Predicting Short Term Price Fluctuations In Cryptocurrency Markets

BSc Computer Science & Artificial Intelligence Final Year Project

University Of Sussex

Candidate: 150645

Supervisor: Dr. Martin Berger

April 2019

Declaration

This report is submitted as a requirement for a BSc Computer Science & Artificial Intelligence undergraduate degree at the university of Sussex.
The content, unless explicitly stated otherwise, is the product of my own labour. The report may be freely copied and distributed provided that credit is given.

Signature:

Acknowledgements

I would like to thank my project supervisor, Dr. Martin Berger, for his enthusiasm and guidance throughout the development of the project.

Glossary

bearish Term used in trading referring to a downtrend pg. 37

bears Term used in trading referring to a person who believes the market to be in an downtrend pg. 37

Binance A Cryptocurrency exchange pg. 10

bullish Term used in trading referring to a uptrend pg. 37

bulls Term used in trading referring to a person who believes the market to be in an uptrend pg. 37

DDL Data Definition Language pg. 50

DML Data Manipulation Language pg. 50

DQL Data Query Language pg. 50

FIFO First In First Out pg. 58

Float Floating point number pg. 12

HTML Hypertext Markup Language pg. 48

IO Input Output pg. 21

MySQL Relational database management system software pg. 50

NoSQL Short for not only SQL pg. 50

partial functions A function that can only operate successfully on specific inputs pg. 20

query Make a request pg. 51

RDBMS Relational database management system pg. 50

relational database management systems Database structured using relations between data pg. 50

SQL Short for structured query language pg. 50

time-series-database A database management system that is designed specifically for time series data pg. 50

TSDB Time-series-database pg. 50

URI Uniform Resource Identifier pg. 38

Summary

The project was inspired by a personal interest in Cryptocurrency trading and the belief in some underlying statistical patterns in financial markets: "Panics are scientifically created. The present panic is the first scientific one, worked out as we figure a mathematical equation"[1], regarding the 1921 U.S recession.

The project presents a prototype application that allows users to trade Cryptocurrencies via the Binance exchange and provides access to historical trading data and prediction models which can be trained in real-time to aid decision making while trading.The software functionality can be categorized into three primary components: obtaining & maintaining Cryptocurrency trading data, Generating predictions and the user interface. Additionally we present an evaluation of candlestick recognition as a means of increasing predictive accuracy and also a discussion on the kind of models that are best suited to financial time series data.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Project Aims

The project presents a prototype software application that allows users to trade Cryptocurrencies vie the Binance exchange and allows for the creation of prediction models in real-time, trained with historical trading data that is kept in a database & is kept up to date via automated processes. Users should be able to place orders via the Binance exchange or in a practise environment. Prediction models should be able to give accurate short term predictions which will require a lot of optimization. We assess a commonly used form of technical analysis, candlestick recognition, and its ability to increase prediction accuracy.

1.2 Target Users

The software was developed for both beginners and experienced Cryptocurrency traders and researchers/students who would like to experiment with prediction models for financial data. Cryptocurrency traders who are just entering the space are able to practise their trading skills without capital risks while experienced traders are able to make real trades via the Binance exchange. The software is in it's infancy and is not suitable for any large scale or commercial application. Further development, specifically regarding security, and the acquisition of hardware would make this scale-able and potentially a viable commercial product.

1.3 Haskell

The purpose of this section is to outline the key features of Haskell, it is not intended as a programming tutorial or a depth-first explanation of the language. We provide explanations and examples of aspects of Haskell which we believe, from our own experience, newcomers may struggle to comprehend and overcome. Our goal is to give an introduction to such that a reader with experience in programming understands how it differs to their preferred language,we do not assume experience with functional programming.The chapter roughly splits into two categories: Haskell's type system and monads.

1.3.1 Strong Type

Fundamentally, computer operate exclusively on bytes; a type system is the mechanism that tell the computer what those bytes are. Haskell uses a strong, static type system.
A strong typing system can eliminate type errors from programs; it can be thought of as a strict application of rules; where a function expects an integer as an argument, the rule is that any input to that function must be an integer and this cannot be broken. Contrasting to weaker typed systems in which the compiler may automatically convert a value to an integer should a function require it. There are different variations of strong typed systems, Haskell's version is essentially: no type will ever be automatically converted to another type to satisfy any conditions, types must be converted explicitly before being passed to functions.

To illustrate, below is a function 'addDecimal' that takes an integer as an argument and returns a float, the result of adding 0.05 to the input argument, Written in Haskell and C.

Haskell Code:

```
-- Convert Int to Float somehow
addDecimal :: Int -> Float
addDecimal x = x + 0.05
-- print the result of addDecimal
main = print $ addDecimal 3
```

C Code:

```
#include <stdio.h>
/* Input : Int  Output: Float */
float addDecimal(int x) {

    /* add decimal to input */
    return x + 0.05;
}
int main(void) {
    /* declare y as an integer with a value of 1 */
    int y = 1;
    /* declare y as the result of addDecimal y */
    y = addDecimal (y);
    /* print y */
    printf("The integer is %d\n", y);
    }
}
```

The Haskell code will not compile, the strong typing rejects our code as we have a mismatch between our types: we are trying to add decimals to an integer, which by it's nature does not have any decimal places. Conversely, our C code will compile without a problem, even though we have that same type mismatch. To make our Haskell code compile, we need to change our argument type to Float.

The advantage of Haskells' strong type system is that it eliminates type errors before they can cause any problems, which is of greater importance when dealing with financial data as is the case with this project. The drawback of this type system is that any type conversion can only be performed by copying the type into a new desired type, which involves more computation and memory usage, which is not required by weaker-types languages such as C.

1.3.2 Static Type

Haskell is statically typed: the type of all expressions within the program must be known at compile time. The Haskell compiler detects expressions with type mismatches and will reject the code at compilation. In contrast to dynamically typed languages such as python where types of expressions do not need to be known at compile time and will compile regardless of type mismatches which almost inevitably leads to run-time errors. The differences between static and dynamic type systems are illustrated below with snippets from Haskell and Python.

Python Code:

```
def argType (x):
    return type(x);
print(argType("Foobar"))
print(argType(5))
```

Haskell Code:

```
import Data.Typeable
argType x = print(typeOf $ x )

-- Will result in compilation error as the type isn't known
main = argType $ 7

-- We declare the type and so now the program will compile
main = argType $ 7 :: Float
```

In both snippets we define a function 'argType', which accepts a single parameter outputs the type of the input. Our Python code, being a dynamically typed system, will compile and produce the correct output for both instances of the 'argType' function.

Our Haskell snippet contains two definitions of the 'main' function, the first instance on will fail at compile time with the error: "Ambiguous type variable 'a0' arising from the literal '7'". This error message is the compiler telling us that it does not know what type should be assigned to '7'.The second 'main' function defined shows the importance of prior knowledge of types in Haskell, it will compile and 'argType' will produce the correct output as we have told the compiler that '7' is of type Float. In practise, the static type system forces the programmer to think ahead about the type of any variables being passed around functions, leading to more concise and readable code while also eliminating run-time errors that are common in dynamically typed languages. Essentially, Haskell's static typing system just means that the programmer needs to pass the right types of expressions into functions or the code will not compile. This does not mean that the type of every variable in the program needs to be declared immediately, types can be declared later on in the program as long as somewhere within the program the type of a value is declared, shown overleaf.

Haskell Code:

```
import Data.Typeable
-- We tell the compiler that argType only accepts Floats
argType :: Float -> IO()
argType x = print(typeOf $ x )

-- Will result in compilation error as the type isn't known
main = argType $ 7
```

In this snippet, we have given the function 'argType' a definition: 'argType' expects a single argument that is of type Float and outputs an IO value (see 1.3.7). With this definition the compiler knows that any value that is passed to the 'argType' function should be of type Float. When the compiler encounters '7' with no type definition it would in our previous example, on the previous page, produce a compilation error, but now the type of '7' can be inferred as Float, as it is being passed to 'argType' which the compiler knows only accepts Floats.

1.3.3 Lazy/Non-strict Evaluation

Haskell uses an evaluation strategy in which the evaluation of expressions is withheld until the result of the evaluation is needed, known as lazy evaluation or call-by-need [5] .In contrast to strict/eager evaluation used by the majority of programming languages where expressions are evaluated as soon as they are declared, regardless of when or if they are actually used by the program. Below illustrates how lazy evaluation works in practise using a Haskell snippet.

Haskell Code:

```
-- x is not evaluated when declared, it is stored in memory as a thunk
x = "foobar"
-- fooBar is also not evaluated and stored in memory as a thunk
fooBar = "i promise " ++ x
-- we don't use either of the thunks so they are not evaluated
main = print "No evaluation of x or fooBar"
-- we now request evaluation of fooBar and x
main = print $ fooBar
```

The expressions 'x' and 'fooBar' are not evaluated upon declaration, instead they are 'promises' to produce the values 'Foobar' and 'i promise foobar' when evaluation is requested. Our first definition of the main function does not require either of the two expressions to be evaluated: we do not try to use 'x' or 'foobar' and so they remain promises. In the first definition of the main function we want to print 'fooBar'; making use of the expression in any form is essentially a request made for the evaluation of the expression and so 'fooBar' will be evaluated, the same applies for the expression 'x' as a request for it's evaluation is made internally by 'fooBar'. If this code were instead to be written in a language using strict evaluation the expressions of 'x' and 'fooBar' would be evaluated immediately, and so 'x' = "Foobar" and 'fooBar' = "i promise Foobar" while in Haskell 'x' = $Promise(Foobar)$ and 'fooBar' = $Promise(ipromiseFoobar)$.

While lazy evaluation does have some benefits and can be used to improve performance, in most cases it actually has the opposite effect. There are two primary reasons why: well written programs have very little avoidable evaluation in the first place and there is a lot of "under the hood" processes required to allow for lazy evaluation. Laziness prevents evaluation of expression until required but this also means that these expressions need to be stored in memory, where they they become known as "thunks". Reading and writing thunks to and from memory incurs a computational cost, which is entirely unnecessary if the expression was going to be evaluated anyway.

1.3.4 Type Inference

Haskell can automatically detect the types of all expressions within the code, known as type inference. Types can be explicitly declared but it is not necessary, whether to do so or not is a matter of personal preference. However, it's not always possible for the compiler to infer types.
The value used to illustrates Haskell's static typing mechanism in 1.3.2 was '7' and you may have noticed that we define '7' as a float, where our intuition tells us that it is an integer. This was not an oversight, '7' was chosen as it exemplifies cases where type inference fails. We illustrate this below, again using our 'argType' function.

Haskell Code:

```
import Data.Typeable
argType x = print(typeOf x)
-- 7 is ambiguous, it may refer to a Int or Float etc
main = argType 7
-- a string isn't ambiguous, can be inferred
main = argType "fooBar"
```

Again, we have two definitions of the 'main' function, the first instance will be rejected at compilation while the second compiles and runs correctly. The first definition is rejected due to the rules of the static type system, yet those same rules are considered to be fulfilled on line 4 even though their is no explicit type definition, this is an example of a successful type inference.

Type inference is successful in the second definition of main as their is no ambiguity: a String, regardless of its content, will always be of type String. It fails in the first definition due to the ambiguity around numbers. There is only one variation of a string, but there are multiple variations of numbers. 7 is a number and the compiler knows that, the ambiguity arises from what type of number 7 is, it could be as our intuition tells us an integer, but it could also refer to a decimal number that as of yet has no decimals places. Haskell's strong type system means that the compiler is not prepared to make assumptions of the type of 7 as other languages might, instead it rejects the code due to the ambiguity; the consideration that has to be made when using type inference is that Haskell's system of strong and static typing is designed to eliminate ambiguity and sometimes an expression may be inherently ambiguous, in which case type inference is not an option.

1.3.5 Variables

In Haskell a variable is a name given to a *expression* and not a name given to a *value* as is the case in imperative languages. By definition a variable is: a quantity that may change within a mathematical problem or experiment. Given that a Haskell variable references an expression, the name variable is misleading: a program does not re-define expressions during execution. In imperative languages variables are a name given to a specific location in a systems memory.The value held at a memory location can be modified which then leads to the variable referencing that location holding a different value. As an example, take the three lines of code below:

```
x = 0
x = 1
x = x + 1
```

An imperative language would have no problem compiling and executing this code; the underlying process being assigning the variable x to a specific memory location and then incrementing the value held at that location two times, giving the final value of 2 which x then holds. Haskell would reject the code at compilation; focusing on the first two lines, we attempt to define the variable x - a expression - two times. Secondly, focusing only on the third line, we try to increment x in a way that is typical in imperative languages. The use of x = x +1 / x++ will inevitably appear in imperative programs where variables are used as 'counters' in loop structures, where the counter variable is incremented at each iteration. This is not possible in Haskell due to the difference in how a variable is defined in functional and imperative languages, namely that a functional variable references an expression and an imperative variable references memory locations. Incrementing a variable in Haskell is equivalent to, assuming we have a variable x where x = 0:

$$(x = x + 1) == (0 == 0 + 1)$$

The above is obviously incorrect and is the reason why Haskell does not have loop structures build into the language. The alternative is of course to use recursive functions in place of loops, we illustrate below how a loop can be 'converted' into a recursive function using pattern matching with guards:

Non-Recursive:

```
While x < 10:
    someFunction()
    x = x + 1
```

Recursive:

```
untilTen x | x < 10 = do someFunction()
                         untilTen $ x+1
           | otherwise = somethingElse ()
```

1.3.6 Pure & Impure Functions

Haskell's type system is used to delineate between pure and impure functions, pure functions must fulfill:

- (1) Given the same input, x, the function must produce the same output, y, for each occurrence of $f(x)$
- (2) Evaluation of the function must have no side-effects

Should a function not fulfill these properties it is known as a impure function, or a function that may have side-effects. A side-effect refers to any change in program state, for example changing the value of a variable or reading/writing to storage, or, relating to this project, making Http requests to external servers. A function that retrieves data from an API, given the same input, will not always produce the same result: the server may be down or data may have been deleted etc. Haskell allows the execution of impure functions through monads, see 1.3.7. Examples of pure and impure functions are shown below:

Pure Function:

```
cubeFunc :: Int -> Int
cubeFunc x = x^3
```

cubeFunc is pure because the output depend only on the argument of the function.

