of this to totalSquaredError*
- *calculate percentage and unbiased estimate of standard deviation*

11.3 Database

The SQLAlchemy package was used to manage the database. SQLAlchemy allows for the same control that raw SQL offers but has the advantage of allowing interaction with entities as python objects. This helped make requests easier to implement as well as helping to keep code visually consistent.

Some examples of database queries used throughout the solution are shown below.

At midnight, daily stats are retrieved and printed to the console. The following requests get these stats, making use of SQL's COUNT, BETWEEN and DISTINCT functions. To get the number of active users, a "crosstab" query is used.

```
1  #number users who have signed up in the last day
2  newSignups = user.query.filter(db.between(user.dateJoined,
   ↪  dayBefore, current)).count()
3
4  #total number of users remaining in the database
5  remainingUsers = user.query.count()
6
7  #number of users who have made a request in the last day
8  activeUsers =
   ↪  db.session.query(apiRequest.user_id).join(user).distinct()
   ↪  .count()
9
10 #total number of requests made in the past day
11 totalRequests =
   ↪  apiRequest.query.filter(db.between(apiRequest.dateTime,
   ↪  dayBefore, current)).count()
12
```

```
#number of requests that were rejected in the last day
rejectedRequests =
↪  apiRequest.query.filter(apiRequest.served==0)
↪  .filter(db.between(apiRequest.dateTime, dayBefore,
↪  current)).count()
```

When an API request is made, a "crosstab" query is made to get a user and the time of their last request.

```
#get the last most recent user/apiRequest datetime where the
↪  user's apikey is equal to the one entered
User = db.session.query(user,
↪  apiRequest.dateTime).outerjoin(apiRequest, user.id ==
↪  apiRequest.user_id).filter(user.apiKey ==
↪  apiKey).order_by(apiRequest.dateTime.desc()).first()
```

11.4 API

Luckily, flask makes sending static files to a user very simple so implementing the sending of the json data was very straightforward - the majority of the work came in the user validation and ensuring requests weren't being made too frequently.

11.4.1 Getting an API key

When a post request from the /api route is sent from the form back to the server, the server checks the form returned with the following algorithm.

- *if the isValid property of the form is 1*
  - *query the user table for the first entry with emailHash = emailHash returned*
  - *if the query returns an entry*
    - *refresh the page with a message telling the user the email is already being used and give them their api key*
  - *else if nothing is returned*
    - *generate a random alphanumeric api key.*
    - *redirect user to a page displaying the api key and some information about usage.*

11.4.2 Returning Data

When an API request is sent to the server (a request is made to the api/data route), the following algorithm is used to determine what data is returned to the user.

- *if the request is made with an "apikey" argument*
  - *if the apikey = "testkey"*
    - *return sample.json*
  - *else*
    - *make a crosstab query to get a user with the apikey entered and the time of their last request*
    - *if the query returns an entry i.e. apikey was found in the database*
      - *if this is the first request made by the user i.e. no datetime was returned in the query OR the last request made was more than 20 seconds ago, return predictions.json*
      - *add entry to the apiRequest table*
- *else i.e. if the query returned None OR the user has already made a request in the last 20 seconds OR no API key was given as an argument*
  - *return invalidGet.json*

An example of the returned predictions file is shown below


```
{
    "Meta Data": {
        "Time": "2019-03-26 08:30:00",
        "Recent Percentage Correct": {
            "+15mins": 86.95652173913044,
            "+30mins": 65.21739130434783,
            "+60mins": 56.52173913043478,
            "+120mins": 39.130434782608695,
            "+240mins": 17.391304347826086,
            "+480mins": 34.78260869565217
        },
        "Recent Standard Deviation Error": {
            "+15mins": 0.00029270882131829135,
            "+30mins": 0.00039269962956368866,
            "+60mins": 0.0003842059865736546,
            "+120mins": 0.0005134437797033753,
            "+240mins": 0.0004310034463049413,
```

```
            "+480mins": 0.00044411817761305156
        }
    },
    "Predictions": {
        "+15mins": 1.1306290909647942,
        "+30mins": 1.1307190022617577,
        "+60mins": 1.1306044206023216,
        "+120mins": 1.1305119970440864,
        "+240mins": 1.131150475591421,
        "+480mins": 1.1306289701163768
    }
}
```


If the user has not entered an API key, if the entered key was invalid or if a request was made in the last 20 seconds, the following file is sent


```
{
    "Error Message": "Either key is invalid or requests are
    ↪  too frequent."
}
```


If there is an error in writing to / querying the database, the following file is sent.


```
{
    "Error Message": "Something went wrong...  Apologies for
    ↪  the inconvenience, please try again."
}
```

Part IV

Testing

Further evidence for the tests can be found in the appendix. A google drive folder where video evidence of the tests can be found here: `https://bit.ly/2WyO9LO`.

All time-based tests were done by changing the system time, as is shown in the videos. Please note that the time changed was in GMT whereas the server was getting the UTC time. At points these dont match up when the test time passed between GMT and BST.

Any tests that included modifying the database or any stored files show the data in each file before and after the test.

12 Predictions

Testing the networks was difficult as there was no solid idea of how a well-trained network "should" act on given data as the forex market is stochastic. We can however evaluate the performance of each network on historic data as well as try to evaluate the networks' abilities to determine and follow trends using fabricated data.

12.1 Performance on Unseen Data

The following repeats the results shown in the final design section. *See A.1* for sample graphs of the test data against predictions made.

Timestep	Window Size	Layer number	Percentage Performance	Mean Squared Error
15 (15*1)	- 5	- 20	- 92.586	0.0047292 0.0047001
30 (15*2)	- 10	- 20	- 91.979	0.0091900 0.0091604
60 (15*4)	- 20	- 30	- 87.311	0.018476 0.018470
120 (15*8)	- 40	- 30	- 88.181	0.037538 0.037942
240 (15*16)	- 65	- 30	- 81.197	0.0047292 0.076780
480 (15*32)	- 90	- 30	- 69.199	0.14996 0.15115

12.2 Analysis of Behaviour on Simple Trends

The following tests examine how all the networks viewed together perform on simple trends. It was thought this would help give a better (qualitative) understanding of how the networks performed. 3 scenarios were used - straight line positive gradient, straight of line negative gradient (to test if behaviour is roughly symmetrical for upwards/downwards movements) and a sine wave.

In addition, tests were done where "noise" was included by randomly adding from -5 to 5 pips to the value that the trend should be. For all tests, the open/close price were values given by the function producing the trend and the high/low price were calculated by adding/subtracting 0.005 to the close price of a timestep.

12.2.1 Straight Line

Graphs of the networks' predictions on straight lines without noise were as follows. The close prices of the data fed to the networks is shown in blue and predictions are shown as lines connected between every price point and shown in orange.

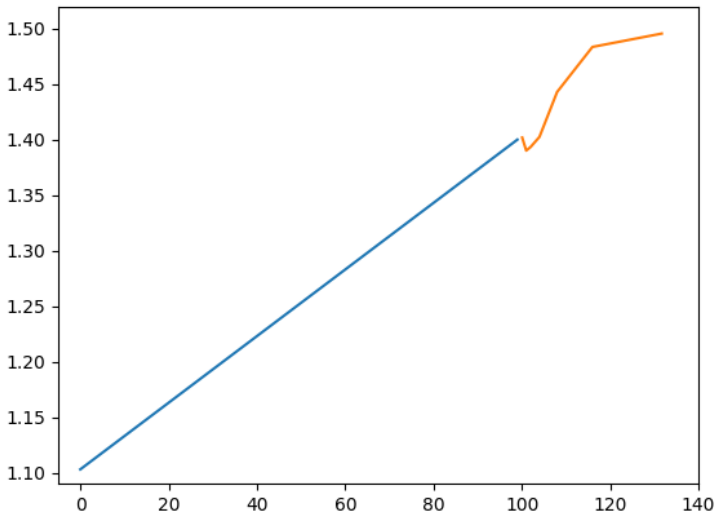

Figure 23: Predictions on a straight line (positive gradient)

From the above, we can see that the networks on the whole were very good at predicting the upwards trends. Surprisingly however, it appears as though the

prediction for 32 timesteps or 8 hours in the future (the final prediction) was actually the best at matching the trends but the predictions made for 1, 2 and 4 timesteps or 15, 30 and 60 minutes respectively were worst at this. It was thought that this is because the window size that these networks used were far smaller than the others. As the market is so volatile, If there is a steady price increase in the short term, it is quite likely that the price will in fact drop. This is potentially as a large number of traders on seeing this increase will start to short the asset to try to capitalise on this or due to traders setting buy orders on upper bounds of the price.

Additionally, it is very reassuring to see that behaviour does indeed appear to be symmetrical for price increases and decreases.

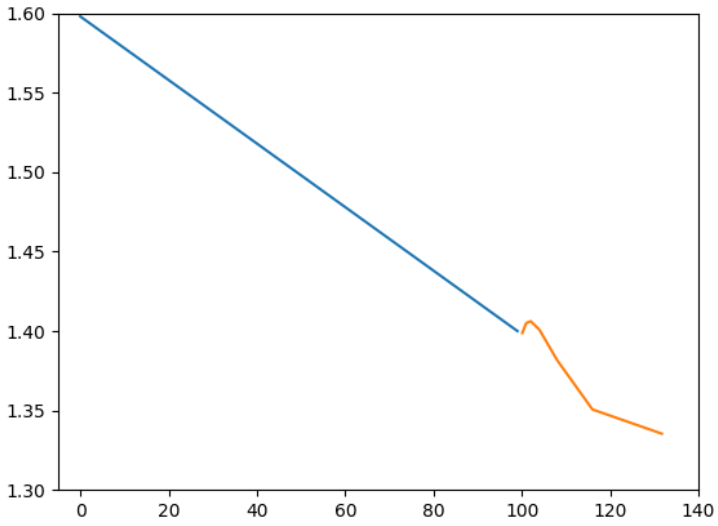

Figure 24: Predictions on a straight line (negative gradient)

Also very reassuringly, adding noise appears to have very little impact on the predictions, with the only visible differences seen in the shorter term predictions which is to be expected given the smaller windows they use.