Impure Function:

```
cubeFunc :: IO()
cubeFunc = do
            print $ "what number do you want to cube?"
            input <- getLine
            let x = (read input :: Int)
            print $ x^3

main = cubeFunc
```

Here cubeFunc is impure because receiving user input and printing are side-effects.

1.3.7 Monads

Haskell programs are structured using monads, which express structure in terms of values and sequences of computation on those values. Monadic operations follow three rules, known as the monad axioms, which are as follows:

Left Identity:

```
(return a >>= f ==  f a ) Alternatively:  (do {x' <- return x;
                                               f x'} == do {f x})
```

Can be read as: Passing a value to the return function bound to the function is the same as applying the function to the value

Right Identity:

```
(m >>= return == m) Alternatively: (do {x <- m;
                                        return x} == do {m})
```

Can be read as: Binding a monodic value to the return function is the same as the initial monad

Associativity:

```
(m >>= f >>= g == m >>= (\x -> f x >>= g))

Alternatively: (do {y <- do {x <- m;
                             f x }
                    g y } == do {x <- m;
                                 do {y <- f x;
                                     g y
                                     }
                                } == do {x <- m;
                                         y <- f x;
                                         g y
                                         })
```

Can be read as: The sequence of monadic operations has no effect on the value

Haskell doesn't check that the monad laws are followed so it is possible to perform a non-monadic operation in a monad, it is up to the programmer to follow these rules.
In practise, monads can be thought of as objects that hold some value (although this is not actually the case it is a good intuition to work from for beginners), in some cases the value held inside the monad can not be extracted (e.g the IO monad) while others allow only monad specific functions to extract the value (e.g the maybe monad), we provide examples for both overleaf.

The Maybe Monad:

The maybe monad represents computations that can go wrong by not returning a value. They can be used as a safety guard for partial functions. Maybe has two functions that allow the value within the monad to be used: Just and Nothing; Just can be used to take a value from the monad while Nothing indicates that the monad was not given a value. For example below we create a partial function 'numberName' which outputs the name of the integer passed in its argument, but we have only programmed a response for input in the range 1:3, we can use maybe to handle cases where an integer outside of this range is passed in the argument and instead of just throwing an exception on failure, we can return Nothing, otherwise we return the result wrapped in a Just type:

```
numberName :: Int -> Maybe String
numberName x | x == 1 = Just "One"
             | x == 2 = Just "Two"
             | x == 3 = Just "Three"
             | otherwise = Nothing

main = print $ numberName 1

Output: Just One

main = print $ numberName 6

Output: Nothing
```

The IO Monad:

Haskell allows functions to have side-effects (see 1.3.6) through the use of the IO monad. IO refers to any operation that receives input from external sources, i.e receiving user input, or outputting some value, i.e displaying the result of a function. The main function in a Haskell program is always tagged with IO, inside the main function we can use IO values in all cases but outside of main we have to specify that a function is of the IO type. For example, below is a Haskell program that will ask for a users name and then response saying that's a nice name.

```
talk :: String -> String
talk = do
          print $ "Hello, What's your name?"
          response <- getLine
          print $ response ++" That's a nice name"

main = talk
```

The program won't compile because both printing and receiving input from the user are impure functions, they need to be wrapped inside the IO monad to allow execution:

```
talk :: IO()
talk = do
          print $ "Hello, What's your name?"
          response <- getLine
          print $ response ++" That's a nice name"

main = talk
```

This principle applies to any case where the program is to display results or receive input from the outside world, whether the input be directly from a user or from a API. In order to use the values obtained through IO all actions to be performed must be done so within the IO monad, i.e from within the function which has the IO type. Pure functions can still be used so long as they are used within the IO function, for example if we have a pure function 'complimentName':

```
talk :: IO()
talk = do
          print $ "Hello, What's your name?"
          response <- getLine
          print $ complimentName response

complimentName :: String -> String
complimentName x = x ++ ", that's a nice name"

main = talk
```

2 Requirements Analysis

2.1 Functional Requirements

The Application must be able to:

- Implement a client-server architecture
- Implement a web server in Haskell
- Provide a web-based user interface
- Allow users to place orders on the Binance exchange
- Implement a relational database & MySQL server
- Automate the creation and population of the database
- Retrieve & store data in real-time
-
- Recognize precursors of price fluctuations in real-time data
- Generate & increase accuracy of prediction models

2.2 Non-Functional Requirements

- The Application should be built using the MVC (Model-View-Controller) architecture with a messaging layer (a HTTP server) to increase modularity
- The messaging layer should keep the database up to date by making HTTP requests to exchanges and parsing the response before sending the data to the MySQL server
- The user interface should built using HTML,CSS and JavaScript; data displayed in the U.I should be taken exclusively from the messaging layer
- Allow users to view any historical trading data and model predictions on request; trading data should be displayed in financial charts (Candlestick, bar, line etc)
- The messaging layer should be able to extract unique trading pairs and intervals from HTTP responses and use them to create tables in the MySQL database
- The messaging layer should receive HTTP responses as byte-strings and convert those byte-strings to strings to handle the possibility that different cryptocurrency exchanges return data in different formats.
- The accumulated historical trading data will be used to train prediction models and be analyzed to find patterns that may be indicators of short term price movements
- Regression and classification models will be used: classification models should give indications of whether to buy or sell a cryptocurrency and regression models to give estimates of peaks/bottoms of trading ranges
- The application should incorporate prediction models that performed best on historical trading data & the messaging layer should recognize patterns in trading that were prominent before significant fluctuations in historical data
- Pattern recognition & candlestick recognition, being a prominent form of technical analysis, will be used when training models and looking for patterns

3 Background Information

This chapter explains the background to this project, including: time-series data, machine learning, technical analysis, Cryptocurrencies and Http.

3.1 Time Series Data

Time series data is a sequence of data points that measure the same variable over time, you'll be familiar with a weather forecast:

Figure 1: A Weather Forecast Time-Series

A weather forecast is a simple time-series: measuring the peak temperature, low temperature and sky conditions at one day intervals. This particular time series contains 8 elements, the weather forecast for each day in a one week interval including the current day. Each element in the time series is composed of:

Day	Peak temperature	Low Temperature	Sky Condition

And so for Friday the time series element is:

Friday	11	3	cloudy/rain

Financial Time-Series

Financial time-series data differs only from a weather forecast in the variables that are measured. Instead of storing variables regarding the weather on a particular day they store variables relating to the price of a particular asset, a financial time series is composed of:

Date	Open	Close	High	Low	Volume

The open, close, high and low elements refer to the assets price that day, the volume element refers to the number of transactions made. For example, assuming we are looking at a companies shares prices for 01/01/2019. The open element would be the price that the companies shares were being exchanged at midnight the previous day, 31/12/2018. The volume is the total amount of shares that were exchanged throughout the day. The low value would be the lowest price that a share in the company was exchanged at that day and likewise the high is the highest price that a share is the company was exchanged. Lets assume that the companies shares opened at $1.00, a total of 10 shares were exchanged throughout the day: 4 were exchanged at $0.98 while the remaining 6 were exchanged at $1.02. The time-series would then look like:

01/04/2019	$1.00	$1.02	$1.02	$0.98	10

3.2 Machine Learning

This section first explains the machine learning methods used in this project, followed by an explanation of our data sets and how we use these for machine learning.

3.2.1 Supervised Learning

Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs[2]. With supervised learning a model is given a set of training data, typically a vector, and a set of results that correspond to each element of the training data. The machine learning model then learns a function that gives the correct output for the training data set.

3.2.2 Unsupervised Learning

Unsupervised learning is a machine learning task that contrasts with supervised learning in that it does not have labelled training data; unsupervised learning models infer relationships in the data sets and classify that data without the help of the having the correct output in advance.

3.2.3 Regression

Regression analysis is a supervised learning method that tries to find the relationship between the response variable and the input variables. This relationship is some function between the input and output variables. The output variable in a regression model is either discrete or continuous, never categorical: the output is always a number, not a category of any kind. Below we demonstrate how regression models learn this function, using Python, we create a training data set 'Training Predictors' where each element consists of two integers, 'Training Response' contains the actual result for each element of 'Training Predictors'. In this example the relationship between the predictors (input) and response variable (output) is simply the addition function.

```
Training_Predictors = [[1,2],[3,4],[5,6],[6,7],[8,9],[9,10]]
Training_Response = [[3],[7],[11],[13],[17],[19]]
Testing_Predictors = [[10,10],[11,11],[12,5]]

RegressionModel = LinearRegression()
RegressionModel.fit(Training_Predictors,Training_Response)


for i in range(len(Testing_Predictors)):
    print(RegressionModel.Predict([Testing_Predictors[i]]))

Results:  [20.] [22.] [17.]
```

We can see from the results at the bottom of the snippet that the regression model has learned that the relationship between the predictors and response variables is simply the addition of both elements of the predictor variable. This same concept is used with models for financial data, only with a more complex function to be inferred.

Formally, regression models relate Y, a *continuous* or *discrete* value, to a function of X and β:

$$Y \approx f(X, \beta).$$

Where:

- β = The Unknown Parameter
- X = The Independent Variable
- Y = The Output Variable

Given a data set consisting of row vectors of predictor variables, $\{x_1...x_n\}$, and a column vector of response variables, $\{y_1...y_n\}$, with each x_i containing a set of predictor variables $\{p_1...p_n\}$. A linear model assumes that y_i is the result of some linear function of the predictor variables in x_i:

$$y_i = \{\beta_1 x_{i_1}....\beta_p x_{i_p} + \epsilon_i\}$$

Where:

- ϵ_i = A variable that has influence over y_i that is not accounted for by β or x

3.2.4 Classification

Classification is a supervised learning task of predicting what category a observation (Input) belongs to, using the labelled training data as a reference. In contrast with regression, the output variable is always categorical, the input (observations) can be sequences of categorical, discrete or continuous values. Below we demonstrate how a classification model functions, again using Python, we create a training set with corresponding labels, the model is expected to learn to classify negative numbers and positive numbers correctly, using 'Testing Data' to verify whether it has done so.

```
Training_Data = [[-1],[1]]
Training_Labels = ['Negative','Positive']
Testing_Data = [[1],[0],[-4],[-65],[100]]

ClasssificationModel = MLPClassifier()
ClassificationModel.fit(Training_Data,Training_Labels)

for i in range(len(Testing_Data)):
    print(Model.Predict(np.asarray(Testing_Data[i]).reshape(-1,1))[0])

Results: Positive Positive Negative Negative Positive
```

As expected, the model learned that any negative input should be classified as 'Negative' and vice versa. The same concept applies when classifying financial data.

3.3 Machine Learning With Time-series Data

This chapter explains how time-series is used in machine learning tasks, we give examples using the weather forecast from 3.1 and show how regression and classification models are used to make different kinds of predictions for the same data set. First we explain how the data needs to be arranged to make it suitable for both tasks.

Time-series data can be formatted for supervised learning purposes. The sliding window method uses a specified window size, W, to create segmentation of time series data, segmentation's created are a collection of time-series data over W periods of time. For example, using the weather forecast, if we used a window size of 3 then the weather forecast for three consecutive days would become one element in the new data set, the first window would then be:

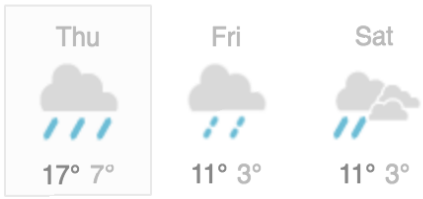

Figure 2: Weather Forecast Window

With the vector representation being:

Thursday	17	7	Cloudy - Rain	Friday	11	3	Cloudy - Rain	Saturday	11	3	Cloudy - Rain

Each window is then used to predict the weather for the day at W+1, with the window above the weather for Sunday would be predicted.

3.3.1 Time-Series data for Regression

Regression analysis predicts continuous or discrete values. In this example using a window size of three we would therefore use a regression model to predict the peak and low temperatures and the sky condition for Sunday, using the vector in 3.3 as the predictors:

Sunday	Predicted High	Predicted Low	Predicted Sky

Using regression analysis on financial time-series is the same as out weather example, the only change is what the predictor and response variables represent. Instead of predicting the peak and low temperature of a given day we predict the peak price and lowest price an asset will be traded at on a given day.

3.3.2 Time-Series Data for Classification

Classification models predict categorical values, they have to be trained with categorical values too: we need to manually add labels to our training data. For the weather forecast example we need to create some appropriate labels for each element in our window: we will label days where it is rainy,cloudy or cold as 'Bad weather', likewise any days which are sunny and hot would be labelled 'good weather'. The training data would then look like:

Day	Peak Temperature	Low Temperature	Sky Condition	Classification Label
Thursday	17	7	Cloudy - Rain	Bad Weather
Friday	11	3	Cloudy - Rain	Bad Weather
Saturday	11	3	Cloudy - Rain	Bad Weather

Bearing in mind that the data has been formatted with a window size of 3, the data set would be a single row vector and not a table as it is shown here. The prediction made from this window would be whether Sunday will be good or bad weather, we don't want to know the specific values. Using classification methods on financial time series works in the same way, instead of classifying predictors as bad or good weather, we classify them as either buy or sell.

3.3.3 Auto-Regressive Models

We make predictions for Cryptocurrency prices using auto-regressive models, defined as $AR(P)$, where p is the order of the model. The order refers to the number of preceding time-series elements used to make predictions - we define this as the window size in 3.3. An auto-regressive model of order p, $AR(p)$ is defined as:

$$AR(p) : X_t = C + \sum_{i=1}^{p} P_i X_{t-i} + e_t$$

- $P_i ... P_p$ are model parameters

C is a constant

e_t is white noise

The model parameters are estimated during training, after which the model can be used to predict an arbitrary number of intervals ahead of the final element of the given time series. We implement an AR model as a collection of individual regression models, with each used to predict a specific element of the time series. We predict four elements of a time series: Volume, High, Low, Close and so we have four regression models that compose the AR model. Each is trained individually on the same data-set using different response variables, using the window method where W is the window size, which represents the number of preceding time series elements used to make predictions - W is also the order of the AR model and so: $AR(P) == AR(W)$.

At prediction time each model makes a prediction for the respective variable, these predictions are then combined to create a time series where each time series is composed of: {Start-Time, Open, High, Low, Close, Volume}. The closing price of the time series at W is used at the open price of the time series at $W + 1$; the start-time is calculated by incrementing the start time of the time series at W by the interval between each element of the time series which is known beforehand and is not involved in the prediction process.

For example, given a time series: T = $\{T_1 ... T_{100}\}$, with equal intervals between each element.
with each element composed of: T(i) = {Start-Time, Low, Open, High, Close, Volume}.
Our AR model is composed of four individual regression models, denoted: M = $\{M_1 ... M_4\}$.
Assuming a window size $W = 5$, therefore AR(5).
At training time each M is trained on $\{T_1 ... T_{95}\}$, leaving $\{T_{95} ... T_{100}\}$ as the first prediction set.
The model can then be defined as:

$$AR(5) : \sum_{x=1}^{M} X_t(m) = C + \sum_{i=1}^{p} P_i X_{t-i} + e_t$$

Where $\sum_{x=1}^{M}$ represents iterating through each individual model
and $X_t(m)$ is the prediction of the model at M_m.
resulting in each X_t consisting of:
{PredictedLow, PredictedHigh, PredictedClose, PredictedVolume}.

3.3.4 Multilayer Perceptrons

A Multilayer perceptron (MLP) is a variant of a feed-forward neural network - a neural network where nodes do not form a cycle: information travels in one direction, see [11]. An MLP has three core layers: input, hidden and output, the hidden layer can contain an arbitrary number of layers each with an arbitrary number of nodes/ neurons. The training process is supervised: training data is accompanied by the actual result for each element, the network tries to learn the correct weights so that it produces the correct output for each training sample, typically generalized to minimizing a loss function. MLP's are suited for this project due to their non-linear activation functions which allow the network to distinguish data that is not linearly separable[10].

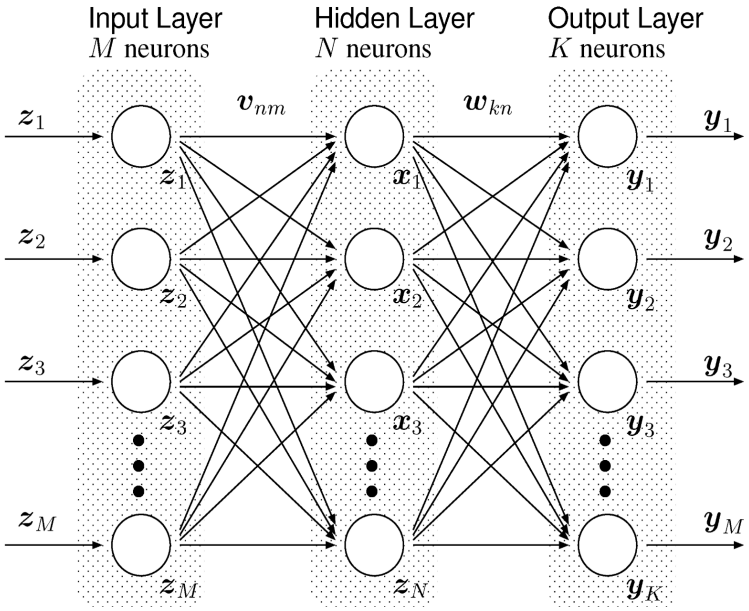

Figure 3: MLP
Source: [12]

The MLP's that we implement use the sigmoid activation function which is non linear. The significance of linear and non-linear functions is discussed in 9, see figures 4 (linear) and 5 (sigmoid) for a visualization of linear and non-linear functions. We provide a discussion with regards to the choice of MLP's and the sigmoid activation function in chapter 11.

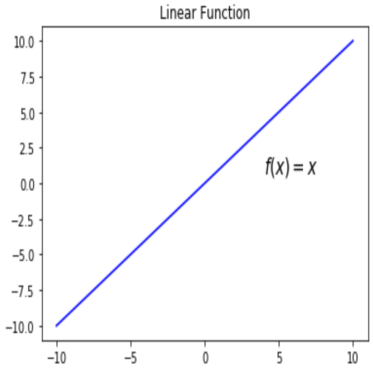

Figure 4: Linear Activation function

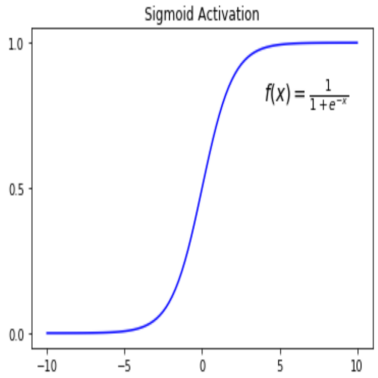

Figure 5: Sigmoid Activation function

3.3.5 MLP Training

MLP's are trained by updating the weights between each layer after each iteration to reduce the output error: For example, assuming we train a MLP with five layers: the input layer, three hidden layers and the output layer we have:

a : The number of nodes in the first layer
b : The number of nodes in the second layer
c : The number of nodes in the third layer
d : The number of nodes in the fourth layer
e : The number of nodes in the fifth layer

With five layers we need four matrices to represent weights between each layer:

W_{ba} : Matrix with b rows and a columns
W_{cb} : Matrix with c rows and b columns
W_{dc} : Matrix with d rows and c columns
W_{ed} : Matrix with e rows and d columns

assuming we have a training set of length x, the first step if to forward propagate:

From layer a - b:

$$S_{bx} = W_{ba} \cdot Z_{ax}$$

which is matrix multiplication and so has time complexity:

$$\mathcal{O}(b \cdot a \cdot x)$$

And then apply the activation function:

$$Z_{bx} = f(S_{bx})$$

The activation function is an element-wise operation, having a time complexity of:

$$\mathcal{O}(b \cdot a)$$

So for forward propagation from layer a to b we have have the total time complexity:

$$\mathcal{O}(b \cdot a \cdot x + b \cdot x)$$

Repeating this for each layer gives the total time complexity for the forward propagation of the network:

$$\mathcal{O}(b \cdot a \cdot x + c \cdot b \cdot x + d \cdot c \cdot x + e \cdot d \cdot x) == \mathcal{O}(x \cdot (ba + cd + dc + ed))$$

For the backward propagation, beginning from the fifth layer to the fourth layer e - d:

$$E_{ex} = f'(S_{ex}) \cdot (Z_{ex} - O_{ex})$$

where E is a matrix of errors for each node in layer h:

The next step is to compute the delta weights:

$$D_{ed} = E_{ex} \cdot Z_{xd}^T$$

where Z_{xd}^T is the transpose of Z_{xd}

Then adjust the weights between the layers:

$$W_{ed} = W_{ed} - D_{ed}^T$$

Giving the time complexity:

$$\mathcal{O}(ex + exd + ed) == \mathcal{O}(e \cdot x \cdot d)$$

Then for the next layer d - c:

$$E_{dx} = f'(S_{dx}) \cdot (W_{de} - E_{ex})$$

$$D_{dc} = E_{dx} \cdot Z_{xc}^T$$

$$W_{dc} = W_{dc} - D_{dc}^T$$

Giving complexity:

$$\mathcal{O}(d \cdot x(e + c))$$

Then for the next layer c - b we have complexity :

$$\mathcal{O}(c \cdot x(d + b))$$

And finally for layer b - a:

$$\mathcal{O}(b \cdot x(c + a))$$

Giving the total complexity:

$$\mathcal{O}(x \cdot (ed + dc + cb + ba))$$

Forwards and backward propagation have the same complexity so for one iteration we have:

$$\mathcal{O}(x \cdot (ed + dc + cb + ba))$$

Multiplied by the number of iterations, the total complexity is:

$$\mathcal{O}(n \cdot x \cdot (ed + dc + cb + ba))$$

3.4 Cryptocurrencies

Cryptocurrencies are a digital currency that run on a blockchain, first introduced by an individual or collective under the name of Satoshi Nakamoto[3]. Cryptocurrencies are decentralized peer-to-peer networks where transactions are confirmed by a consensus protocol and cryptographic algorithms, independent of a central authority. A detailed explanation of blockchain or Cryptocurrencies is beyond the scope of this report, for the purposes of this report we will consider them as assets that can be traded on financial markets.

We obtain historical data for Cryptocurrencies Binance. On exchanges Cryptocurrencies are given trading pair names, composed of three to five letter acronyms of the name of the Cryptocurrencies. Each trading pair available on exchanges is given a name that is the combination of the acronyms of the two assets in the trading pair. e.g Joseph & Bloggs would be abbreviated to JSP/BLG. Each trading pair has historical data recorded at the following intervals:

{ 1 Hours, 2 Hours, 4 Hours, 6 Hours, 12 Hours, 1 Day, 3 Days, 1 Week, 1 Month }

Later in the report we present analysis of prediction models on the Ethereum / United states dollar tether pair at one hour intervals, abbreviated to 'ETH/USDT 1H'. The 'ETH/USDT 1H' data set is a time series of financial data, beginning from the start of trading for the 'ETH/USDT' trading pair on the Binance exchange, 21/08/2017 , and ending at 01/04/2019 giving roughly 15,000 time series elements consisting of:

DateTime	Open Price	Close Price	HighPrice	Low Price	Volume

3.5 Technical Analysis

When trading Cryptocurrencies/stocks either technical analysis (T.A) or fundamentals analysis (F.A) is used to guide decision making. T.A is a trading discipline used to identify trading opportunities by analyzing statistical trends in market data. F.A is a method for evaluating an asset in terms of its underlying intrinsic value, examining economic, financial and other relevant factors. The software implements T.A, essentially the belief that past market data is and indicator of future market movements; T.A employs models and trading rules based on price & volume transformations (technical indicators), intra and inter interval price correlations (i.e intra-day, inter-hourly), recognition of chart patterns and the belief that current market conditions already reflect an assets underlying intrinsic value.

3.5.1 Candlestick Patterns

Market data is often visualized using candlestick charts, candlesticks show an assets opening,closing, high and low price for a given interval. The upper wick shows the highest price of the asset for a given interval and vice verse for the lower wick, the body shows the price movement in the interval, if the asset's price went down during the interval the body is red and if the assets price went up during that interval the body is green. The bottom of a green candlestick body is the opening price of the asset and the top of the body is the price of the asset at the end of that interval, if the body of the candlestick is red then this applies inversely. see figure 6.

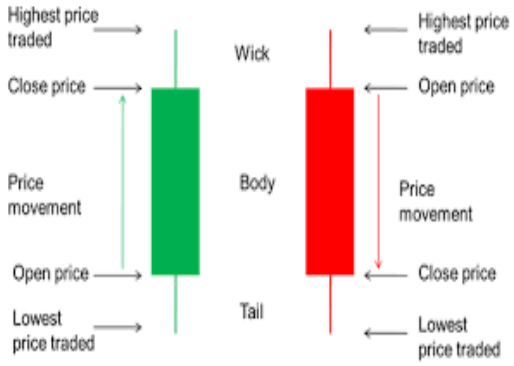

Figure 6: Candlestick components

Candlesticks are favoured by most traders as they are able to provide a visual representation of where buying and selling is occurring in the market, removing any emotional bias from the traders evaluation, and can be used alongside any other form of T.A [4]. Certain candlesticks are believed to be indicators of a change or continuation of the current market trend and can be recognized on a mathematical basis. see figures 7,8.

Figure 7: Candlestick components
Formula: $(((H-L) > 3*(O-C)\&((C-L)/(.001+H-L) > 0.6)\&((O-L)/(.001+H-L) > 0.6)))$

The hammer candlestick is believed to indicate a bullish continuation, with the intuition being that bears have tried to push the price down, indicated by the long lower wick, but bulls rejected the bears attempts, pushing the asset back above its opening price.

Figure 8: Candlestick components
Formula: $(O = C)$

The Doji candlestick is believed to indicate uncertainty, neither bullish or bearish, the intuition being that both the bulls and the bears have tried to push the asset in both directions but have cancelled each-other out.