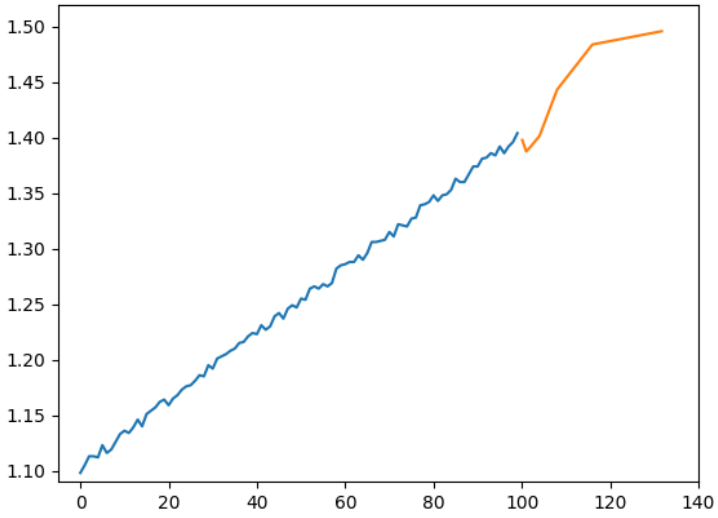

Figure 25: Predictions on a straight line with noise (positive gradient)

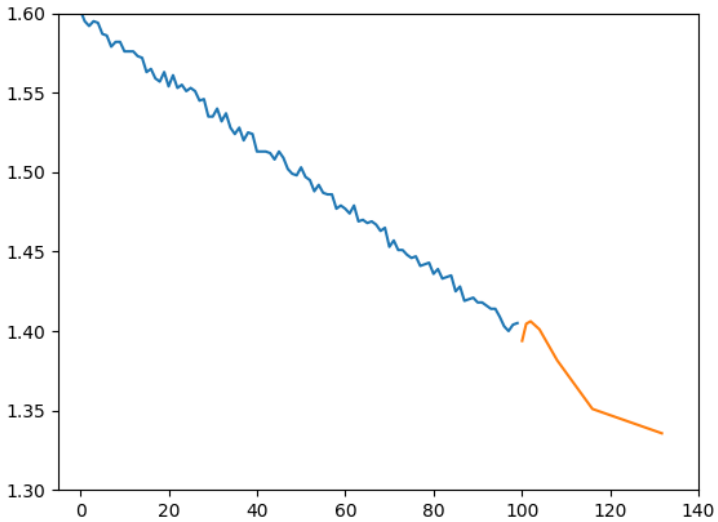

Figure 26: Predictions on a straight line with noise (negative gradient)

12.2.2 Sine Wave

For the tests on the sine waves, it was felt the most useful, and potentially most difficult test would be trying to predict future prices given data just before a peak or trough. This is to see if the networks were able to pick up on the steadily increasing or decreasing gradients and make predictions accordingly. Given the tests above showed that the behaviour was symmetrical for price increases/decreases, only one set of tests is shown below.

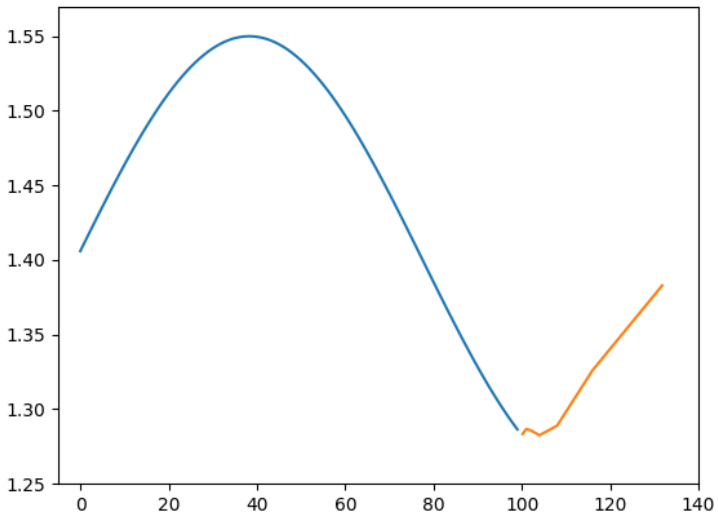

Figure 27: Predictions on a sine wave

Again we see that the shorter term networks have sometimes predicted movements in opposition to what would be expected, likely for the same reasons as above. The longer term prediction exceeded any expectations of how they would perform, determining the upwards trend surprisingly well. Again, it is shown that noise makes little impact to the networks' abilities to pick up on the trends in the longer term.

12.2.3 Comments on the tests

These tests show that the networks on the whole, were very good at predicting trends. It was shown that they were in fact giving "useful" predictions instead of just quoting the most recent close price. Initially it was surprising to see that the longer term predictions appeared to perform better than shorter term ones on the upwards and downwards trends however this was thought to be justified

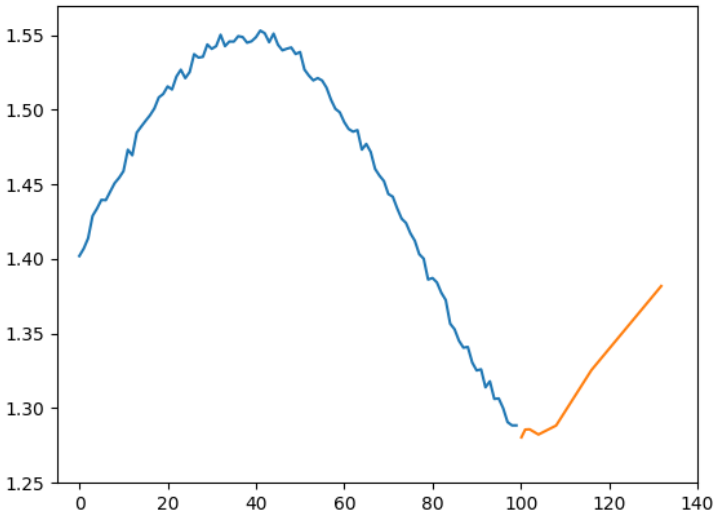

Figure 28: Predictions on a sine wave with noise

given the shorter windows used by the shorter term predictions as well as the nature of the market.

12.3 Parsing data to networks

The dimensions of data returned were calculated in the following way. For the n availiable datapoints the number of paris of inputs/targets m is given by:

$$m = n - timestepToPredict - windowSize + 1 \tag{3}$$

Thus when splitting, there should be splitProp * m sets of training inputs/tagets. The number of inputs considered for a given m number of input/target pairs is given by:

$$inputNo = m + windowSize - 1 \tag{4}$$

For each m set of inputs/targets there is one mean calculated that is used to normalise the inputs i.e number of means = m

The following tests check that data is being correctly formatted to be input to the networks.

testID	Input	Expected Output	Passed
1.1	call process.get(3, 2, "twentyLines.csv"), (windowSize of 3, timestepToPredict 2) splitProp of 0.8	Returned dimensions: 14/6 training/testing inputs, 12/4 training/testing means, 12/4 training/testing targets	Y
1.2	call process.get(3, 2, "twentyLines.csv"), splitProp of 0.8	Returned values: training inputs are first 14 datapoints, test inputs' first windowSize-1 inputs are last windowSize-1 training inputs, ith target is the (windowSize+timestepToPredict)th close price	Y
1.3	call getBatch on the test inputs, means, targets (batchProp 0.5)	Returned dimensions: 2 sets of windows (inputs), 2 targets	Y
1.4	call getBatch on the test inputs, means, targets (batchProp 0.5)	Returned values: All values betweem -1 and 1	Y

13 Site

13.1 Navigation

testID	Test	Expected Outcome	Passed
2.1.1	Load site	Home page shown	Y
2.1.2	Click on "About"	About page shown	Y
2.1.3	Click on "API"	API page shown	Y

13.2 Main Page

testID	Test	Expected Outcome	Passed
2.2.1	Load page	"Current time" shows the current time, "Showing data for" corresponds to the current data being shown. Current data being shown matches the most recent availiable data	Y
2.2.2	Move prediction range slider	Text-box value updates and chart redraws with corresponding number of predictions	Y
2.2.3	Move historic range slider	Text-box value updates and chart redraws with corresponding number of historic data points	Y
2.2.4	Load site after 10pm on Friday	Weekend message shown	Y
2.2.5	Load site after 10pm on Sunday	Weekend message not shown	Y

13.3 API signup

testID	Test	Expected Outcome	Passed
2.3.1	Enter "hello" into email field	Javascript alert seen	Y
2.3.2	Enter "test@gmail.com"	Redirected to API key page and key is shown	Y
2.3.3	Enter "test@gmail.com" again	Page is reloaded with message and API key shown	Y
2.3.4	Click "example query"	sample.json shown	Y
2.3.5	Check database	new user with apikey retrieved earlier shown	Y

14 API

testID	Test	Expected Outcome	Passed
3.1	Use key tied to "test@gmail.com"	Current predictions.json file returned	Y
3.2	Immediately request with the same API key again	invalidGet.json returned	Y
3.3	Request with invalid API key	invalidGet.json returned	Y
3.4	Request with no API key	invalidGet.json returned	Y
3.5	Check database	2 requests from user_id 1, one of them one served, one not. Requests with invalid keys not shown in database	Y
3.6	Make request with API key tied to "test@gmail.com" again (more than 20 seconds after previous)	Request served and request saved in database	Y

15 Updates

The last 4 tests below test the purging of inactive users stats retrieved at midnight. These tests follow from those carried out in the API section (keeping the user with API key associated to "test@gmail.com" along with the requests made by them). For the first three of these a user is first added over a month before the current date. They should make two requests within 20 seconds of each other. This is to test that the purging of inactive users works as expected and that stats retrieved are only from the last day. For the last test, a new user is also added however it is expected that the user should not be removed.

testID	Test	Expected Outcome	Passed
	Normal Operation		
4.1.1	Current time is a "multiple" of 15 minutes	Server tries to update the price data	Y
4.1.2	Timestamp of data received is the same as the data currently stored	Old data is kept, keep checking for updated data	Y
4.1.3	When new data is received	New data is saved to the file	Y
4.1.4	When new data is received	Predictions are made and updated in less than 30 seconds	Y
4.1.5	Check database	New prediction entry has been added	Y
4.1.6	Run flask app	Server checks for new prices immediately when first request is made	Y
	Erroneous Price Data		
4.2.1	Data received has key "Error"	Old data is kept, keep checking for updated data	Y
4.2.2	Data received has key "Note"	Old data is kept, keep checking for updated data	Y
4.2.3	Data received is in .csv format	Old data is kept, keep checking for updated data	Y
4.2.4	Bad connection and so no data retrieved	Old data is kept, keep checking for updated data	Y
	Midnight updates		
4.3.1	At midnight	atMidnight function called	Y
4.3.2	At midnight	Console displays: 1 new user signed up, 1 user removed, 1 remaining user, 1 active user, 3 total requests, 1 rejected request	Y
4.3.3	Check database	Inactive user removed	Y
4.3.4	Call Purge when a user has been inactive for 29 days	User not removed	Y

Part V

Evaluation

16 Comments on Testing

The solution passed all the tests outlined in the test plan. Testing was mainly focused on the ability of the site to update, including when unexpected data is retrieved from AlphaVantage, as well as on the functionality of the API provided by the site. The scope of user interaction with the site itself is very limited and so not many tests had to be done on this to ensure the site worked as expected.

As discussed at length, throughout training there was a concern that networks were converging on solutions which quoted the last close price of the window. This is what motivated the "baseline" that the networks should be compared to. Even though the qualitative prediction tests in 12.2 were very promising, results in 12.1 show that 2 of the 6 networks (predicting the 2 hour and 8 hour prices respectively) produced larger MSE values than the baseline on the test data. This is likely due to the stochastic nature of the market. The price could easily move in any direction and thus predicting the most recent close price as the expected price at a point in the future is often a very good, if not the most effective overall prediction, even though it is not very "useful" for a user to see.

Overall testing of the predictions themselves was thought to be very successful as it was shown the networks are able to predict trends more effectively than was suspected when the networks were being trained. In addition, all networks had a percentage accuracy greater than 50% (one of the two criteria specified in 6.5)

17 Comparison with Specification

The solution met all the specification points outlined *(see 5)*. The way in which they were met is outlined below:

Predictions

1. Networks making predictions up to 8 hours were produced. Even though the predictions for the 8 hour mark did not pass the baseline, the predictions made were thought to still be useful to the end user and so this was included in the solution.
2. The networks took in real-time data from AlphaVantage every 15 minutes.

3. Predictions at worst took 5 seconds to be made and saved to json/the database
4. Percentage accuracies as well as standard deviation error for recent predictions were made.
5. (Desirable) The networks were tested against the baseline and their behaviour was examined before being used in the final implementation. All networks had an "accuracy" above 50% on the test data however not all had a lower MSE than the baseline. These networks were still thought to be giving helpful predictions and so were used.

Webpage

1. Recent prices and predictions were displayed graphically on the site.
2. Percentage accuracies and standard deviations were both displayed on the site graphically.
3. The about page gave the user guidance on how the site should be used. The graphs/sliders and other elements were all named.

API

1. The API returned the same json file that is used by the website to display values graphically.
2. The API "sign-up" page is quick and easy to use. Only an email is needed. New emails
3. User emails are stored by their hash - never stored in plaintext.
4. User emails are hashed client-side so they are not sent in plaintext.
5. Inactive users are purged after a month.
6. All user requests were stored in a database
7. Daily metrics are calculated and outputted to the console at midnight.

18 Comments on Predictions on Real-Time Data

When the trained nets were moved over to the site and predictions were made/assessed, the performance of the networks was always lower than that observed when the networks were tested on the historic test data

There were a few reasons proposed as to why this could have been.

- **AlphaVantage only provides data to 4 d.p (1 pip) -** Given that price changes are very small (on the scale of 1/10 of a pip), it is possible that more precision was needed to be able to make effective predictions. Additionally, the networks were trained on data with 5 d.p.

- **The nature of the market had changed -** The training data used spanned from 2010-2016 - it could be that the behaviour of the EUR/USD market has changed in the three years since, which would result in the network performing worse.
- **The market was more volatile due to recent political uncertainty -** The final solution was finished within a week of the brexit deadline, which could have made price movements more extreme and thus harder to predict.

19 Future Improvements

Assuming that the behaviour of all currency pairs are similar, predictions for different pairs could be added to the site and API without much extra overheard. This would greatly increase the usefulness of the solution to an actual end user, who would likely be trading multiple currency pairs at one time.

19.1 Predictions

Even though the predictions passed the specified criteria, it was felt a lot more could be done to improve on them.

Without changing the network structures of outputs, more experimentation could be done during training. Batch size and learning rate were lowered for the networks making longer predictions out of necessity, however different values for both of these could be used deliberately to try to improve results. In addition, the number of training iterations (or epochs) used could be altered, e.g. using a much larger number of iterations, decreasing the learning-rate or increasing the batch size or using fewer training iterations (known as early stopping), a common technique for RNNs to help prevent overfitting on test data (although this did not appear to be a problem as).

There are many more approaches that could have been taken to predicting prices. The networks took in only the OHLC data, however other data such as the volume traded could also be useful to train the networks on as this could give a sense of the volatility at a given point in time, part of this was due to the limitations of AlphaVantage however, which only provides OHLC data.

Another prediction model that could be tested is one that builds on previous predictions to make future ones - one network is used to predict the data for the next timestep, and then the same network takes in that prediction to give a prediction for the timestep after that. The potential issue with this design is that at the moment the networks take in 4 datapoints whereas the predictions give out one value.

One the other hand the targets/outputs of the network could be redesigned. Targets could be the gradient of a regression line of time on price for the range of timesteps from the most recent to the timestep for which the price is being predicted. The most recent close price could be made the origin to help simplify calculations. The network outputs could then be a prediction of the gradient of this line and from this, a prediction of the price can be found. This method is much more focused on the trend of the price rather than a specific value in a point in time and so it is thought that this would give much more satisfactory results.

19.2 Site

More could be done to the general apprearance of the site. The layout of the main page especially seems rather bare and patchy and generally the design feels outdated. Using 3rd patry style libraries such as Bootstrap could be a good solution for this.

The price graph could definitely be improved on as well. Chart.js allowed displaying of historic data and predictions on one graph which was thought to be better for the user - giving a better feel for the data. However historic OHLC data is displayed as 3 separate line series, which is not common practice for financial data. In the future an OHLC bar chart or "Japanese candlestick" chart should be used instead as this would be a far more familiar graph format for users.

20 Conclusion

Overall, while there were some doubts about the predictions, the project was successful, achieving all mandatory specification points outlined in the analysis. What was perceived to be the low quantitative performance of the networks was justified by the stochastic nature of the market and any concerns regarding this were alleviated when observing the qualitative tests done on trends in 12.2.

Part VI

Appendix

A Test Evidence

A google drive folder where video evidence of the tests can be found here: `https://bit.ly/2WyO9L0`.

A.1 Predictions

The following graphs show real prices vs. the prediction made for that timestep for each network on a sample of the test data. Real prices are in blue, predictions in orange.

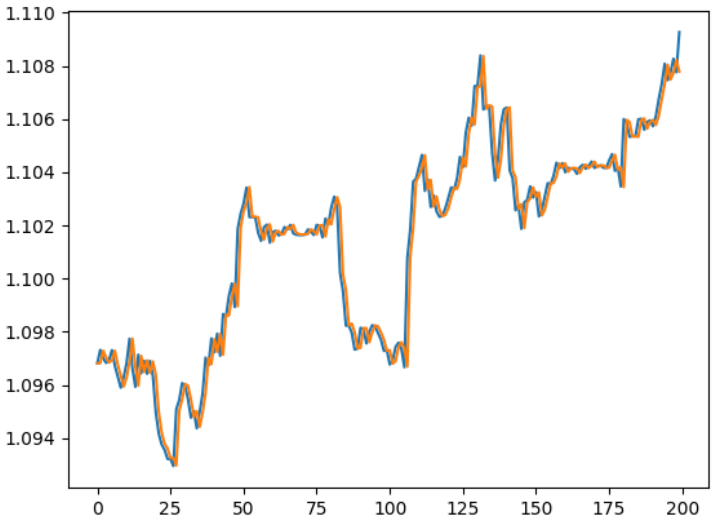

Figure 29: 15 minute predictions

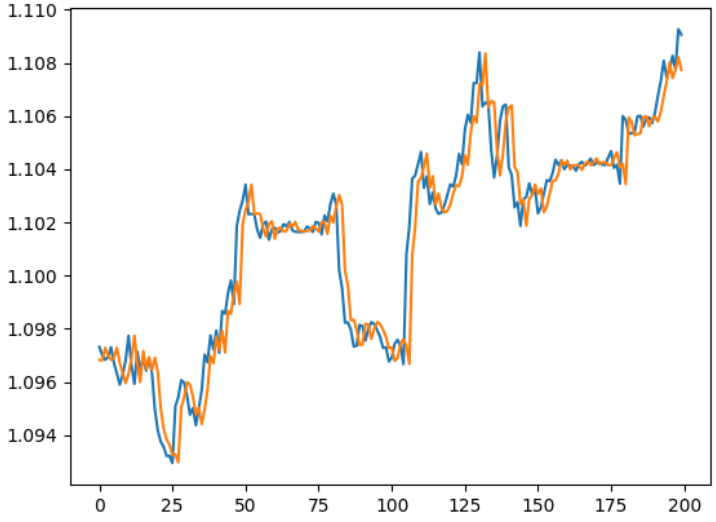

Figure 30: 30 minute predictions

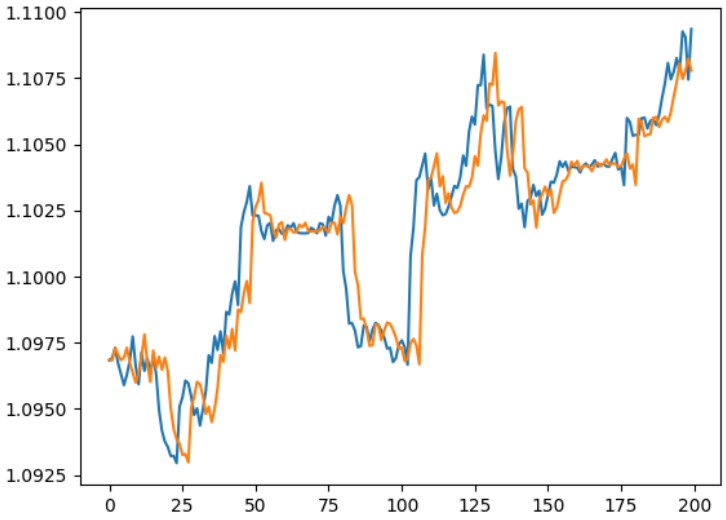

Figure 31: 1 hour predictions

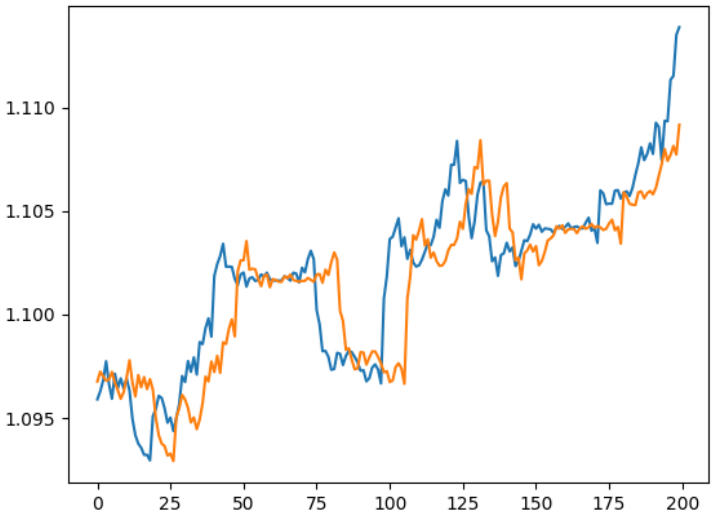

Figure 32: 2 hour predictions

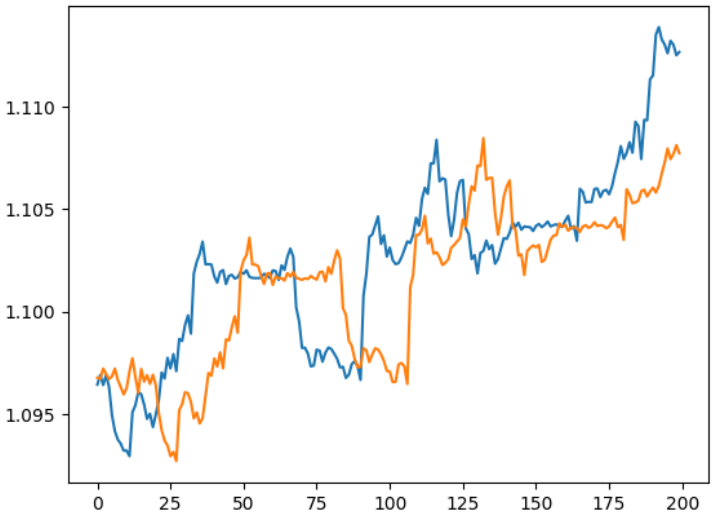

Figure 33: 4 hour predictions

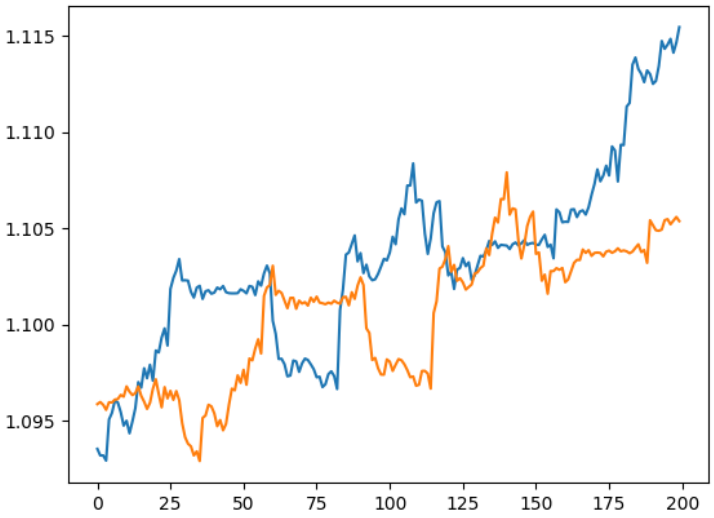

Figure 34: 8 hour predictions

B Source Code

B.1 Networks

B.1.1 Training

The test data used to train the networks could no longer be found on Kaggle. It has not been included below.