Where:

- H = Interval High
- L = Interval Low
- O = Interval Open
- C = Interval Close

3.6 HTTP

HTTP - Hypertext Transfer Protocol is used to exchange data across the internet, it is based on the client-server architecture, see figure 9, a client is a machine that requests data from a server, which is another machine with access to a database that processes and responds to client requests. e.g when going to a web-page your computer becomes a client and the machine that is running the website is the server.

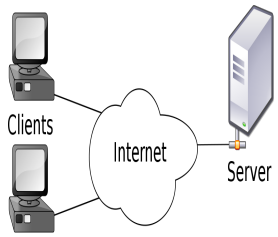

Figure 9: Client Server Architecture

Http has three key features that make it a simple but powerful exchange protocol; it is media independent, meaning that any kind of data can be transferred and it uses the request-response message exchange where the client sends a request, the server processes the request and sends a response, after which the client-server connection is terminated. The request-response message exchange means that Http protocol is stateless: client-server connections are alive only during the request-response exchange and thus Http is connection-less: no client-server connection is maintained. Figure 10 illustrates client sever interaction via Http.

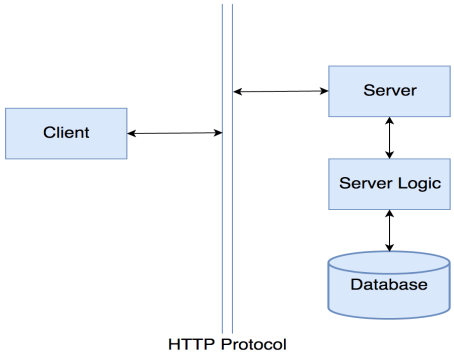

Figure 10: Http Protocol

Http messages are either requests or responses, both have the same general format:

- A Start-Line
- Headers
- Empty Line (Indicates the end of the headers)
- The Message body (Optional.)

The start line differs depending on whether the message is a request or response: requests have a request line and responses have a response line.

Http requests contain a request method which indicates to the server what kind of process needs to be performed. There are seven different request methods but we focus on GET and POST methods as these the most fundamental to understanding Http message exchanges. The get request method is used to retrieve information from the server and a post request is used to send data to the server. Each request also contain a URI which is the target server. Request headers allow additional information to be passed to the server, we won't go into much detail as it's not necessary to understand the basic concept of Http.

After receiving and processing a request a server returns a response message to the client; responses contain a status code which indicates the status of the processing of the request i.e success, failure. Again they may contain headers but that's not important here, the message body of a response contains the data that was requested in the received request message which the client can then extract and do with it what it will.

4 Professional & Ethical Considerations

4.1 Cryptocurrency Regulation and Taxation

The United Kingdom has no legislation in place that specifically relating to Cryptocurrencies; concerns raised by the use of Cryptocurrency include: money laundering, customer protection and taxation. HMRC released these guidelines in their 2014 brief: Revenue and Customs Brief 9 (2014): Bitcoin and other Cryptocurrencies[2]:

4.1.1 Value Added Tax (VAT)

At the time of writing, HMRC applies no VAT to cryptocurrency including: income received from Bitcoin [Cryptocurrency] mining; income received by miners for activity other than mining, including the provision of a service in connection with verification of transactions; exempt under article 135 (1)(d) of the EU VAT Directive. and when Bitcoin [Cryptocurrency] is exchanged for pound Sterling or any other foreign currency. VAT applies only in cases where Cryptocurrency is used as a payment method in an exchange of goods or services.

4.1.2 Corporation Tax

Any profit or loss on exchange movements between currencies is taxable; the same legislation applies to Cryptocurrency exchanges.

4.1.3 Income Tax

Profits and losses of any non-incorporated entity must be reflected of their accounts and are taxable under standard income tax legislation. In the U.K, citizens are afforded a personal allowance of £11,825; that is, per year, income of up to £11,825 tax free. Tax rate then increases incrementally from 20% to 45% after the allowance is exceeded.

4.1.4 Capital Gains Tax

Gains and losses of a non-incorporated entity are chargeable for capital gains tax; for corporations standard Corporate tax legislation applies.

all Cryptocurrency transactions that a user may make will be recorded in the database, available for any relevant authority to inspect should they be inclined to do so.

5 Requirements Analysis

5.1 Functional Requirements

The Application must be able to:

- Serve multiple users in parallel in a client-server architecture
- Be scalable & maintainable
- Implement a server in Haskell
- Provide a web-based user interface
- Allow users to place orders on the Binance exchange
- Implement a relational database & MySQL server
- Automate the creation and population of the database
- Retrieve & store data in real-time
- Recognize precursors of price fluctuations in real-time data
- Generate & increase accuracy of prediction models
- Allow users to generate prediction models in real time from the user interface

5.2 Non-Functional Requirements

- The Application should be built using the MVC (Model-View-Controller) architecture with a messaging layer (a HTTP server) to increase modularity
- The messaging layer should keep the database up to date by making HTTP requests to exchanges and parsing the response before sending the data to the MySQL server
- The user interface should built using HTML,CSS and JavaScript; data displayed in the U.I should be taken exclusively from the messaging layer
- Allow users to view any historical trading data and model predictions on request; trading data should be displayed in financial charts (Candlestick, bar, line etc)
- The messaging layer should be able to extract unique trading pairs and intervals from HTTP responses and use them to create tables in the MySQL database
- The messaging layer should receive HTTP responses as byte-strings and convert those byte-strings to strings to handle the possibility that different cryptocurrency exchanges return data in different formats.
- The accumulated historical trading data will be used to train prediction models and be analyzed to find patterns that may be indicators of short term price movements
- Regression and classification models will be used: classification models should give indications of whether to buy or sell a cryptocurrency and regression models to give estimates of peaks/bottoms of trading ranges
- The application should incorporate prediction models that performed best on historical trading data & the messaging layer should recognize patterns in trading that were prominent before significant fluctuations in historical data
- Pattern recognition & candlestick recognition, being a prominent form of technical analysis, will be used when training models and looking for patterns

6 Software Development Approach

This chapter explains the two dominant approaches to software development: Waterfall and Agile, before presenting the approach taken for this project. A brief summary of the two would be that waterfall stipulates that the software should be developed procedurally and rigorously tested to ensure the correct end result while agile encourages smaller components of the software to be developed in parallel with the focus on achieving the desired functionality and then cleaning up the code.

Waterfall development

The waterfall development approach is the traditional software development approach, treating software development as a linear process where each development phase must be completed before work on the next phase begins. It is typically used where requirements are well documented and fixed, the technology used is well understood and their are no ambiguous requirements. The main advantage of this approach is its simplicity and procedural nature; disadvantages are that it does not allow for changes to software requirements and software is only functional towards the latter stages of the development cycle.Due to these reasons, it is generally not a good fit for on going projects where new technologies are being used and where software requirements are at a high risk of changing during development.

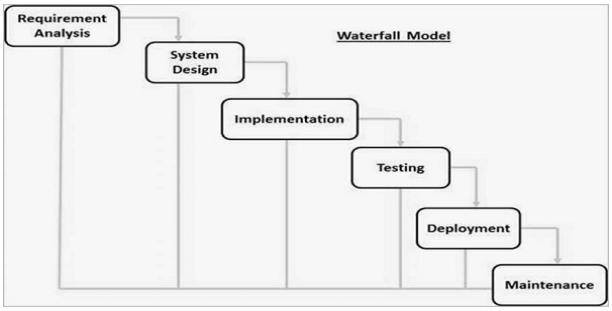

Figure 11: Waterfall Development Cycle

Agile Development

The agile development approach is in many ways the opposite to waterfall: software development is treated as a non-linear process that must accommodate for changes to software requirements. Emphasis is placed on short feedback loops and an iterative approach to building smaller functional components of the software. Agile is beneficial to projects for which not all requirements are known or are at a high risk of changing, for projects which will consistently undergo small changes and for projects which require software components to be developed concurrently.

Figure 12: Agile Development Cycle

Project Development Approach

The project was developed using an agile approach. Waterfall development was not suited to this project for a number of reasons:

- The project required software components to be developed concurrently
- Being a larger project with a concrete deadline, it would be unwise to have to wait until the latter stages of development to integrate all components.
- Software requirements were at a high risk of changing
- Software was developed using new technologies
- Personal preference of the developer to have smaller functional components which can be built upon

The project was planned out in advance, however, in implementation these plans were at best only loosely followed. The project was developed using a new programming language (Haskell) and a new programming paradigm (the functional paradigm). The switch to the functional paradigm meant that development and learning were occurring in parallel, which lead to constant changes to software components as more understanding of the paradigm was obtained.

An example of how the agile approach was utilized in the project is shown in figure 9: when integrating the messaging layer to the MySQL server. The first iteration of the process was to manually create the database and populate it with some test values, the tables names were then hard coded into the messaging layer and used to test the functionality for withdrawing entries from the database. Once that functionality was shown to be correct, the second phase was to use dynamic values to withdraw entries from the database, using values passed in Http requests sent to the messaging layer as SQL query parameters. Once the querying functionality has been proven to be correct, the final phase was to have the database constructed directly from the messaging layer, using values obtained from API requests to exchanges to create and populate the database.

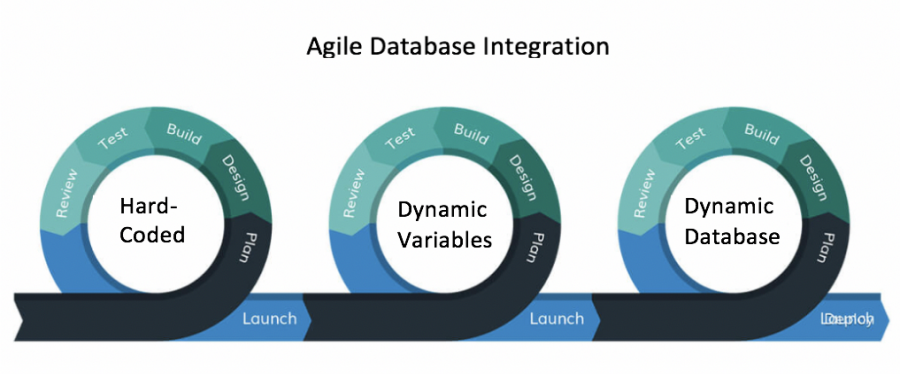

Figure 13: Agile Database Integration

7 Software Architecture

The software was designed using the Model-View-Controller (MVC) architectural pattern with the addition of a messaging layer. MVC splits software into three functionally independent but structurally inter-connected components call the model, view and controller. The model is the central component, containing the application logic, e.g what processes should occur for a given input. The view is the external manifestation of the internal processes of the model, e.g displaying graphs or model predictions. The controller allows external communication with the software, accepting input and relaying that to the view or model. The goal of MVC is to separate the internal functionality of the software from how that functionality is display or interacted with by the user. Each component is treated as a separate object with its own individual responsibilities, allowing the software components to be developed simultaneously and making development and debugging easier due to the reduced code base and set of responsibilities of each component.
This chapter explains each part of the software architecture in terms of its role in the MVC design pattern, it may be helpful to see figure 14 before proceeding.

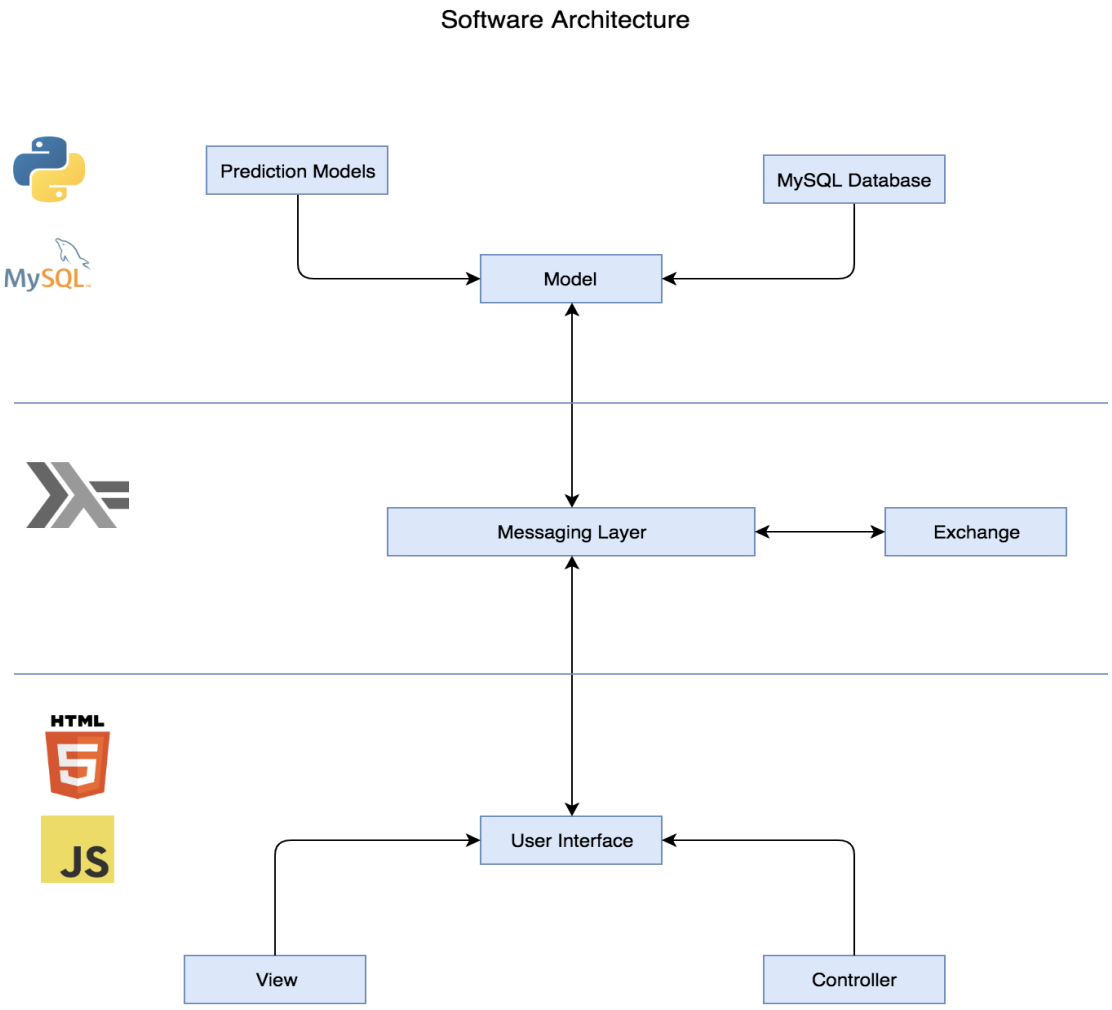

Figure 14: MVC & Messaging Layer

7.1 User Interface

We implement a web based user interface using JavaScript and HTML, the U.I presents real time market data to the users and allows them to create and use prediction models on real time data. The web-page design is attathed to the appendix.