lstmPrice.py

```
"""
lstm network that takes in the 4 data points for the current time
↪  step
"""

import torch
from torch import autograd, nn
import torch.nn.functional as F
import torch.optim as optim

```

```
10
11  class model(nn.Module):
12
13      def __init__(self, itemSize, input_no=4, hidden_no=10,
        ↪  lstm_layer_no=1, out_no=1, learning_rate=0.4,
        ↪  useCuda=True):
14          super(model, self).__init__()
15
16          self.useCuda = useCuda
17          self.itemSize = itemSize
18
19          self.hidden_no = hidden_no
20
21          #number of lstm layers
22          self.layer_no = lstm_layer_no
23
24          #lstm layer with input_no in features and hidden_no out
            ↪  features
25          self.lstm = nn.LSTM(input_no, hidden_no, self.layer_no)
26
27          #output layer
28          self.linear = nn.Linear(hidden_no, out_no)
29
30          #lstm hidden states
31          self.hidden = self.init_hidden()
32
33          self.optimizer = optim.Adam(self.parameters(),
            ↪  lr=learning_rate)
34          self.lossFunc = nn.MSELoss()
35
36
37      def init_hidden(self):
38          #init a blank hidden/cell state
39          if self.useCuda:
40              return (torch.zeros(self.layer_no, self.itemSize,
                ↪  self.hidden_no).cuda(),
41                      torch.zeros(self.layer_no, self.itemSize,
                        ↪  self.hidden_no).cuda())
42          else:
43              return (torch.zeros(self.layer_no, self.itemSize,
                ↪  self.hidden_no),
44                      torch.zeros(self.layer_no, self.itemSize,
                        ↪  self.hidden_no))
45
46
47      #pass forword one batch
```

```
48      def forward(self, x):
49          outs = []
50
51          lstm_out, self.hidden = self.lstm(x, self.hidden)
52          net_out = self.linear(lstm_out)
53
54          for i in range(0, len(net_out)):
55              predictions = net_out[i]
56              outs.append(predictions[-1])
57
58          return torch.stack(outs)
```

process.py

```
"""
put data into correct form for network input/target outputs
"""

import torch
from torch import autograd
import random as r


#read in all data from the given file path
def readIn(dataPath):
    toReturn = []

    with open(dataPath, 'r') as file:
        #skip header line
        next(file)
        for line in file:
            ohlc = line.split(',')[1:-1]
            toReturn.append([float(x) for x in ohlc])

    return toReturn


#get the targets for each set of inputs
def getTargets(noPrev, toPredict, data):

    targets = []

    #for each window in the data
    for i in range(0, len(data) - noPrev - toPredict + 1):
        #take the close price
        targets.append([data[i+noPrev+toPredict-1][-1]])

    return autograd.Variable(torch.tensor(targets).float())


#get the mean of each window in the batch
def getMeans(noPrev, toPredict, data):

    means = []

    localTotal = 0

```

```
44      #get the total of the first window
45      for i in range(0, noPrev - 1):
46          localTotal += data[i][-1]
47
48      #for each window in the data
49      for i in range(0, len(data) - noPrev - toPredict + 1):
50          #move the current window across 1
51          if i > 0:
52              #take away the oldest timestep
53              localTotal -= data[i-1][-1]
54          #add the next timestep
55          localTotal += data[i+noPrev-1][-1]
56
57          means.append([localTotal/noPrev])
58
59      return autograd.Variable(torch.tensor(means).float())
60
61
62  #return raw data as torch tensor
63  def getInputs(noPrev, toPredict, data):
64      return
        ↪  autograd.Variable(torch.tensor(data[:len(data)-toPredict]))
65
66
67  #normalise the inputs
68  def prepareInputs(means, inputs, noPrev):
69
70      batchIn = []
71
72      #for each window
73      for i in range(0, len(inputs) - noPrev + 1):
74          window = []
75          for j in range(i, i+noPrev):
76              window.append(inputs[j] - means[i])
77
78          batchIn.append(torch.stack(window))
79
80      return torch.stack(batchIn) * 100
81
82
83  #get a batch
84  def getBatch(means, inputs, targets, noPrev):
85      #take a proportion of the training data
86      batchProp = 0.5
87      length = int(len(means)*batchProp)
88      startIndex = r.randint(0, len(means) - length)
```

```
89
90      splitStart = split(startIndex, means, inputs, targets,
        ↪  noPrev)
91
92      #pass the latter halves of the split into another split
93      splitEnd = split(length, splitStart[1], splitStart[3],
        ↪  splitStart[5], noPrev)
94
95      batchMeans = splitEnd[0]
96      batchIns =  prepareInputs(batchMeans, splitEnd[2], noPrev)
97      batchTargets = (splitEnd[4]-batchMeans)*100
98
99      #return batch of "normalised inputs/targets"
100     return batchIns, batchTargets
101
102
103 #split all data into training/testing batches
104 def splitData(means, inputs, targets, noPrev):
105     #split data into training/testing
106     splitProp = 0.8
107     splitIndex = int(len(means)*splitProp)
108
109     return split(splitIndex, means, inputs, targets, noPrev)
110
111
112 #return all data
113 def get(noPrev, toPredict, dataPath):
114     #get all data
115
116     data = readIn(dataPath)
117
118     inputs = getInputs(noPrev, toPredict, data)
119     targets = getTargets(noPrev, toPredict, data)
120     means = getMeans(noPrev, toPredict, data)
121
122     #move to GPU
123     inputs = inputs.cuda()
124     targets = targets.cuda()
125     means = means.cuda()
126
127     return splitData(means, inputs, targets, noPrev)
128
129
130 #split tensor at index windowNo
131 def split(windowNo, means, inputs, targets, noPrev):
```

```
132     return means[:windowNo], means[windowNo:],
        ↪  inputs[:windowNo+noPrev-1], inputs[windowNo:],
        ↪  targets[:windowNo], targets[windowNo:]
133
134
135 """
136 noPrev = 10
137 toPredict = 1
138 dataPath = "data/overfit.csv"
139
140 means, _, inputs, _, targets, _ = get(noPrev, toPredict,
    ↪  dataPath)
141
142 print(inputs)
143 print(means.view(1, len(means)))
144 print(targets.view(1, len(targets)))
145
146 m1, m2, i1, i2, t1, t2 = split(4, means, inputs, targets, noPrev)
147
148 print(m1)
149 print(m2)
150 print(i1)
151 print(i2)
152 print(t1)
153 print(t2)
154
155 returnI = prepareInputs(means, inputs, noPrev)
156 print(inputs)
157 print(returnI)
158
159 batchMe, batchIn, batchTa = getBatch(means, inputs, targets,
    ↪  noPrev)
160
161 print(batchMe)
162 print(batchIn)
163 print(batchTa)
164 """
```

train.py

```
"""
train network for given number of epochs
"""


import torch
from torch import autograd, nn
import torch.nn.functional as F
import torch.optim as optim

import process


def iterate(net, means, inputs, targets, noPrev, noEpochs):

    net.train()

    for j in range(0, noEpochs):

        #get batch
        batchIn, batchTa = process.getBatch(means, inputs,
        ↪  targets, noPrev)

        #clear the hidden/cell states
        net.hidden = net.init_hidden()

        #clear the accumulated values in the network and
        ↪  optimiser
        net.zero_grad()
        net.optimizer.zero_grad()

        #pass inputs to network
        out = net(batchIn)

        #calculate the loss
        loss = net.lossFunc(out, batchTa)

        #back-propagate
        loss.backward(retain_graph=True)
        net.optimizer.step()


        #display the loss and a sample of the predictions
```

```
42          if j % 1 == 0:
43
44              print("iteration: " + str(j) +  " Loss: " +
                ↪  str(loss.item()))
45              toView = 5
46
47              print("target: " + str(batchTa[:toView].view(1,
                ↪  toView)))
48              print("predi.: " + str(out[:toView].view(1, toView)))
49
50              print()
```

trainLSTM.py

```
"""
train and assess a network for a number of timesteps
"""


import process
import train
import models.lstmPrice
import assess

import torch
from torch import autograd, nn
import torch.nn.functional as F
import torch.optim as optim


windowSize = 90
lstmNo = 30
lstmLayerNo = 1
timestepToPredict = 32

loadOld = True


outPath =
↪  "trainedNets/{}-{}-{}-{}.pth".format(timestepToPredict*15,
↪  windowSize, lstmNo, lstmLayerNo)

#get all testing/training data
trainMe, testMe, trainIn, testIn, trainTa, testTa =
↪  process.get(windowSize, timestepToPredict, "data/OHLC15.csv")

net = models.lstmPrice.model(windowSize, hidden_no=lstmNo,
↪  lstm_layer_no=lstmLayerNo, learning_rate=0.002)
net = net.cuda()

#load an old network in
if loadOld:
    net.load_state_dict(torch.load(outPath))

#assess the network
assess.testNetwork(net, testIn, testTa, testMe, windowSize,
↪  plot=loadOld)
```

```
39
40  #run the training
41  train.iterate(net, trainMe, trainIn, trainTa, windowSize, 50)
42
43  #assess the network
44  assess.testNetwork(net, testIn, testTa, testMe, windowSize,
    ↪  plot=True)
45
46
47  #choose whether to save an old network
48  if loadOld:
49      a = input("save new? ")
50      if a == 'y':
51          #save the model's state
52          torch.save(net.state_dict(), outPath)
53
54  else:
55      #save the model's state
56      torch.save(net.state_dict(), outPath)
57
58  #assess the network
59  assess.testNetwork(net, testIn, testTa, testMe, windowSize,
    ↪  plot=True)
60
```

assess.py

```
"""
asses a trained network's ability on unseen data
"""

import torch
from torch import autograd, nn
import torch.nn.functional as F
import torch.optim as optim

import matplotlib.pyplot as plt
import numpy as np

import process

typicalSpread = 0.00007


#calculate the precentage of predictions within the actual high/low price
def calculatePercentage(prediction, raw):
    correct = 0
    startRaw = len(raw) - len(prediction)

    for i in range(0, len(prediction)-1):
        #if between the high and low
        if (prediction[i] < raw[startRaw + i + 1][1] and
            prediction[i] > raw[startRaw + i + 1][2]):
            correct += 1

    return correct/len(prediction) * 100


#return the close price of each timestep
def getClosePrices(inputs):
    toReturn = []
    for i in range(0, len(inputs)):
        toReturn.append(torch.tensor([inputs[i][-1][3]]))
    return torch.stack(toReturn).cuda()


#main function, assess and graph the predictions
def testNetwork(net, rawIns, targets, means, windowSize, plot=True):
```

```
42
43      inputs = process.prepareInputs(means, rawIns, windowSize)
44
45      net.eval()
46
47      #clear the hidden/cell states
48      net.hidden = net.init_hidden()
49
50      #get the network outputs and the last close price of each
        ↪  batch
51      out = net(inputs)
52      mostRecentClosePrices = getClosePrices(inputs)
53
54
55      print("Predictions:")
56      lossFunc = nn.MSELoss()
57      loss1 = lossFunc(out, (targets-means)*100)
58
59      #percentageError = calculatePercentage(prediction, expected)
60      meanSquaredError = loss1.item()
61
62      print("Correctly predicted within high/low: " +
        ↪  str(calculatePercentage(out/100 + means, rawIns)))
63      print("Mean squared error: " + str(meanSquaredError) + "\n")
64
65
66      print("Most recent close price")
67      loss2 = lossFunc(mostRecentClosePrices, (targets-means)*100)
68      meanSquaredError = loss2.item()
69
70      print("Correctly predicted within high/low: " +
        ↪  str(calculatePercentage(mostRecentClosePrices/100 +
        ↪  means, rawIns)))
71      print("Mean squared error: " + str(meanSquaredError) + "\n")
72
73
74      if plot:
75          plotPredictions(out/100 + means, targets)
76
77
78      #return percentageError, meanSquaredError
79
80
81  #graph a sample of predictions vs the actual prices
82  def plotPredictions(out, targets):
83      toPlot = 200
```

```
84      start = 100
85
86      #change the pytorch tensors to numpy arrays for matplotlib
87      out =
        ↪  out[start:start+toPlot].view(toPlot).detach().cpu().numpy()
88      targets =
        ↪  targets[start:start+toPlot].view(toPlot).detach().cpu().numpy()
89
90      plt.plot(targets)
91      plt.plot(out)
92      plt.show()
```

B.1.2 Testing Networks

The following source code was used as part of the testing

checkProcess.py

```
import process

windowSize = 3
timestepToPredict = 2

#get all data from twentyLines.csv
trainMe, testMe, trainIn, testIn, trainTa, testTa =
↪  process.get(windowSize, timestepToPredict,
↪  "data/twentyLines.csv")

#output each tensor returned
print("Training means: {}".format(len(trainMe)))
print("Testing means: {}\n".format(len(testMe)))

print("Training inputs: {}, {}".format(len(trainIn), trainIn))
print("Testing inputs: {}, {}\n".format(len(testIn), testIn))

print("Training targets: {}, {}".format(len(trainTa), trainTa))
print("Testing targets: {}, {}\n\n".format(len(testTa), testTa))

#use the returned data to retrieve a batch
batchIn, batchTa = process.getBatch(testMe, testIn, testTa,
↪  windowSize)

#output the batch
print("Batch Inputs: {}".format(batchIn))
print("Batch Targets: {}".format(batchTa))
```

This is an example of how the test data for the simple trends were created.

sineNoise.py

```
import numpy as np
import matplotlib.pyplot as plt
import math
import random as r

outpath = "data/sineNoise.csv"

origin = 1.4

timesteps = 200

#use a sine function to get the closeprices for a number of timesteps
close = []
for i in range(0, timesteps+1):
    close.append(origin + 0.15*(math.sin(0.04*i))+ 0.001*r.randint(-5,5))

#get the open, high, low prices from the close price
openP = close[:timesteps]
close = np.asarray(close)[1:]
high = (close + 0.005)
low = (close - 0.005)

#write data to csv
f = open(outpath, 'w')
for i in range(0, timesteps):
    string = "-,{},{},{},{},-\n".format(openP[i], high[i], low[i], close[i])
    f.write(string)

f.close()

#plot the data
plt.plot(high)
plt.plot(low)
plt.plot(close)
plt.show()
```

graphAll.py

```
import process
import train
import models.lstmPrice
import assess

import torch
from torch import autograd, nn
import torch.nn.functional as F
import torch.optim as optim

import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
from matplotlib.collections import LineCollection


#paths to all the trained networks to be used
netPaths = ["trainedNets/15-5-20-1.pth", #checked
            "trainedNets/30-10-20-1.pth", #checked
            "trainedNets/60-20-30-1.pth", #checked
            "trainedNets/120-40-30-1.pth", #checked
            "trainedNets/240-65-30-1.pth", #checked
            "trainedNets/480-90-30-1.pth"] #checked


#take in csv data and return pytorch tensor
def getData(path):

    with open(path) as f:
        data = f.readlines()

    toReturn = []
    for i in range(0, len(data)):
        ohlc = data[i].split(',')[1:-1]
        toReturn.append([float(x) for x in ohlc])

    #get all past data points in reverse chron. order
    toReturn.reverse()
    return torch.tensor(toReturn)


#just get the close price of each timestep in a list for graphing
def getClose(path):
    with open(path) as f:
```

```
45          data = f.readlines()
46
47      toReturn = []
48      for i in range(0, len(data)):
49          ohlc = data[i].split(',')[1:-1]
50          toReturn.append(float(ohlc[-1]))
51
52      return toReturn
53
54
55  #get a window for a network
56  def getSlice(data, steps):
57      #calculate mean close price of region
58      total = 0
59      for i in range(0, steps):
60          total += data[100-steps+i][-1].item()
61      mean = total/steps
62
63      print(torch.stack([data[-steps]]))
64
65      return (torch.stack([data[-steps:]])-mean)*100, mean
66
67
68  #init a network and load the saved parameters
69  def loadNet(path):
70      data = path[:-4].split('-')
71      net = models.lstmPrice.model(int(data[1]),
72                                   hidden_no=int(data[2]),
73                                   lstm_layer_no=int(data[3]),
74                                   useCuda=False)
75
76      net.load_state_dict(torch.load(path))
77      return net, int(data[1])
78
79
80  #make the predictions for each network
81  def predict():
82
83      allData = getData(dataPath)
84      toReturn = []
85
86      #get the prediction for each timestep
87      for i in range(0, len(netPaths)):
88          net, noSteps = loadNet(netPaths[i])
89          data, mean = getSlice(allData, noSteps)
90
```

```
91          net.init_hidden()
92          out = net(data).item()
93          prediction = (net(data).item()/100)+mean
94
95          toReturn.append(prediction)
96
97      return toReturn
98
99
100 #graph a sample of predictions vs the actual prices
101 def plotPredictions(out, targets):
102     prev = 100
103
104     mpl.style.use('default')
105
106     fig, ax = plt.subplots()
107
108     prediction = [[(prev, targets[-100+(100-prev)])]]
109     for i in range(0, len(out)):
110         prediction[0].append((prev + 2**i, out[i]))
111
112     print(prediction)
113     close = np.asarray(targets[100-prev:100])
114     predLine = LineCollection(prediction, colors=['C1'])
115
116     ax.plot(close, "C0")
117     ax.add_collection(predLine)
118
119     plt.xlim((-5, prev+40))
120     plt.ylim((1.3, 1.6))
121
122     plt.show()
123
124
125 dataPath = "data/dscNoise.csv"
126 results = predict()
127 close = getClose(dataPath)
128 plotPredictions(results, close)
```

B.2 Site

B.2.1 Back-End

config.py

```
"""
site configurations
"""

class Config(object):
    SECRET_KEY = 'secret-key'
    SQLALCHEMY_DATABASE_URI = "sqlite:///app.db"
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

database.py

```
"""
initialises the database
helps solve problem with circular imports
"""


from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()
```

models.py

```
"""
all sqlalchemy database models
"""

from app import db

class user(db.Model):
    __tablename__ = "user"

    id = db.Column(db.Integer, primary_key=True)
    emailHash = db.Column(db.String(32), index=True, unique=True)
    apiKey = db.Column(db.String(16), index=True, unique=True)
    dateJoined = db.Column(db.DateTime)

```

```
15      requests = db.relationship("apiRequest",
        ↪  back_populates="user", lazy='select')
16
17      def __repr__(self):
18          return '<User {}>'.format(self.id)
19
20
21  class apiRequest(db.Model):
22      __tablename__ = "request"
23
24      id = db.Column(db.Integer, primary_key=True)
25      dateTime = db.Column(db.DateTime)
26      served = db.Column(db.Boolean)
27
28      user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
29      user = db.relationship("user", back_populates="requests",
        ↪  lazy='select')
30
31      def __repr__(self):
32          return '<Request {} {}>'.format(self.user_id,
            ↪  self.dateTime)
33
34
35  class prediction(db.Model):
36      __tablename__ = "prediction"
37
38      dateTime = db.Column(db.DateTime, primary_key=True)
39      pred15 = db.Column(db.Float)
40      pred30 = db.Column(db.Float)
41      pred60 = db.Column(db.Float)
42      pred120 = db.Column(db.Float)
43      pred240 = db.Column(db.Float)
44      pred480 = db.Column(db.Float)
45
46      def __repr__(self):
47          return '<Prediction {}>'.format(self.dateTime)
```

app.py

```
"""
entry point for flask run
contains about and main page routes
"""


from flask import Flask, render_template, request, flash,
↪  redirect, url_for, Response, send_from_directory
from flask_migrate import Migrate

import datetime, time
import json, requests

from database import db
from config import Config

from api import api
from update import update


app = Flask(__name__)
app.config.from_object(Config)

#init the database
with app.app_context():
    db.init_app(app)

app.register_blueprint(api)
app.register_blueprint(update)

migrate = Migrate(app, db)


import models


jsonPath = 'static/json/'
predictionsFile = 'predictions.json'
alphaVantageDataFile = 'data.json'


#returns hh:mm time from datetime object
def getTime(datetime):
```

```
43      time = str(datetime).split()[1]
44      return time[:5]
45
46
47  #route for about page
48  @app.route("/about")
49  def about():
50      return render_template("about.html")
51
52
53  #route for home page
54  @app.route("/")
55  def home():
56      time = datetime.datetime.utcnow()
57      timeString = getTime(time)
58
59      weekendMessage = ""
60
61      #check if market is closed because of the weekend
62      if time.weekday() == 4 and time.hour > 21 or time.weekday()
        ↪  == 5 or time.weekday() == 6 and time.hour < 22:
63          weekendMessage = "Market is currently closed for the
            ↪  weekend <br> - please check back at 10pm UTC on
            ↪  Sunday"
64
65
66      #get the prices
67      with open(jsonPath + alphaVantageDataFile) as f:
68          data = json.load(f)
69          dataTime = getTime(data["Meta Data"]["4. Last
            ↪  Refreshed"])
70
71          timeLabels = []
72          highPrices = []
73          closePrices = []
74          lowPrices = []
75
76          #load in all prices to memory
77          for key, value in data["Time Series FX (15min)"].items():
78              timeLabels.append(key.split()[1][:-3])
79              highPrices.append(float(value['2. high']))
80              closePrices.append(float(value['4. close']))
81              lowPrices.append(float(value['3. low']))
82
83          timeLabels.reverse()
84
```

```
85          for i in range(1, 33):
86              timeLabels.append('+' + str(i))
87
88          #arrange data in chronological form
89          closePrices.reverse()
90          highPrices.reverse()
91          lowPrices.reverse()
92
93
94      #load in predictions from the api json file
95      with open(jsonPath + predictionsFile) as f:
96          data = json.load(f)
97
98          predictions = [closePrices[-1]]
99          for key, value in data["Predictions"].items():
100             predictions.append(value)
101
102         percentages = []
103         for key, value in data["Meta Data"]["Recent Percentage
            ↪ Correct"].items():
104             percentages.append(value)
105
106         stDevs = [0]
107         for key, value in data["Meta Data"]["Recent Standard
            ↪ Deviation Error"].items():
108             stDevs.append(value)
109
110
111     return render_template("home.html",
112                            weekendMessage=weekendMessage,
113                            timeNow=timeString,
114                            dataTime=dataTime,
115                            timeLabels=timeLabels,
116                            highPrices=highPrices,
117                            closePrices=closePrices,
118                            lowPrices=lowPrices,
119                            percentages=percentages,
120                            stDevs=stDevs,
121                            predictions=predictions)
122
123
124 if __name__ == "__main__":
125     app.debug = False
126     app.run()
```

update.py