7.2 Messaging Layer

The addition of a messaging layer further enhances the separation of responsibilities, eliminating any remaining functional inter-dependencies of the MVC components which tend to seep through the standard MVC architecture.The role of the messaging layer is essential to be the arbitrator in all interactions between the MVC components. In the standard MVC design, the view is able to request data from the model but with the messaging layer this is no longer the case. Instead the view will request the data from the messaging layer which in turns requests it from the model and then relays the response to the view.

We implement the messaging layer as a Http server, utilizing the client-server architecture and the HTTP exchange protocol explained in 3.6. We made this decision due to the view being web-based, the need to maintain communications with cryptocurrency exchanges and to allow for easy integration of other programming languages, which we use to develop the prediction models. Separation of responsibilities is followed internally through the use of a Haskell stack project, which follows a similar philosophy to to MVC encouraging the code to be separated into modules which are imported to other modules when needed, again allowing simultaneous development of modules and thus simplifying development and debugging.

7.3 Model

This chapter describes the two components of the model; 7.3.1 first outlines the types of database that would be suitable for this project before explaining SQL and our database implementation. 7.3.2 justifies our use of python for prediction modelling and how the prediction models are integrated with the rest of the software, not describing the prediction models themselves.

7.3.1 Database

This chapter explains how we manage our data, we had three options available: SQL databases, NoSQL (Not Only SQL) databases or time-series-database (TSDB). Of course our first choice would be to use a specialized TSDB, however, we are limited to the software available on the university systems and so this was not a valid option, the same applies to NoSQL databases and so we implement a MySQL relational database. See [7] for an explanation of a time series database and [8] for NoSQL databases. We begin with an explanation and demonstration of SQL & then discuss why a SQL RDB is sub-optimal for our data set and how we try to negate some of the resulting performance issues.

SQL - Structured Query Language

SQL allows access to and manipulation of databases, designed specifically for relational database management systems (RDBMS). SQL is composed of various types of statements, each can be considered a sub-language to SQL[6]:

- Data Query Language (DQL)
- Data Definition Language (DDL)
- Data Manipulation Language (DML)

We will demonstrate how SQL is used by creating a simple database which stores the weather for U.K cities, in the example we will use the cities Bradford and Brighton. The intention of the rest of this paragraph is to give an understanding of how a database is created, populated and queried. We choose to build a database that stores the weather as we use a weather forecast to explain time series data in 3.1, which we hope will allow the reader to conceptualize our slightly more complex database using financial time series.

DDL is used to create, alter and remove database objects. eg. to create the database for storing the weather for U.K cities we would first need to create the database and then a database table for cities and weather, the tables are linked with each-other through the foreign key, explained in CHAPTER XYZ:

SQL/DDL Code:

```
CREATE DATABASE UK_Weather;

CREATE TABLE Cities(
    City VARCHAR(50);
    County VARCHAR(50);
   PRIMARY KEY(City,County)
);
CREATE TABLE Weather(
    Peak_Temperature INT(3);
    Low_Temperature INT(3);
    Sky_Condition VARCHAR(20);
```

```
    Date DATE;
    City VARCHAR(50);
    FOREIGN KEY City REFERENCES Cities City;
    PRIMARY KEY(City,Date)
);
```

DML is used to insert, delete and modify data in the database. e.g to insert weather data for Brighton and Bradford we first need to add Bradford and Brighton to the Cities table, then add the weather for a given day to the Weather table, we will add weather for the date 01/01/2019 as an example:

SQL/DML Code:

```
INSERT INTO Cities (City,County)
VALUES ("Bradford" , "West_Yorkshire"),
       ("Brighton" , "East_Sussex");

INSERT INTO Weather (Peak_Temperature,Low_Temperature,Sky_Condition,Date,City)
VALUES (25,14,Clear-Sunny,01/01/2019,Brighton),
       (2,-4,Grey-Miserable,01/01/2019,Bradford)
```

DQL is used to query the database. eg. to extract the weather data for Bradford and Brighton on 01/01/2019 we would write:

SQL/DQL Code:

```
SELECT * FROM (
    SELECT * FROM (
        SELECT * FROM Weather WHERE
                City = "Brighton" OR City = "Bradford") WHERE
                Date BETWEEN 01/01/2018 AND 01/01/2019);
```

The multiple SELECT statements are syntactic, needed due to multiple conditions, it's not necessary to understand this. We now have SQL code that can create a database, insert and extract data; Our software uses a larger but similar database to the one we just created, only storing different values under different table names with more relationships between them. Later in the chapter we refer to 'trading pairs table', 'exchanges table' and 'intervals table'. 'Trading pairs table' refers to a database table that contains the names of all Cryptocurrency trading pairs in much the same way as the 'Cities' table contains the names of cities in out weather database.'Intervals table' refers to a table containing every interval which our Cryptocurrency trading data is recorded at and 'Exchanges table' refers to a table containing all Cryptocurrency exchanges that we have data from. 'Trading pairs table' is related to both the 'Exchanges' and 'Intervals' tables through a foreign key relationship, in the same way that the 'Weather' table is related to the 'Cities' table in the above.

The rest of this chapter is intended to indicate the scale of the data that needs to be managed, explain the issues encountered due to using a RDBMS and not a TSDB or NoSQL database and how we manage those issues.

Time-series data accumulates extremely quickly, we can estimate the number of data entries for a single cryptocurrency using:

$$N \times \left(\sum_{i=1}^{T} X(i) = EntriesPerYear(A(i)) \times \text{ D} \right)$$

- N = Number of trading pairs
- T = Number of time-series intervals
- D = Number of data points per entry
- A = [1m, 3m, 5m, 15m, 30m, 1h, 2h, 4h, 6h, 12h, 1d, 2d, 3d, 1w, 1m]
- EntriesPerYear = The number of the time intervals per year (e.g 525,600 minutes per year)

For reference, Bitcoin (BTC) has 100 trading pairs on the Binance exchange, trading is recorded at 15 different intervals, shown in A above, assuming we store only the basic elements of a time series {Open, High, Low, Close, USD Volume, BTC Volume }, we get:

$$100 \times \left(\sum_{i=1}^{15} X(i) = EntriesPerYear(A(i)) \times 6 \right) = 525,514,359$$

Due to this the majority of developers (68%) prefer to use NoSQL databases when working with time-series data[9], to avoid the scaling issues encountered when using traditional SQL RDB's. However, we were restricted to the use of A RDBMS which is not optimal but it is still suitable. Our data set is highly relational, spanning hundreds of different trading pairs for a single base currency as well as multiple intervals and exchanges, it is in a sense what RDBMS' were designed for, but scale-ability is an issue.

The scale-ability issue refers to problems that arise when handling large data sets; a RDBMS stores data in fixed size pages which are embedded into a data structure, typically a B-tree (see [13]) which is what enables indexing of the data pages. These indices are used to retrieve data with a specified value without having to traverse the whole data set. When the B-tree or other data structure becomes too large to fit into a systems memory it causes a performance bottleneck as data then has to be manipulated through an I/O pipeline between system memory and disk, the bottleneck arising from the relatively slow speed of the disk.

The primary issue encountered regarding scale ability was the speed of SQL queries, which could take minutes when retrieving data from a fully populated table consisting of data from all intervals and exchanges. To alleviate the issue we changed the structure of out database, creating associative tables for each trading pair and interval combination. Associative tables represent many-to-many relationships and maintain the relational properties of the data by containing key references to the individual data tables. In this case, our associative tables maintain relations by having key references to the Exchange,Interval and Trading-Pairs tables and increase the speed of queries by breaking up what would previously have been in a single table into 15 separate tables.

7.3.2 Prediction Models

Due to the lack of machine learning libraries available for Haskell, the prediction models used in the software are developed in python. The python component of the software has a integrated Http listener to receive and respond to requests from the messaging layer, allowing for real-time prediction modelling as stipulated in the software requirements. In addition, with the messaging layer serving as a Http server it allows the software to be used remotely and by multiple users in parallel. Having both the messaging layer and the python predictive modelling components communicate via Http creates a software application than can be distributed across multiple systems.

8 Software Automation

This section explains how we create and populate the database; the typical approach for similar projects would be to first develop a database and then proceed to develop software that utilizes that database. This approach is not suitable for this specific project, it would require us to manually make a list of all trading pairs on each exchange and then, due to our use of associative tables, create 15 database tables for each recorded trading pair. Instead, we developed software that automates this process, the only requirement being that the messaging layer is running and from there the process is initiated with a single HTttp request.

Exchanges do not offer a list of all trading pairs on their API, to obtain one a request is made to the API for the latest price of all assets which returns, along with other things, the latest price of every trading pair on the exchange. We exploit the format in which the data is returned: the name of each trading pair in the Http response message is preceded by the string 'Symbol: ' and so at each occurrence of that string we create a new list element of everything that proceeds it. Then inside each list element, the only sequence of capital letters is the name of the trading pair and so by filtering out all occurrences of lower case characters we have our desired result of a list of all trading pairs, illustrated in figure 15 below. The resulting list is then sent to the MySQL sever and forms the contents of the trading pairs table.

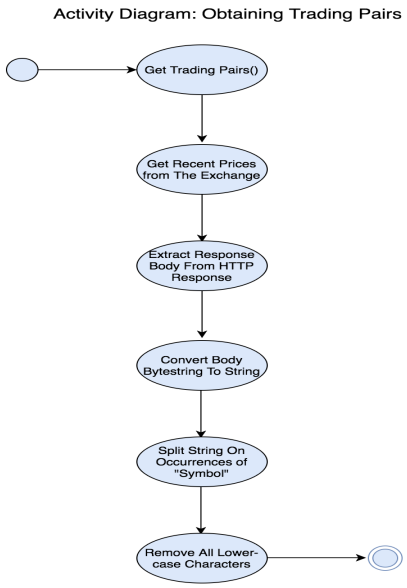

Figure 15: Retrieving Trading Pairs From Exchanges

The intervals used is financial time-series data are standardized and so are embedded in the software as a list, which is then used to populate the intervals table. The software then creates associative tables for each trading pair and interval combination; all elements from the intervals and trading pairs tables are taken and used to complete a SQL query template. The templates provide the required syntax of SQL statements and the semantics are added by concatenating strings, in this case the string to be added to the template query would be the concatenation of the trading pair and the interval. Each completed template is then sent to the MySQL server and executed, resulting in the creation of the associative tables. Figure 16 illustrates the database creating process.