```
"""
updates price and prediction data every 15 minutes in the background
"""

import datetime, threading, time
import json, requests
from flask import Blueprint, current_app

import jsonPredictions

import app
from app import db
from models import user, apiRequest, prediction

jsonPath = 'static/json/'
predictionsFile = 'predictions.json'

alphaVantageDataFile = 'data.json'
alphaVantageKey = "2XFPRGYPL0RM2GQ8"
alphaVantageURL = "https://www.alphavantage.co/query?function=FX_INTRADAY&from_symbol=EUR&to_symbol=USD&i

purgeDays = 30


update = Blueprint('update', __name__)


#if the api call doesn't return new data, keep the old values
def returnPrevious():
    print('API Call failed...')
    print('Serving the last valid data file')

    with open(jsonPath + alphaVantageDataFile, 'r') as f:
        data = json.load(f)
        return data


#update data.json, return the time of the data sent
def getPriceData():
    api_url = alphaVantageURL + alphaVantageKey

```

```
42      try:
43          response = requests.get(api_url)
44
45          if response.status_code == 200:
46              data = json.loads(response.content.decode('utf-8'))
47          """
48          with open("mockData/CSV.csv") as f:
49              data = json.loads(f)
50          """
51
52          for key in data.keys():
53              #if error message returned
54              if key == 'Error Message' or key == "Note":
55                  return returnPrevious()
56
57          print("Get successful.")
58          with open(jsonPath + alphaVantageDataFile, 'w') as f:
59              json.dump(data, f, indent=4)
60          return data
61
62      except:
63          return returnPrevious()
64
65
66  #save predictions to database
67  def savePredictions(time, predictions):
68      #write predictions to database
69      print("Saving Predictions")
70
71      with app.app.app_context():
72          #create instance of prediction class
73          pred = prediction(
74              dateTime = datetime.datetime.strptime(time, "%Y-%m-%d
                ↪  %H:%M:%S"),
75              pred15 = predictions[0],
76              pred30 = predictions[1],
77              pred60 = predictions[2],
78              pred120 = predictions[3],
79              pred240 = predictions[4],
80              pred480 = predictions[5],
81          )
82
83          #write to database
84          db.session.add(pred)
85          db.session.commit()
86
```

```
87
88  #return datetime.datetime object
89  def getDateTime(timeString):
90      return datetime.datetime.strptime(timeString, "%Y-%m-%d
        ↪  %H:%M:%S")
91
92
93  #get string from a datetime object
94  def getDateString(dateTime):
95      return datetime.datetime.strftime(dateTime, "%Y-%m-%d
        ↪  %H:%M:%S")
96
97
98  #get the correct prediction to compare with the actual value
99  def retrievePrediction(entry, timestep):
100     if timestep == 1:
101         return entry.pred15
102     elif timestep == 2:
103         return entry.pred30
104     elif timestep == 4:
105         return entry.pred60
106     elif timestep == 8:
107         return entry.pred120
108     elif timestep == 16:
109         return entry.pred240
110     elif timestep == 32:
111         return entry.pred480
112
113
114 #calculate the recent "percentage accuracies" and standard
    ↪  deviation of each prediction
115 def assessPredictions(priceData, time, timestep):
116     timeNow = getDateTime(time)
117     #set the cutoff lookback period to the size of the
        ↪  alphavantage compact period (100 timesteps)
118     prevCutoff = timeNow -
        ↪  datetime.timedelta(minutes=15*(99+timestep))
119     futureCutoff = timeNow -
        ↪  datetime.timedelta(minutes=15*(timestep))
120
121     #retrieve all predictions that could potentially be evaluated
122     with app.app.app_context():
123         predictions = prediction.query.filter(
124                 prediction.dateTime > prevCutoff).filter(
125                 prediction.dateTime < futureCutoff).order_by(
126                 prediction.dateTime.desc()).all()
```

```
127
128         noPredictions = len(predictions)
129
130         #return 0 if would return a div 0 error
131         if noPredictions < 2:
132             return 0, 0
133
134         totalCorrect = 0
135         totalError = 0
136         totalSquaredError = 0
137         for i in range(0, noPredictions):
138             #get the label in the json data with the prediction +
                ↪  offset time
139             predTime = getDateString(predictions[i].dateTime +
                ↪  datetime.timedelta(minutes=15*timestep))
140             ohlc = priceData[predTime]
141             predPrice = retrievePrediction(predictions[i], timestep)
142
143             if predPrice < float(ohlc["2. high"]) and predPrice >
                ↪  float(ohlc["3. low"]):
144                 totalCorrect += 1
145
146             totalError += abs(float(ohlc["4. close"])-predPrice)
147             totalSquaredError += abs(float(ohlc["4.
                ↪  close"])-predPrice)**2
148
149         #return percentage correct and unbiased estimate of stdev
150         #unbiased estimate st. dev. ~ (n/n-1 *
            ↪  ((x^2/n)-(x/n)^2)^(1/2)
151         return ((totalCorrect/noPredictions)*100,
152                 (noPredictions/(noPredictions-1))**0.5 *
                    ↪  ((totalSquaredError/noPredictions)-(totalError/noPredictions)**2)**0.5)
153
154
155 #make predictions for a set of data
156 def getPredictions(time, priceData):
157     #check if predictions have already been made for this time
158     with open(jsonPath + predictionsFile, 'r') as f:
159         data = json.load(f)
160         predictionsTime = data["Meta Data"]["Time"]
161
162     #if not, get new predictions and add a db entry
163     if predictionsTime != time:
164         print('Getting Predictions...')
165
166         percentages = []
```

```
167             stdevs = []
168             for i in range(0, 6):
169                 percentage, stdev = assessPredictions(priceData,
                    ↪  time, 2**i)
170                 percentages.append(percentage)
171                 stdevs.append(stdev)
172 
173             predictions = jsonPredictions.predict(time, percentages,
                ↪  stdevs)
174             savePredictions(time, predictions)
175             print('Done')
176         else:
177             print("Predictions already made.")
178 
179 
180     #remove inactive users and get stats
181     def atMidnight(current):
182         #remove inactive users
183         noRemoved = purge(current)
184         #return stats
185         getStats(current, noRemoved)
186 
187 
188     #remove inactive users
189     def purge(current):
190         #TODO yeet inactive users
191         #look at api rejection of requests to get the right things
192         #for each
193 
194         #get list of all users
195         print("\nRemoving inactive users...")
196         count = 0
197 
198         with app.app.app_context():
199             #get all users
200             allUsers = user.query.all()
201 
202             for i in range(0, len(allUsers)):
203                 User = allUsers[i]
204                 #query request table for last request made by the
                    ↪  user
205                 lastRequest =
                    ↪  apiRequest.query.filter_by(user_id=User.id).order_by(apiRequest.dateTime.de
206 
207                 #if user has made a request check how many days since
                    ↪  the last request
```

```
208             if lastRequest != None and
                ↪  (current-lastRequest.dateTime).days > purgeDays:
209                 db.session.delete(User)
210                 db.session.commit()
211                 count += 1
212
213             #if user has never made a request use signup date
214             else:
215                 if (current-User.dateJoined).days > purgeDays:
216                     db.session.delete(User)
217                     db.session.commit()
218                     count += 1
219
220     return count
221
222
223 #get stats for the last day
224 def getStats(current, noRemoved):
225
226     dayBefore = current - datetime.timedelta(days=1)
227
228     print(dayBefore.date())
229
230     with app.app.app_context():
231         #query tables for stats
232         #number users who have signed up in the last day
233         newSignups =
            ↪  user.query.filter(db.between(user.dateJoined,
            ↪  dayBefore-datetime.timedelta(minutes = 1),
            ↪  dayBefore+datetime.timedelta(minutes = 1))).count()
234         #total number of users remaining in the database
235         remainingUsers = user.query.count()
236         #number of users who have made a request in the last day
237         activeUsers =
            ↪  db.session.query(apiRequest.user_id).join(user).distinct().count()
238         #total number of requests made in the past day
239         totalRequests =
            ↪  apiRequest.query.filter(db.between(apiRequest.dateTime,
            ↪  dayBefore, current)).count()
240         #number of requests that were rejected in the last day
241         rejectedRequests =
            ↪  apiRequest.query.filter(apiRequest.served==0).filter(db.between(apiRequest.date
            ↪  dayBefore, current)).count()
242
243
244     print("\nDAILY SUMMARY: ")
```

```
245         print("Users removed: {}".format(noRemoved))
246         print("New signups: {}".format(newSignups))
247         print("Remaining users: {}".format(remainingUsers))
248         print("Active users: {}".format(activeUsers))
249         print("Total requests: {}".format(totalRequests))
250         print("Rejected requests: {}\n\n".format(rejectedRequests))
251 
252 
253     #get data when app first is run
254     @update.before_app_first_request
255     def activate_job():
256         #get latest prices
257         print('Getting Prices...')
258         firstData = getPriceData()
259         firstTime = firstData["Meta Data"]["4. Last Refreshed"]
260         #predict based off these
261 
262         time1 = datetime.datetime.utcnow()
263         getPredictions(firstTime, firstData["Time Series FX
            ↪  (15min)"])
264         time2 = datetime.datetime.utcnow()
265         print("Predictions made and saved in: {}
            ↪  seconds\n".format((time2-time1).total_seconds()))
266 
267         #run updates in the background
268         def update():
269             data = getPriceData()
270             dataTime = data["Meta Data"]["4. Last Refreshed"]
271 
272             check = False
273 
274             while True:
275                 #get current time
276                 current = datetime.datetime.utcnow()
277 
278                 #if new alphaVantage data expected
279                 if current.minute % 15 == 0:
280                     #check for new data
281                     check = True
282                 #if midnight
283                 if current.hour == 0 and current.minute == 0:
284                     #remove inactive users
285                     atMidnight(current)
286 
287                 if check:
288                     print("\nChecking for new prices...")
```

```
289                 newData = getPriceData()
290                 newTime = newData["Meta Data"]["4. Last
                    ↪  Refreshed"]
291 
292                 #if prices have updated, stop checking
293                 if dataTime != newTime:
294                     print("Prices updated.")
295 
296                     #get predictions and time how long before
                        ↪  they are retrei
297                     time1 = datetime.datetime.utcnow()
298                     getPredictions(newTime, newData["Time Series
                        ↪  FX (15min)"])
299                     time2 = datetime.datetime.utcnow()
300 
301                     print("Predictions made and saved in: {}
                        ↪  seconds\n".format((time2-time1).total_seconds()))
302 
303                     check = False
304                 else:
305                     print("Prices Not updated...\n")
306 
307                 #another delay so the console doesn't get too
                    ↪  congested
308                 time.sleep(30)
309 
310             #delay before checking the time again
311             time.sleep(45)
312 
313     #create and start a new thread for the update function
314     thread = threading.Thread(target=update)
315     thread.start()
```

api.py

```
"""
all routes and functionality for the API
"""


import random, string

from flask import render_template, request, flash, redirect, url_for, Response, send_from_directory, Blueprint

import datetime, time
import json, requests

from app import db
from models import user, apiRequest


jsonPath = 'static/json/'

sampleFile = 'sample.json'
predictionsFile = 'predictions.json'
usageErrorFile = 'invalidGet.json'
errorFile = 'error.json'

api = Blueprint('api', __name__, template_folder='templates')


#generate a random
def createKey():
    toReturn = ''
    #keep generating keys until a unique one is made

    while True:
        toReturn = ''.join(random.choices(string.ascii_letters + string.digits, k=16))
        User = user.query.filter_by(apiKey=toReturn).first()
        if User is None:
            break

    return toReturn


#route for api page
```

```
42  @api.route("/api", methods=["GET", "POST"])
43  def apiHome():
44
45      if request.method == "GET":
46          return render_template("api.html")
47
48      elif request.method == "POST":
49          #if email already in use, reject
50          #if email valid, give key to user and add email/key to
            ↪  database
51
52          #if email entered was invalid, return to api main page
53          if request.form.get("isValid") != '1':
54              return render_template("api.html")
55
56          else:
57              emailHash = request.form.get("email")
58              User =
                ↪  user.query.filter_by(emailHash=emailHash).first()
59
60              #if email was found
61              if User is not None:
62                  flash("This email is already in use")
63                  flash("Your API key is: %s" % User.apiKey)
64                  return render_template("api.html")
65
66              else:
67                  #generate random alphanumeric string
68                  apiKey = createKey()
69                  #get date in string yyyy-mm-dd format
70                  date = datetime.date.today()
71
72                  newUser = user(
73                      emailHash = emailHash,
74                      apiKey = apiKey,
75                      dateJoined = date
76                  )
77
78                  #try to add user entry to database
79                  try:
80                      db.session.add(newUser)
81                      db.session.commit()
82                      return redirect(url_for('api.showKey',
                        ↪  key=apiKey))
83
84                  except:
```

```
85                      flash("There was an error, please try
                        ↪  again...")
86                      return render_template("api.html")
87
88
89  #route for showing api key when valid email is entered
90  @api.route("/show_api_key")
91  def showKey():
92      key = request.args['key']
93      return render_template("showKey.html", key=key)
94
95
96  #route for api requests
97  @api.route("/api/data")
98  def returnData():
99      #if an api key is provided
100     if 'apikey' in request.args:
101         apiKey = request.args['apikey']
102
103         #if the example get request
104         if apiKey == 'testKey':
105             return send_from_directory(jsonPath, sampleFile)
106
107         try:
108             #get the last most recent user/apiRequest datetime
                ↪  where the user's apikey is equal to the one
                ↪  entered
109             User = db.session.query(user,
                ↪  apiRequest.dateTime).outerjoin(apiRequest,
                ↪  user.id == apiRequest.user_id).filter(user.apiKey
                ↪  ==
                ↪  apiKey).order_by(apiRequest.dateTime.desc()).first()
110
111             #if a valid get
112             if User is not None:
113
114                 #setup flag
115                 serveRequest = False
116
117                 timeNow = datetime.datetime.utcnow()
118
119                 #if no datetime was returned
120                 if User[1] == None:
121                     serveRequest = True
122
123                 elif (timeNow-User[1]).total_seconds() > 20:
```

```
124                     serveRequest = True
125
126                 #create a new request entry
127                 Request = apiRequest(
128                     dateTime = timeNow,
129                     served = serveRequest,
130                     user_id = User[0].id
131                 )
132
133                 db.session.add(Request)
134                 db.session.commit()
135
136                 if serveRequest:
137                     return send_from_directory(jsonPath,
                        ↪  predictionsFile)
138
139         except:
140             #if error in querying/writing to database send an
                ↪  error file back
141             return send_from_directory(jsonPath, errorFile)
142
143
144     #if api key not/provided or is invalid, send an invalid
        ↪  response back
145     return send_from_directory(jsonPath, usageErrorFile)
```

jsonPredictions.py

```
import torch

import static.nets.lstmPrice
import json

netPaths = ["static/nets/15-5-20-1.pth", #checked
            "static/nets/30-10-20-1.pth", #checked
            "static/nets/60-20-30-1.pth", #checked
            "static/nets/120-40-30-1.pth", #checked
            "static/nets/240-65-30-1.pth", #checked
            "static/nets/480-90-30-1.pth"] #checked

dataPath = "static/json/data.json"
outPath = "static/json/predictions.json"


def getData(path):
    #get all past data points in reverse chron. order
    with open(path) as f:
        data = json.load(f)["Time Series FX (15min)"]

    toReturn = []
    for _, timestep in data.items():
        ohlc = []
        for _, price in timestep.items():
            ohlc.append(float(price))
        toReturn.append(ohlc)

    toReturn.reverse()
    return torch.tensor(toReturn)


def getSlice(data, steps):
    #calculate mean close price of region
    total = 0
    for i in range(0, steps):
        total += data[100-steps+i][-1].item()
    mean = total/steps

    return (torch.stack([data[-steps:]])-mean)*100, mean


def loadNet(path):
```

```
44      data = path[:-4].split('-')
45      net = static.nets.lstmPrice.model(int(data[1]),
46                                        hidden_no=int(data[2]),
47                                        lstm_layer_no=int(data[3]),
48                                        cuda=False)
49
50      net.load_state_dict(torch.load(path))
51      return net, int(data[1])
52
53
54  def predict(time, percentages, stdevs):
55
56      allData = getData(dataPath)
57
58      jsonData = {}
59      jsonData['Meta Data'] = {}
60      jsonData['Meta Data']['Time'] = time
61      #to add to
62
63      jsonData['Meta Data']['Recent Percentage Correct'] = {}
64      jsonData['Meta Data']['Recent Standard Deviation Error'] = {}
65
66      jsonData['Predictions'] = {}
67
68      toReturn = []
69
70      #get the prediction for each timestep
71      for i in range(0, len(netPaths)):
72          net, noSteps = loadNet(netPaths[i])
73          data, mean = getSlice(allData, noSteps)
74
75          net.init_hidden()
76          out = net(data).item()
77          prediction = (net(data).item()/100)+mean
78
79          toReturn.append(prediction)
80
81          """
82          print(mean)
83          print(out)
84          print(prediction)
85          print()
86          """
87
88          key = '+{}mins'.format(15*(2**i))
```

```
89          jsonData['Meta Data']['Recent Percentage Correct'][key] =
            ↪  percentages[i]
90          jsonData['Meta Data']['Recent Standard Deviation
            ↪  Error'][key] = stdevs[i]
91          jsonData['Predictions'][key] = prediction
92
93      with open(outPath, 'w') as outfile:
94          json.dump(jsonData, outfile, indent=4)
95
96      return toReturn
```

B.2.2 Front-End

site.css

```
/*Title banner at top of scrren*/
.title {
    background-color: DimGray;
    font-size: 1em;
    font-family: "Trebuchet MS", Helvetica, sans-serif;
    color: White;
    padding: 6px 0px 6px 0px;
}

.title h1 {
    text-align: center;
}

/*navbar styling*/
.navbar {
    background-color: LightSlateGray;
    font-size: 1.2em;
    font-family: "Trebuchet MS", Helvetica, sans-serif;
    color: White;
    text-align: center;
    position: sticky;
    top: 0;
    padding: 0px 0px 0px 0px;
}


.navbar a {
    text-decoration: none;
    color: inherit;
    display: inline-block;
    padding: 20px;
}

/*make link for current page bold on navbar*/
.navbar-currentPage {
    font-size: 1.1em;
    font-weight: 550;
}

.row {
    padding: 2% 2% 5% 2%;
```

```
}

/*default columns as 50% of the width*/
.column {
    float: left;
    width: 50%;
    height: 100%;
}

/*40:60 width split*/
.left {
    width: 40%;
}

.right {
    width: 60%;
}

/*clear all after the row class*/
.row:after {
    content: "";
    display: table;
    clear: both;
}

.row h3 {
    padding-top: 3%;
}

.weekendMessage{
    color: red;
}

.chartContainer{
    text-align: center;
    padding: 0% 2% 5% 2%;
}

.bodyContent {
    padding: 5px;
    font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
```

layout.html

```
<!--Base template for all pages-->

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>{% block title %}{% endblock %}</title>
        <link rel="stylesheet" type="text/css" href="{{
        ↪  url_for('static', filename='site.css')}}" />
        <script src="{{ url_for('static',
        ↪  filename='Chart.js')}}"></script>
        <script src="{{ url_for('static',
        ↪  filename='md5.min.js')}}"></script>
    </head>

    <body>
        <div class="title">
            <h1>Forex EUR/USD Price Predictor</h1>
        </div>

        <div class="navbar">
            {% block navs %}
            {% endblock %}
        </div>

        <div class="bodyContent">
            {% block content %}
            {% endblock %}
            <hr/>
            <footer>
        </div>
    </body>
</html>
```

home.html

```
<!--Main page-->

{% extends "layout.html" %}

{% block title %}
Predictions Page
{% endblock %}

{% block navs %}
<a href="{{ url_for('about') }}" class=navbar-item>About</a>
<!--highlight the home heading in the navbar-->
<a href="{{ url_for('home') }}" class=navbar-currentPage>Home</a>
<a href="{{ url_for('api.apiHome') }}" class=navbar-item>API</a>
{% endblock %}

{% block content %}

<!--Display times/prediction percentage accuracies side by
↪  side-->
<div class=row>
    <div class="column left">
        <h1>Predictions Page</h1>
        <div class=times>
            <h3>Current Time: {{timeNow}} (UTC)</h3>
            <h3>Showing Data for: {{dataTime}} (UTC)<h3>
        </div>
        <div class=weekendMessage>
            <h3>{{weekendMessage|safe}}</h3>
        </div>
    </div>

    <div class="column right">
        <div id='barChart'>
            <canvas id="percentageChart" width=100%
            ↪  height=50%></canvas>
        </div>
    </div>
</div>

<div class=chartContainer>
    <div class=row>
            <!--Sliders and values for amount of data shown on
            ↪  price chart-->
```

```
41              <div class=column>
42                  <h4>Previous data shown:</h4>
43                  <input id="historic" type="range" min="10"
                    ↪  max="95" value=35 step="5">
44                  <input type="text" id="historicText" value="35"
                    ↪  disabled>
45              </div>
46              <div class=column>
47                  <h4>Predictions Shown:</h4>
48                  <input id="predictions" type="range" min="2"
                    ↪  max="7" value=5 step="1">
49                  <input type="text" id="predictionText" value="4"
                    ↪  disabled>
50              </div>
51      </div>
52
53      <div id='lineChart'>
54          <canvas id="priceChart" width=100% height=50%></canvas>
55      </div>
56  </div>
57
58  <script>
59
60      var historicSlider = document.getElementById('historic')
61      var predictSlider = document.getElementById('predictions')
62      var historicText = document.getElementById('historicText')
63      var predictionText =
        ↪  document.getElementById('predictionText')
64
65      var noPrev = historicSlider.defaultValue
66      var toPredict = predictSlider.defaultValue
67
68      var labels = {{timeLabels|safe}}
69
70      var timesteps = [0, 1, 2, 4, 8, 16, 32]
71      var predictions = {{predictions|safe}}
72      var stDevs = {{stDevs|safe}}
73
74      var predictionPoints = []
75      var plusStDev = []
76      var minusStDev = []
77
78
79      //put prediction data in x/y form
80      for(var i = 0 ; i < 7 ; i ++) {
81          var xVal = 100 + timesteps[i] - 1
```