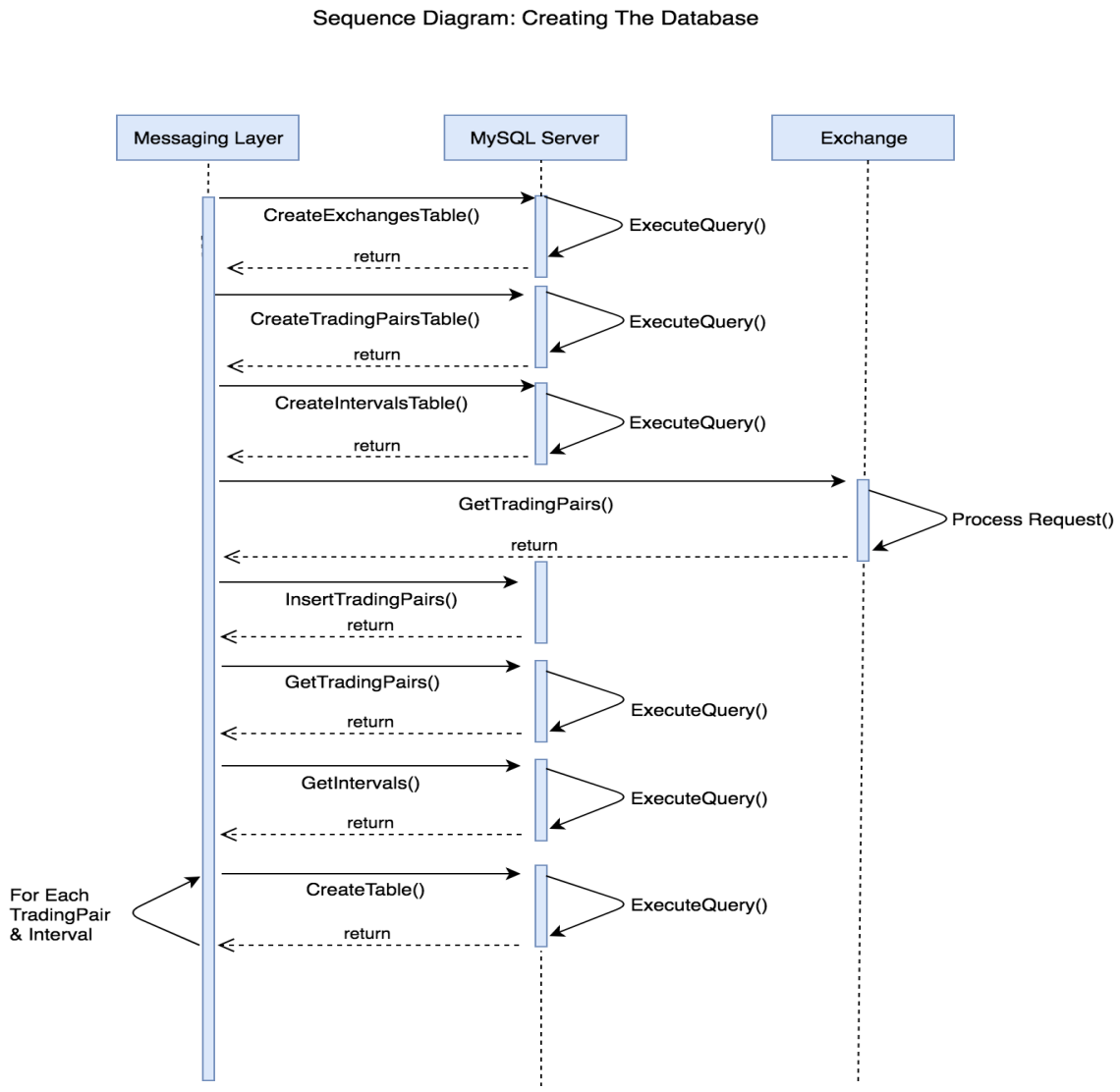

Figure 16: UML Sequence: Creating The Database

The database can then be populated by sending another Http request to the server. The first step of the process is to make a Http request to the exchange to obtain the current server time, which is used as a recursion termination condition, and then extract all trading pairs and intervals from the database. Exchanges limit the number of results returned in requests for trading data to a maximum of 1000 and so it is not a case of making a single Http request, parsing it and moving on to the next trading pair. The values extracted form the trading pairs and intervals table are used to create the Http requests to the exchange and used later to complete SQL query templates to send the API response to the right table. The population control structure is a sequence loops, the outer most iterates the set of trading pairs, the inner loop iterates the intervals set and the inner-most loop makes recursive calls to itself, each iteration making a Http request to the exchange, parsing the response and sending the parsed response to the MySQL server. The Inner-most makes recursive calls to itself until all trading history of a given asset at given intervals is obtained; this is achieved by

extracting the closing time of the last entry of the API response during parsing and using it as the start time of the Http request to the exchange at the next iteration. The termination condition is that the start time is greater than the server time obtained at the start of the population process to ensure that an infinite loop cannot be entered. The population process is illustrated in figure 17 and the API response parsing in figure 18.

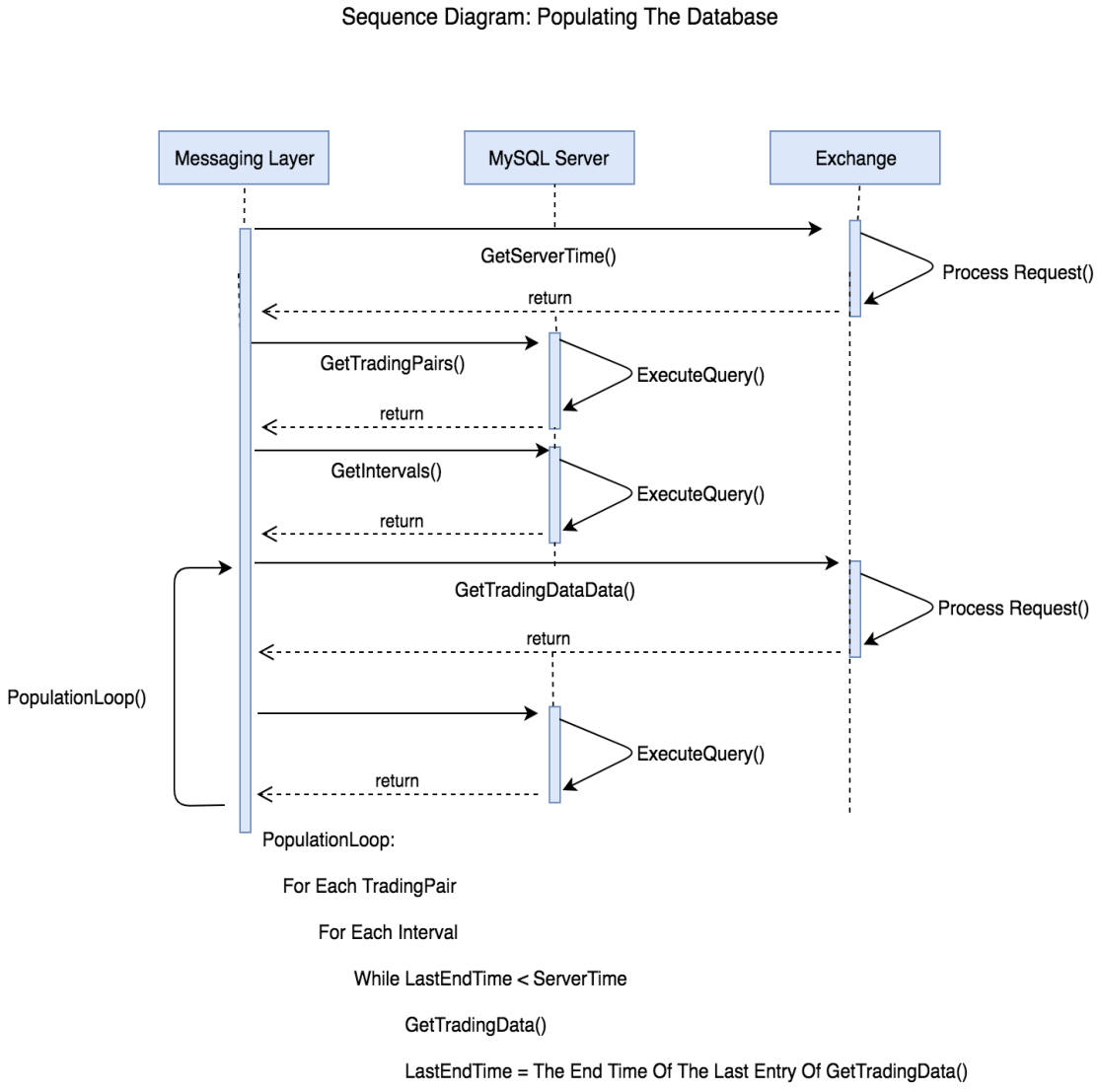

Figure 17: UML Sequence: Populating The Database

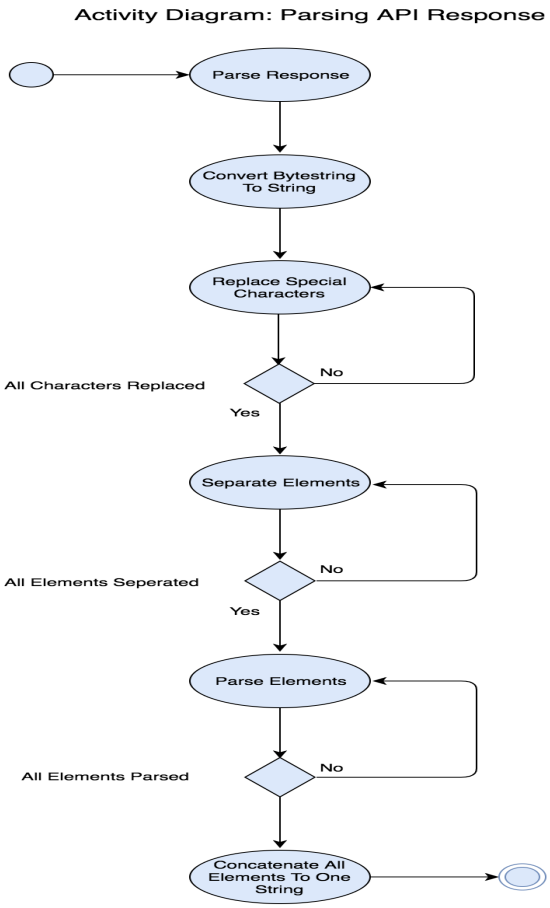

Figure 18: UML Activity: Parse API Response

9 Linear & Non-Linear Predictions

This chapter shows the practical implications of using linear or non-linear models; we begin with a presentation of how a linear model performs followed by a non-linear model on the same data set. We use the ETH/USDT 1H data set throughout this chapter, containing data beginning from 28/01/2017 and ending on 04/04/2019, containing roughly 15,000 entries at 1 hour intervals with the final 50 hours used as a validation set. We use AR models of order 10 , AR(10) (see 3.3.3).

Linear Model

Figure 19 shows the result of predicting 50 hours of trading using a linear model: predictions in red, actual in blue.

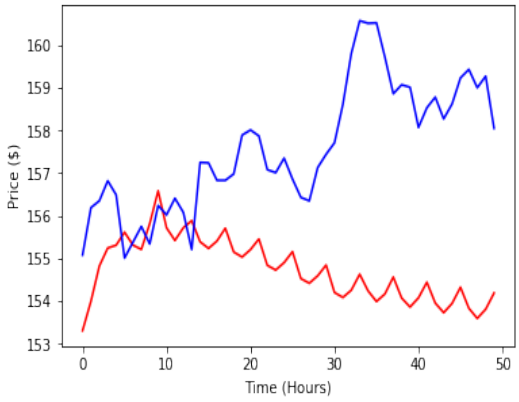

Figure 19: Linear Model Predictions

We observe that the linear model begins to repeat the same pattern after around 10-15 hours ahead of the last known time series. When the model begins predicting, the first window used to predict the next time series is the last 'Window size-1' elements of the training data set; the model stores this as the 'test window' and after each prediction the 'test window' removes the first element and appends the prediction, operating a FIFO structure. After 'Window size-1' predictions have been made then the next 'test window' contains no known data, only predictions made by the model. Here linear models become useless; due to the assumed linear relationship between the predictors and the response, and the data set used to make the predictions now containing no real-world data, the model will repeat the same pattern in its predictions, see figure 20.

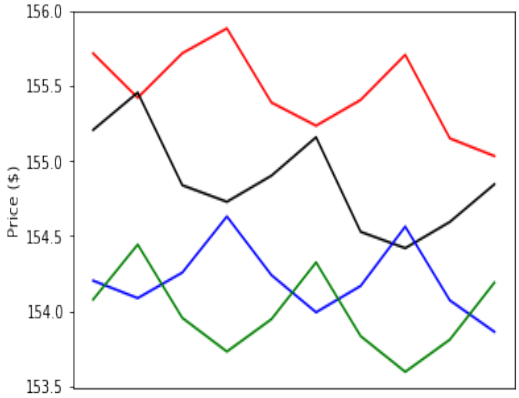

Figure 20: Linear Model Predictions: Patterns

Figure 20 shows the models predictions is 10 hour sequences: The red line shows hours 10:20, Black 20:30, Blue 30:40 and green 40:50. Two distinct patterns can be seen between the sequences: 10:20 hour sequence is almost identical to the 30:40 hour sequence(figure 21), as is the 20:30 sequence to the 40:50 hour sequence(figure 22).

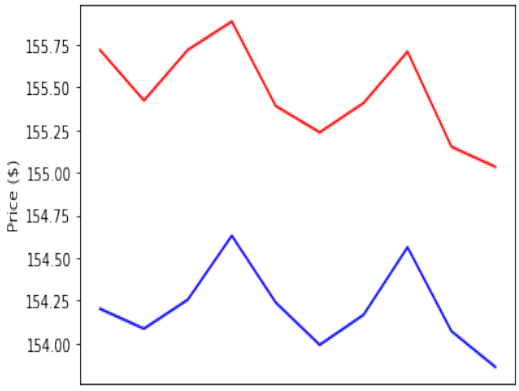

Figure 21: Linear Model Predictions: Correlation 1

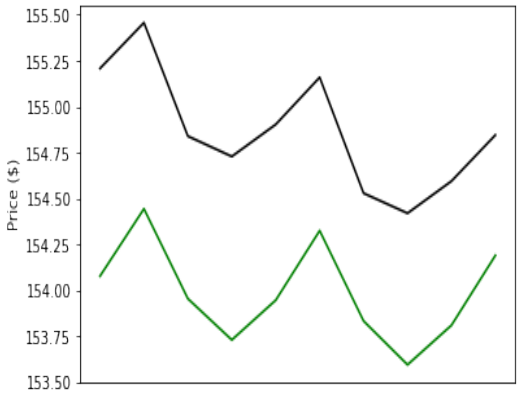

Figure 22: Linear Model Predictions: Correlation 2

Interestingly, both patterns also appear to be the inverse of each other; we cannot explain exactly why this is the case but we can assume that the linear model has concluded that the proceeding 'window size' hours are almost the opposite of the preceding 'window size' hours of trading. Additionally, we can conclude that the reason the model produces these repeated patterns is due to the fact that after 'window size -1' hours of predictions, the next 'window' used to predict the next time series is composed entirely of model predictions. Having learned a linear function in training, any predictions made by the model will fit that linear function and thus any predictions made from those predictions will also match that function, causing it to be repeated indefinitely until it reaches a point where any remaining statistical influence of the real-world data has been eroded and predictions are made that fulfill the learned linear function perfectly, shown in figure 23.

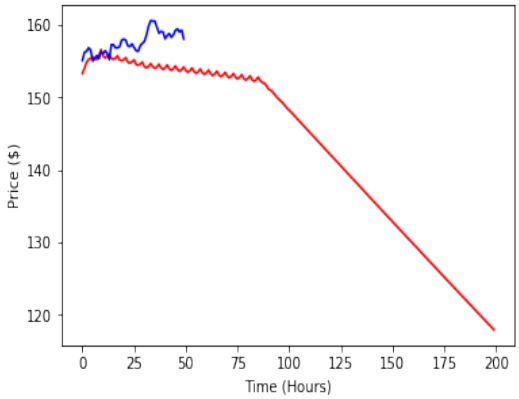

Figure 23: Linear Model Predictions

Obviously, no linear model can be expected to produce accurate predictions an arbitrary period of time ahead in any financial market, which are obviously non-linear and the reason for implementing multi-layer perceptrons. Linear models are not completely inadequate however, the window size is the key to prediction accuracy, below we show the results of making various n-ahead prediction with varying window sizes, again using the ETH/USDT 1h data set.

Window Size	n-ahead Predictions	MSE			
		Low	Close	High	Volume
W = 5	5	0.56	0.44	0.59	176076459
	10	1.07	0.77	0.47	156194241
	15	2.86	2.44	9.55	163548905
	20	6.94	7.68	12.54	164678108
	25	11.06	12.27	16.42	175227822
W = 10	10	0.70	3.03	6.68	436822544
	20	0.92	2.11	6.62	418417947
	30	1.53	2.08	5.43	381711221
	40	4.88	5.21	7.15	351692630
	50	6.23	6.35	7.84	324983129
W = 15	15	1.93	1.46	1.08	136657482
	30	10.70	5.18	4.18	761878535
	45	9.58	6.5	8.21	551992270
	60	24.7	23.86	25.87	465701505
	75	51.67	53.27	55.08	418735103

We can see a direct negative correlation between the number of predictions ahead and the window size used to predict them. Figure 24 shows this correlation: we plot the accumulated MSE for each multiple of the window size, we remove the volume MSE due to the different scale.
From these results we can conclude that linear models can provide reasonable accurate forecasts in the very short term, up to around 2 intervals ahead of the window size used, before predictions lose relevance to the real-world and just continue the current trend towards infinity or negative infinity.

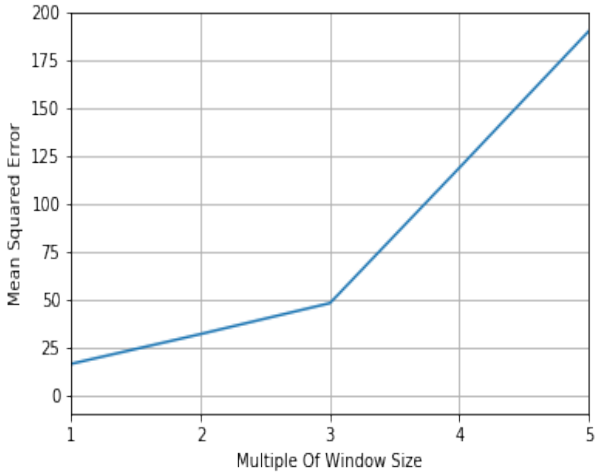

Figure 24: Window Size Influence On Prediction Accuracy: Linear

Non-Linear Model

We now present our non linear model, a MLP with a hidden layer consisting of layers: 20,10,5 using the sigmoid activation function. We first present the MSE for the same window size and n-predictions as with the linear model:

Window Size	n-ahead Predictions	MSE			
		Low	Close	High	Volume
W = 5	5	55.30	0.50	27.57	101849361
	10	174.83	148.83	151.23	332946567
	15	360.0	395.80	314.29	694071649
	20	366.9	431.92	325.10	652970938
	25	354.24	419.44	316.51	634403271
W = 10	10	2635.03	2414.74	1838.25	1085668612
	20	2548.86	2382.44	1802.31	893382404
	30	2486.02	2316.03	1755.67	858509640
	40	2497.68	2322.78	1772.47	793690506
	50	2539.38	2364.51	1805.94	781629135
W = 15	15	544.34	348.67	460.10	92397421
	30	2202.32	1818.97	1417.71	578368966
	45	2499.83	2029.19	1516.94	901441751
	60	2397.0	1923.77	1425.72	903649072
	75	2335.23	1858.07	1360.20	1003210903

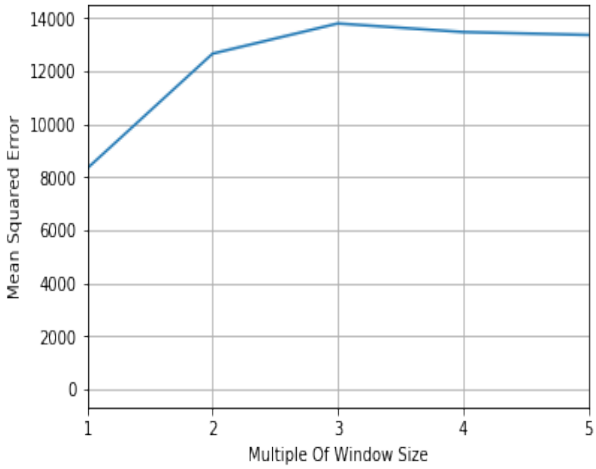

Figure 25: Window Size Influence On Prediction Accuracy: MLP

We can see that the non-linear model is effected less by the increasing n-ahead predictions,although the results are actually considerable worse than using a linear model, we discuss this in chapter 11 but here we are just trying to show the limitations of assuming a linear relationship with financial time series. We show in figure 23 how a linear model performs when all statistical influence of the real-world data has dissipated. This is not the case with non-linear models, see figure 26, where we plot the predicted closing price of the asset 200 hours ahead of the last time series of the training set.

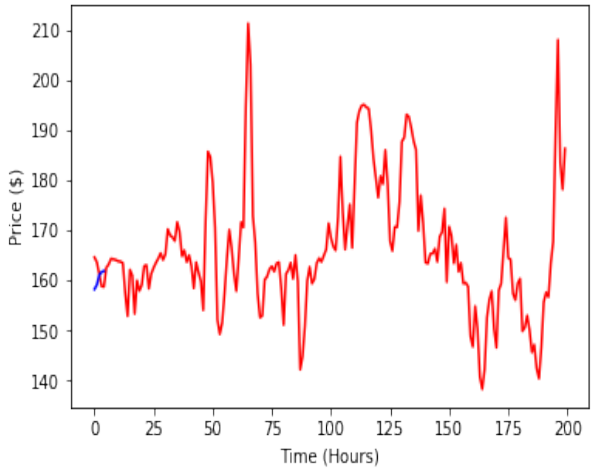

Figure 26: Non-Linear Model Predictions

Clearly the non-linear prediction model actually learns to some extent how the market behaves, as opposed to the linear model finding one particular trend and repeating it indefinitely.

10 Candlestick recognition & Prediction Accuracy

This chapter discusses the effectiveness of using candlestick recognition in the training sets of the prediction models. We use the ETH/USDT 1H data set with AR(100) models, withholding the last 100 hours of trading as a validation set. Results are averaged over 10 iterations and in total 62% of the training data had recognized candle patterns. The results are displayed below and are plotted in figures 27 and 28, we discuss the results in 11.

	MSE			
	Low	Close	High	Volume
Linear With Candlesticks	15	22	35	155430281
Linear No candlesticks	8	7	8	148439819
MLP With Candlesticks	16809	9865	17148	2665544202
MLP No Candlesticks	2185	2820	2334	217179674

	$R^2 Score$			
	Low	Close	High	Volume
Linear With Candlesticks	0.999	0.999	0.999	0.595
Linear No candlesticks	0.995	0.999	0.999	0.591
MLP With Candlesticks	0.997	0.998	0.998	0.896
MLP No Candlesticks	0.997	0.997	0.997	0.788

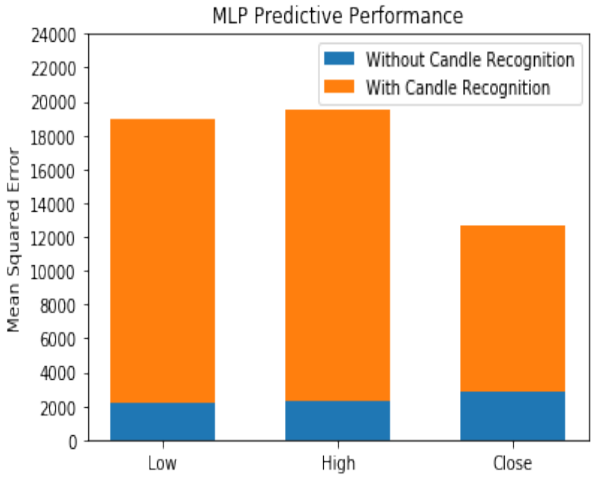

Figure 27: MLP Predictive performance with candlestick recognition

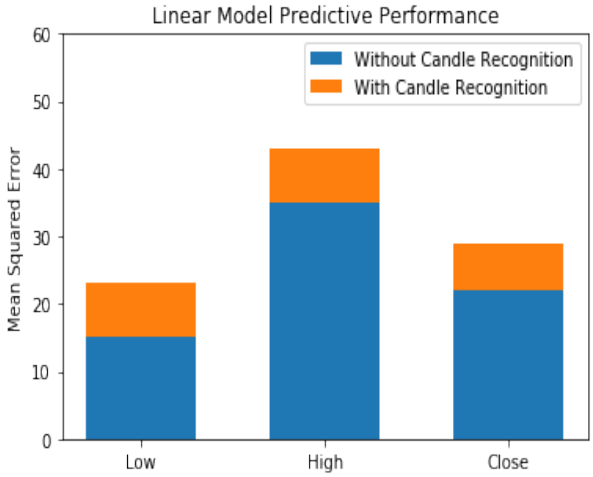

Figure 28: Linear Model Predictive performance with candlestick recognition

11 Project Evaluation

This chapter discuss the limitations of the project and potential solutions to those problems and asses to what degree the project reaches the project aims.

Functional Programming

Before this project we only had experience using imperative programming languages; we believe that we are now more than competent in the use of Haskell. Monads are generally considered to be the most difficult aspect of Haskell to newcomers, specifically the IO monad, have been understood and implemented. This was though a time-consuming task and it came at the cost of having a much less detailed evaluation of prediction models.

Obtaining And Maintaining Training Data

The software developed can effectively obtain and store all data required, provided the appropriate hardware is available. One limitation with regards to this area of the project is that we are unable to actually store all of the data we have access too. We use the university's SQL server to host our database; the university restricts each students storage space to only a few gigabytes and our data set in in the hundreds of gigabytes, additionally our own hardware does not have the available storage space either. We work around this by only storing USDT trading pairs on the Sussex server; we have tested our software to populate every trading pair by using our own machine successfully, but the space needed to be freed up afterwards. The data is kept up top date in real time when the messaging layer is running and can handle multiple users, each continually making requests for data, in parallel.

Generating Predictions

Our project web-page displays model predictions for a specified cryptocurrency, models can be trained in real time and generate predictions for a predetermined amount of intervals ahead. The limitation here is that due to the computational cost and time required we are yet to find optimal models to be used. One route that we are considering for future work is to instead try and predict trading ranges instead of trying to make accurate price predictions, which should be less computational expensive and perhaps, due to the error margins of the prediction models, more helpful for the users.

Evaluating Aspects of T.A For Improving Prediction Accuracy

We evaluated candlestick patterns as a tool for improving prediction accuracy. Our results are not conclusive; it is clear from the high difference in prediction accuracy that the addition of candlestick recognition does have a significant impact on predictions but the results from the linear model suggest not so much. We believe that the results of the non-linear model are distorted by the low recognition rate (62%) much more than the linear model due to the non-linear assumptions it makes. To have conclusive results we would need to increase the recognition rate which would mean creating novel candlestick types. This would be time consuming as we would need to analyze the data and try to find correlations between a specific pattern and short term market movements for a unknown amount of different pattern formations, though this is an area that could be worked on in the future.

12 Conclusions

We set out to create software where the primary goal was to generate price predictions for Cryptocurrencies. This goal was broken down into three main components: Obtaining & Maintaining the data sets, Creating prediction models and then presenting these to the user to assists in trading decisions. We then decided to develop the software in Haskell, due to it's prevalence in the financial sector and to challenge ourselves to learn the functional programming paradigm. We consider all of these objectives have been met, although it is currently only a proof of concept. We had to work around a number of limitations, specifically computational power and storage space, to advance the application would require acquisition of hardware which would come at a non-trivial cost. Our assessment of prediction models is flawed and their are better models available for our specific data set but we did not have time to implement them.

13 Future Work

This section presents some ideas for future work on the project.