```
82          predictionPoints.push({
83                  x: labels[xVal],
84                  y: predictions[i]
85          })
86          plusStDev.push({
87                  x: labels[xVal],
88                  y: predictions[i] + stDevs[i]
89          })
90          minusStDev.push({
91                  x: labels[xVal],
92                  y: predictions[i] - stDevs[i]
93          })
94      }
95
96      //percentage accuracy chart
97      var accuracy =
        ↪  document.getElementById('percentageChart').getContext('2d')
98      var percentageChart = new Chart(accuracy, {
99          type: 'bar',
100         data: {
101             labels: ["15 min", "30 min", "1 hour", "2 hour", "4
                ↪  hour", "8 hour"],
102             datasets: [
103                 {
104                     label: 'Percentage predicted within low/high
                        ↪  price',
105                     backgroundColor: '#40A2ff',
106                     data: {{percentages|safe}},
107                 }
108             ]
109         },
110         options: {
111             responsive: true,
112             title: {
113                 display: true,
114                 text: 'Recent Percentage Accuracy'
115             },
116             ticks: {
117                 min: 0, // minimum value
118                 max: 100 // maximum value
119             }
120         }
121     });
122
123
```

```
124     var price =
        ↪  document.getElementById('priceChart').getContext('2d')
125     var PriceChart = getPriceChart()
126
127     //if historic price slider changed, update values and redraw
        ↪  chart
128     historicSlider.oninput = function() {
129         noPrev = this.value
130         historicText.value = this.value
131         PriceChart = getPriceChart()
132     }
133
134     //if prediction slider changed, update values and redraw
        ↪  chart
135     predictSlider.oninput = function() {
136         toPredict = this.value;
137         predictionText.value = this.value - 1
138         PriceChart = getPriceChart()
139     }
140
141     //define the price chart
142     function getPriceChart()
143     {
144         var PriceChart = new Chart(price, {
145             type: 'line',
146             data: {
147                 labels: {{timeLabels|safe}}.slice(100-noPrev, 100
                    ↪  + timesteps[toPredict-1]),
148                 //historic data is drawn with straight lines,
                    ↪  predictions with bezier curves
149                 datasets: [
150                     {
151                         label: 'Close Price',
152                         data:
                            ↪  {{closePrices|safe}}.slice(100-noPrev,
                            ↪  100),
153                         borderColor: '#3399ff',
154                         fill: false,
155                         tension: 0
156                     },
157                     {
158                         label: 'High Price',
159                         data:
                            ↪  {{highPrices|safe}}.slice(100-noPrev,
                            ↪  100),
160                         borderColor: "#ff9900",
```

```
                        fill: 2,
                        tension: 0
                    },
                    {
                        label: 'Low Price',
                        data:
                        ↪  {{lowPrices|safe}}.slice(100-noPrev,
                        ↪  100),
                        borderColor: "#ff9900",
                        fill: false,
                        tension: 0
                    },
                    {
                        label: 'Predictions',
                        data: predictionPoints.slice(0,
                        ↪  toPredict),
                        borderColor: "#32cd32",
                        fill: false
                    },
                    {
                        label: '+1 St. Dev',
                        data: plusStDev.slice(0, toPredict),
                        borderColor: "#953ecd",
                        fill: 5
                    },
                    {
                        label: '-1 St. Dev',
                        data: minusStDev.slice(0, toPredict),
                        borderColor: "#953ecd",
                        fill: false
                    }
                ]
            },
            options: {
                responsive: true,
                bezier: false,
                title: {
                    display: true,
                    text: 'Historic data and predictions'
                }
            }
        });

        return PriceChart
    }

```

```
204
205 </script>
206
207 {% endblock %}
```

api.html

```
<!--Api main page-->

{% extends "layout.html" %}

{% block title %}
API
{% endblock %}


{% block navs %}
<a href="{{ url_for('about') }}" class=navbar-item>About</a>
<a href="{{ url_for('home') }}" class=navbar-item>Home</a>
<!--highlight the API heading in the navbar-->
<a href="{{ url_for('api.apiHome') }}"
 ↪  class=navbar-currentPage>API</a>
{% endblock %}


{% block content %}

<div class=row>
    <h1>Get an API key</h1>

    <div class="errorMessage">
        <ul>
        {% for message in get_flashed_messages() %}
            <li>{{ message }}</li>
        {% endfor %}
        </ul>
    </div>

    <!--From for entering email-->
    <form name="emailForm" id="emailForm" onsubmit="hashEmail()"
     ↪  action="{{url_for('api.apiHome')}}" method="post">
        Enter your email: <br>
        <input type="text" name="email" placeholder="Enter email
         ↪  here">
        <input type="hidden" name="isValid">
        <input type="submit" value="Get Key">
    </form>

    <br>
```

```
40      <a target='blank' href="/api/data?apikey=testKey" >Example
        ↪  Query</a>
41      <br>
42
43  </div>
44
45  <script type="text/javascript">
46      var salt = 'OkRLWyqj'
47
48      function hashEmail() {
49          //validate email
50
51          var email = document.emailForm.email.value
52          var regex =
            ↪  /^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]
53
54          //do not hash if invalid
55          if (email.match(regex) == null)
56          {
57              alert("The email entered is invalid, please try
                ↪  again")
58              document.emailForm.isValid.value = 0
59          }
60
61          //hash email and send if valid
62          else
63          {
64              document.emailForm.email.value =
                ↪  md5(document.emailForm.email.value+salt)
65              document.emailForm.isValid.value = 1
66          }
67      }
68
69  </script>
70
71
72
73  {% endblock %}
```

showKey.html

```
<!--Page showing user their API key-->

{% extends "layout.html" %}

{% block title %}
    Show Key
{% endblock %}

{% block navs %}
<a href="{{ url_for('about') }}" class=navbar-item>About</a>
<a href="{{ url_for('home') }}" class=navbar-item>Home</a>
<a href="{{ url_for('api.apiHome') }}" class=navbar-item>API</a>
{% endblock %}


{% block content %}
    <div class=row>
        <h3>Your API key is: {{key}} </h3>
        <p>
            This key is linked to your email address.
            <br>Your email address will never be stored or sent
            ↪ in plaintext.
            <br><strong>NOTE:</strong> This key will be invalid
            ↪ if not used for longer than a month.
        </p>
    </div>
{% endblock %}
```

about.html

```
<!--About page-->

{% extends "layout.html" %}

{% block title %}
About
{% endblock %}

{% block navs %}
<!--highlight the about heading in the navbar-->
<a href="{{ url_for('about') }}"
 ↪  class=navbar-currentPage>About</a>
<a href="{{ url_for('home') }}" class=navbar-item>Home</a>
<a href="{{ url_for('api.apiHome') }}" class=navbar-item>API</a>
{% endblock %}

{% block content %}

<div class=row>
    <h1>About</h1>

    <p>
        This site displays intraday predictions for prices of
        ↪  EUR/USD in 15 minute intervals. Predictions were made
        ↪  using LSTM networks created and trained with the
        ↪  Pytorch framework, with networks being fed only
        ↪  open/high/low/close data. Training data used can be
        ↪  found <a target="blank"
        ↪  href="https://www.kaggle.com/rsalaschile/forex-eurusd-dataset">here</a>.
        ↪  Real time data is retrieved from <a target="blank"
        ↪  href="https://www.alphavantage.co/">AlphaVantage's
        ↪  API<a/>. Please note that predictions are not updated
        ↪  over the weekend (10pm Friday to 10pm Sunday UTC
        ↪  time).
    </p>

    <h3>Main Page</h3>
    <p>
```

```
27          The main page displays the prices of up to 95 previous
            ↪ intervals (roughly 24 hours) and predictions up to 8
            ↪ hours. The range sliders above the price graph can be
            ↪ used to change the number of predictions/previous
            ↪ prices shown. Each series on the chart can be
            ↪ hidden/unhidden by clicking on the series names
            ↪ directly on top of the chart.
28      <br><br>
29          Recent performance (from up to the last 24 hours) for
            ↪ each prediction is also shown - both in terms of the
            ↪ percentage of predictions made correctly (predictions
            ↪ made within the actual high/low price of the
            ↪ interval), shown at the top of the page in the bar
            ↪ chart, and in terms of the standard deviation error
            ↪ of a prediction, shown by the purple series on the
            ↪ price chart. Both charts were created with <a
            ↪ target="blank"
            ↪ href="https://www.chartjs.org/">Chart.js</a>.
30      </p>
31
32      <h3>API</h3>
33      <p>
34          Should you wish to get the data numerically instead, you
            ↪ can use the API. Your email will never be stored or
            ↪ sent in plaintext.
35      <br><br>
36          Please note that a request made within 20 seconds of
            ↪ another will not be serviced.
37      </p>
38
39  </div>
40
41  {% endblock %}
```

Bibliography

[1] *50 Forex & Trading Industry Statistics & Trends From 2018* [2018].
URL: *https://brokernotes.co/forex-trading-industry-statistics/*

[2] *Candlestick Chart* [n.d.].
URL: *https://datavizcatalogue.com/methods/candlestick$_c$hart.html*

[3] *Database Normalization: 5th Normal Form and Beyond* [n.d.].
URL: *https://mariadb.com/kb/en/library/database-normalization-5th-normal-form-and-beyond/*

[4] *Forex Trading Articles Archives* [n.d.].
URL: *https://forextraininggroup.com/best- timeframes-trading-forex/*

[5] Hannah, F. [2017], 'Beginner's guide to currency trading'.
URL: *https://www.independent.co.uk/money/beginners-guide-to-currency-trading-a7568956.html*

[6] Hatzakis, S. [2019], '9 best forex brokers 2019'.
URL: *https://www.forexbrokers.com/guides/forex-trading*

[7] Lioudis, N. K. [2018*a*], 'Forex leverage: A double-edged sword'.
URL: *https://www.investopedia.com/articles/forex/07/forex$_l$everage.asp*

[8] Lioudis, N. K. [2018*b*], 'Forex trading: A beginner's guide'.
URL: *https://www.investopedia.com/articles/forex/11/why-trade-forex.asp*

[9] LLC, S. L. C. M. [2016], 'Calcuating forex costs in trading'.
URL: *https://www.youtube.com/watch?v=gt5sQN19VAk*

[10] Momoh, O. [2018], 'Stop order'.
URL: *https://www.investopedia.com/terms/s/stoporder.asp*

[11] *Open-high-low-close Chart* [n.d.].
URL: *https://datavizcatalogue.com/methods/OHLC$_c$hart.html*

[12] *Pound plunges after Leave vote* [2016].
URL: *https://www.bbc.co.uk/news/business-36611512*

[13] Russell, J. [n.d.], 'Learn about short selling currency and how it works in forex market'.
URL: *https://www.thebalance.com/what-it-means-to-go-short-in-investment-terms-1344960*

[14] Staff, I. [2017], '2
URL: *https://www.investopedia.com/terms/t/two-percent-rule.asp*

[15] Staff, I. [2018], 'Ohlc chart'.
URL: *https://www.investopedia.com/terms/o/ohlcchart.asp*

[16] *The cost of trading forex* [n.d.].
URL: *https://learn.tradimo.com/dont-go-broke-protect-your-capital/the-cost-of-trading-forex*

[17] *Training and Test Sets: Splitting Data — Machine Learning Crash Course — Google Developers* [n.d.].
URL: *https://developers.google.com/machine-learning/crash-course/training-and-test-sets/splitting-data*

[18] *WaveNet: A Generative Model for Raw Audio* [n.d.].
URL: *https://deepmind.com/blog/wavenet-generative-model-raw-audio/*