13.1 Predicting Support/Resistance lines

During development of the prediction models, we made a mistake in our code; instead of using the predicted price of an asset as the opening price of the asset at the next interval of trading we instead assigned the opening price of the last know interval to each prediction. I.e the opening price of the last element of the training set was assigned to every subsequent element, producing some unexpected and interesting results:

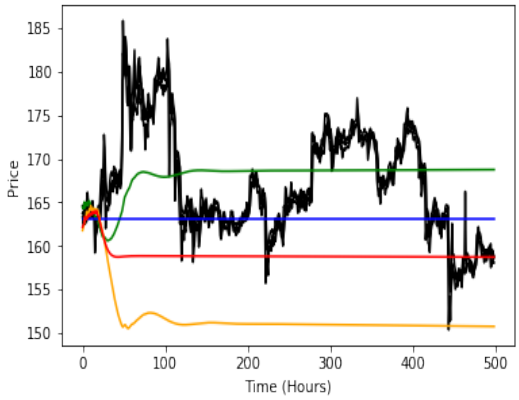

Figure 29: Trading Range Predictions

Using the same data set as in 7.3.2: the black line is the actual price of the asset (all price values: High,Low,Close,Open).Red is the predicted low with window size 5, blue is predicted close window size 15, orange is predicted low window size 15 and green is predicted high window size 15.Each of the lines appear to form long-term trading ranges: In T.A a trading range is formed by support/Resistance zones, when a resistance zone is broken it becomes a support zone and vice versa: a perfect example being the red line in 29 where the support is tested 4 times: 3 times between 100:250 hours and a fourth time at roughly 410 hours before being broken at roughly 450 hours and then becoming a resistance zone between 450:500 where the assets' price is consolidating and the red line will become either a resistance or support zone afterwards. We have looked at this with varying data sets and window sizes and this example is not an anomaly, although in some cases the predictions do not seem to be relevant to the assets price the majority of cases do seem to act as trading ranges.

References

[1] Charles August Lindbergh. *The Economic Pinch, 1923. pg.95*

[2] Stuart J. Russell, Peter Norvig (2010) *Artificial Intelligence: A Modern Approach, Third Edition, Prentice Hall*

[3] Satoshi Nakamoto (2008) *Bitcoin: A Peer-to-Peer Electronic Cash System*

[4] Steve Nison *Japanese Candlestick Charting Techniques: A Contemporary Guide to the Ancient Investment Techniques of the Far East.* November 2001. New York Institute of Finance.

[5] Paul Hudak (1989) *conception,evolution and application of functional programming.* Yale University Department of Computer Science

[6] Chatham, Mark (2012). *Structured Query Language By Example - Volume I: Data Query Language. p. 8.*. ISBN 978-1-29119951-2.

[7] *https://www.influxdata.com/time-series-database/*

[8] *https://www.mongodb.com/nosql-explained*

[9] *https://www.percona.com/blog/2017/02/10/percona-blog-poll-database-engine-using-store-time-series-data/*

[10] Cybenko, G. 1989. *Approximation by superpositions of a sigmoidal function Mathematics of Control, Signals, and Systems, 2(4), 303–314.*

[11] Zell, Andreas (1994) *imulation Neuronaler Netze [Simulation of Neural Networks] (in German) (1st ed.). Addison-Wesley. p. 73.* ISBN 3-89319-554-8.

[12] Teijiro Isokawa1, Haruhiko Nishimura and Nobuyuki Matsui (2012) *Quaternionic Multilayer Perceptron with Local Analyticity* ISBN 3-89319-554-8.

[13] Bayer, R. McCreight, E. (1972) *Organization and Maintenance of Large Ordered Indexes"*

wpage

A
Messaging Layer Flowchart

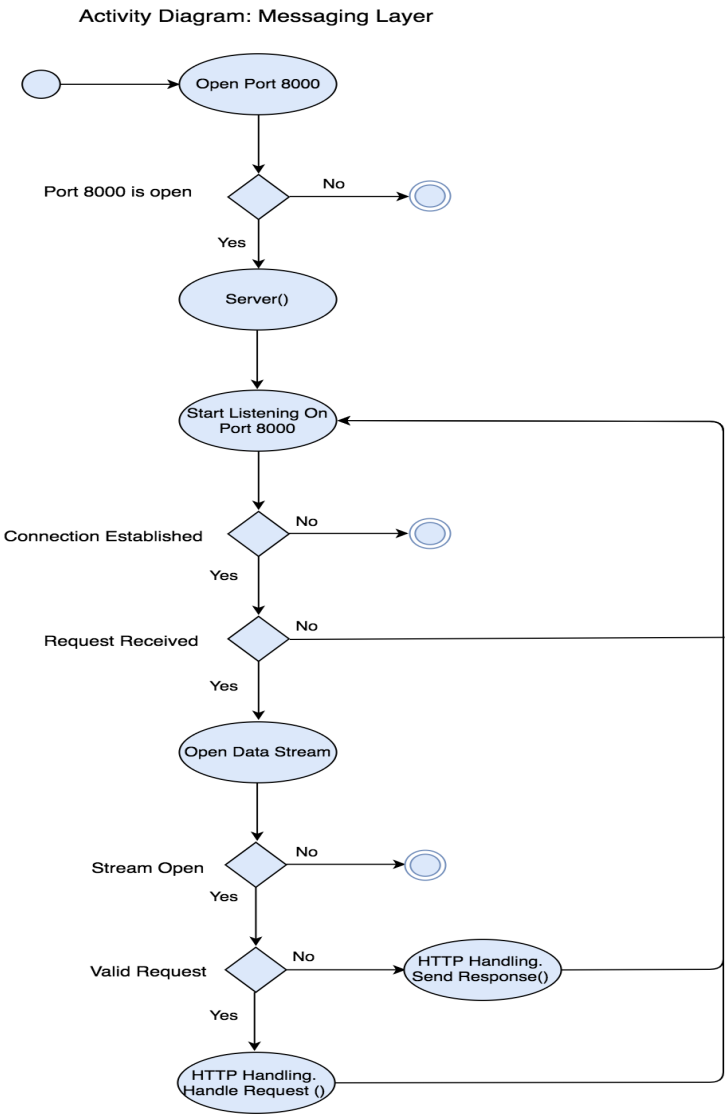

Figure 30: Messaging Layer Flow

B Activity Diagram: Receiving Trading Data

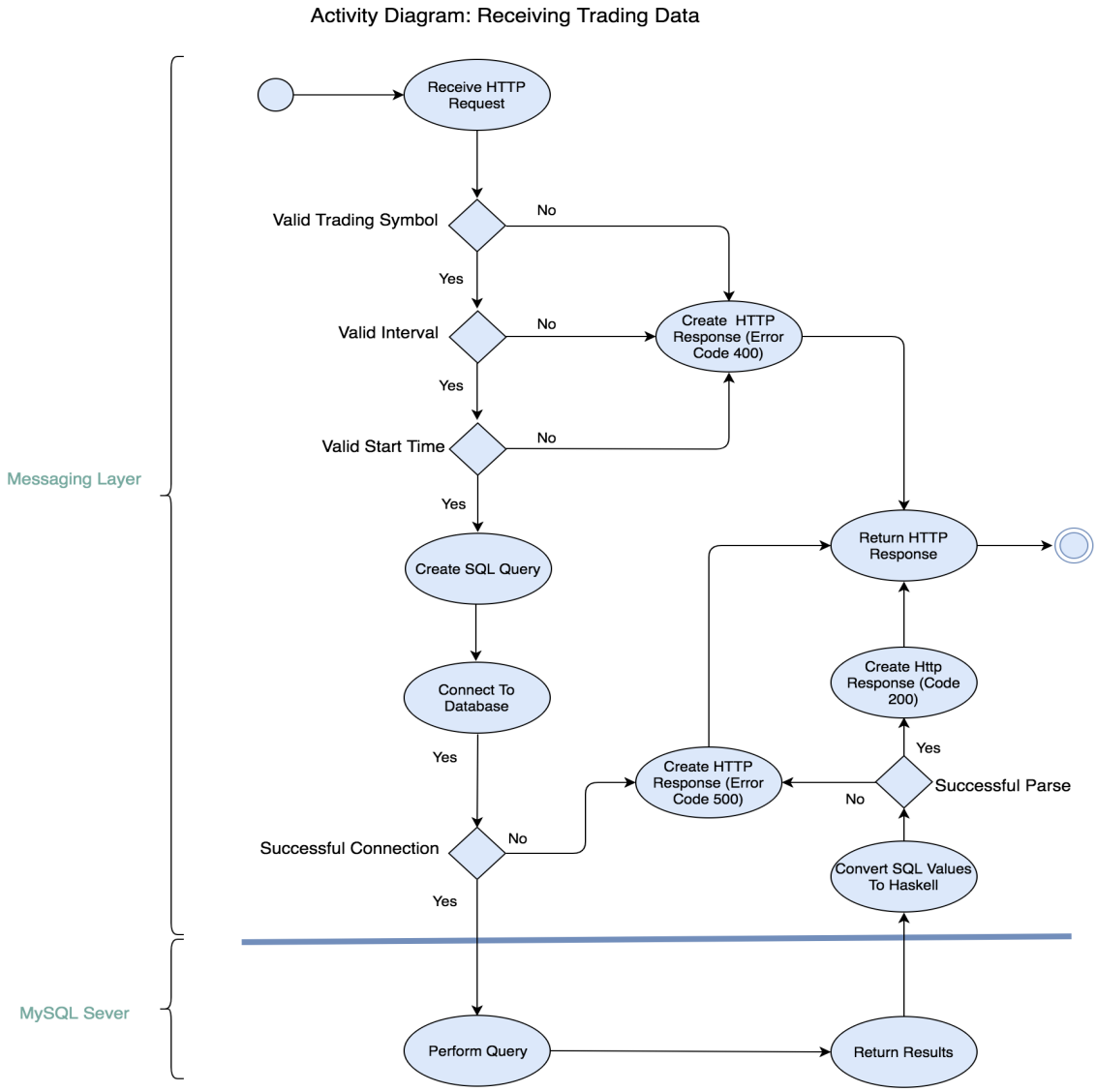

Figure 31: UML Activity: Receiving Trading Data

C Sequence Diagram : Retrieving Trading Data

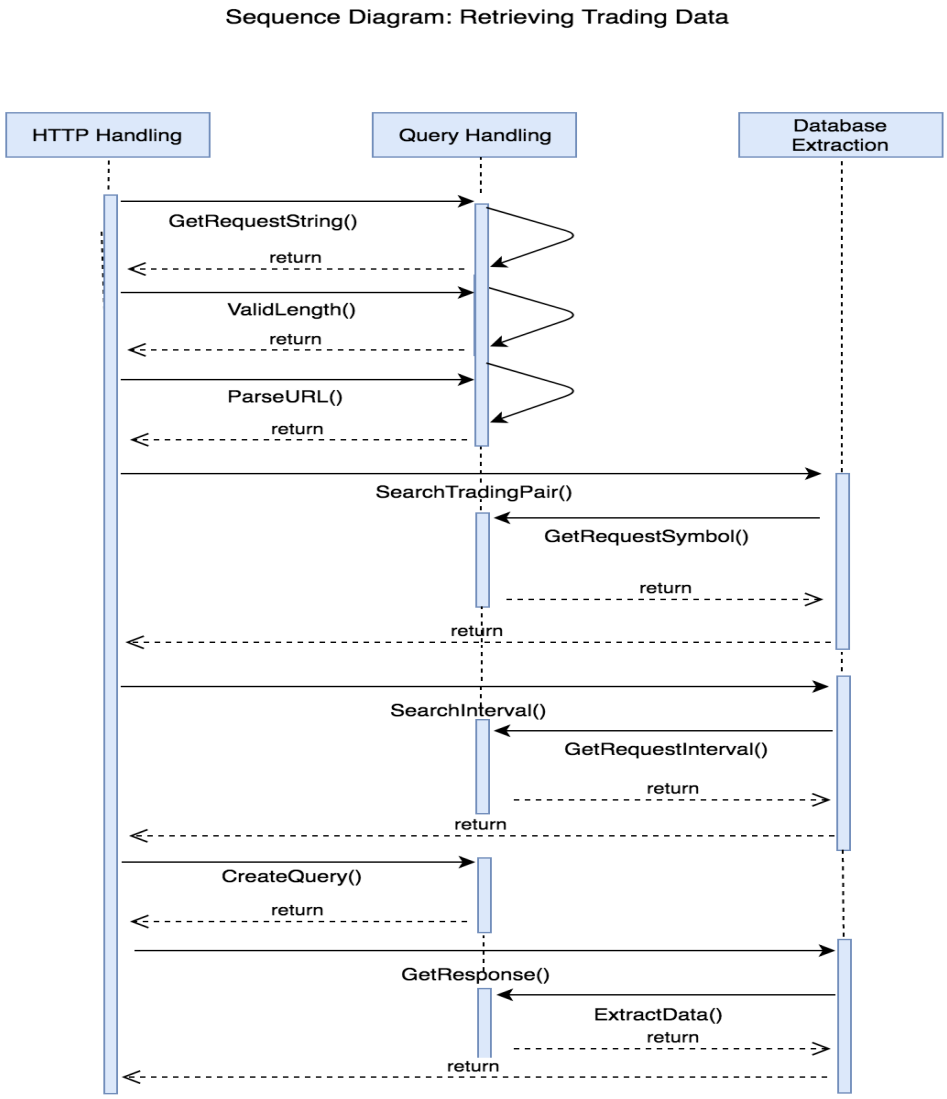

Figure 32: UML Sequence: Retrieving Trading Data