Bitcoin Price Prediction

CS 4240 Spring 2018

Jinhyeong Kim

Sunday, April 22, 2018

Introduction

In recent spike of interest of cryptocurrency and blockchain technology, people have begun researching and building new businesses. Billions of dollar have been invested in the technology despite its incomplete and premature nature. General consensus is that cryptocurrency and blockchain will stay and will prove themselves useful in future. In addition, the market cap of cryptocurrency has grown tremendously for past few months, and people and institutions are diving into cryptocurrency market to gain profit. However, unlike stocks and bank note exchanges, cryptocurrency market is highly volatile due to its independence to other economic factors. The major factors that influence the market are emotional drives from general public, media outlets' portrayal of the market, and promises made by cryptocurrency developers. These factors cannot be easily computed into reliable mathematical models, which are necessary for traders. The goal of this final project was to see if building a mathematical model is completely unreasonable for cryptocurrency market and attempting to predict future cryptocurrency value is futile. The purpose of this final project is not to find the best model to predict the future prices, but to explore historical data and manipulate them with knowledge gained from CS 4240 course. By the end of this project, these questions will be answered:

- What is the accuracy of each mathematical/statistical model applied to the aggregated data?
- How different is the predicted future price from the actual price point?

Data

In order to predict future price values, historical transaction prices, input features, and machine learning models were needed. Bitcoin was selected to be explored within the scope of this final project. There were several constraints that needed to be considered when desired datasets were decided to be collected.

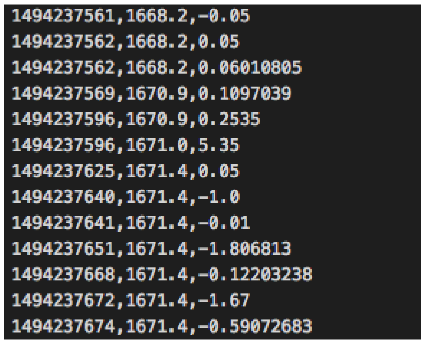
Constraints

1. Bitcoin is being traded all over the world.
2. Each country has several exchange platforms.
3. Bitcoin can be traded with either banknotes or any other cryptocurrency.
4. There is no stable/predictable input feature whose data are publicly available.

Due to infeasibility of collecting every transactions all over the world, only three regions were selected for the investigation: USA, Europe, and Japan, from which most of Bitcoin transaction volumes originated. Within those regions, the popular methods of purchasing/selling Bitcoin are banknote-Bitcoin exchange, Tether-Bitcoin exchange, and XRP-Bitcoin exchange. If cryptocurrency-to-cryptocurrency exchanges were to be accounted for this project, then historical prices for each cryptocurrency had to be collected, which are difficult in terms of the number of hours required for researching and cleaning. Therefore, only banknote-Bitcoin exchange method was chosen.

For USD, 4 platforms were chosen: Bitfinex, Bitstamp, Coinbase, and Kraken. For EUR, 2 platforms were chosen: Coinbase and Kraken. For JPY, Bitflyer and coincheck were chosen.

Figure 1: Data from Bitcoinchart



```
1494237561,1668.2,-0.05
1494237562,1668.2,0.05
1494237562,1668.2,0.06010805
1494237569,1670.9,0.1097039
1494237596,1670.9,0.2535
1494237596,1671.0,5.35
1494237625,1671.4,0.05
1494237640,1671.4,-1.0
1494237641,1671.4,-0.01
1494237651,1671.4,-1.806813
1494237668,1671.4,-0.12203238
1494237672,1671.4,-1.67
1494237674,1671.4,-0.59072683
```

Machine learning models required ample number of data points for better accuracy; therefore, secondly data were searched. Unfortunately, each platform either provided historic data averaged daily, or data averaged minutely but only for past month. Due to the limitation of public data, the process for this project delayed significantly.

Fortunately, there was this website called Bitcoinchart [1] which aggregated all of historical transactions for all of the platforms that were needed for this project. The datasets were not averaged in any time frame, but recorded every transaction that had happened since the beginning of each platform service. The only downside of this website was its speed of updating the data, a factor that was negligible for this project.

Figure 1 shows raw data obtained from Bitcoinchart. First column is UNIX epoch time in milliseconds (10 digits). Second column is the price of the particular transaction. Third column is the amount purchased or sold. Negative number indicates sold amount, while positive number indicates the amount purchased through the particular platform.

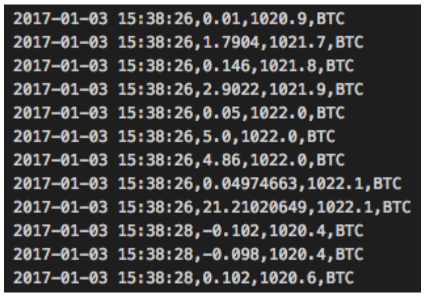
Bitcoinchart however did not provide any data for Bitfinex since Dec 22, 2016. The rest of data point from Dec 22, 2016 until first week of April 2018 was retrieved from someone's personal project [2]. The website offered the data from various platforms, including Bitfinex, for free. Data were separated in multiple files aggregated monthly. Total of 17 files were downloaded from the website. Figure 2 shows the snippet of dataset. First column is human-readable date. Second column is amount purchased/sold. Third column is the price, and the last column is the symbol of cryptocurrency exchanged.

In addition, because the unit of price from each region is not unified, data for exchange rate were necessary. There were no free public data that provided exchange rates in small intervals, and the ideal datasets for this project cost too much, due to high demand for such data for high frequency trading firms. Therefore, a compromise was made to retrieve data aggregated daily. Figure 3 shows a snippet of raw EUR-USD conversion data obtained from OFX
[3]. PointInTime is the timestamp in UNIX timestamp in microseconds. InterbankRate is the conversion rate, where the amount in EUR is multiplied by InterbankRate to get the amount in USD.

Input features

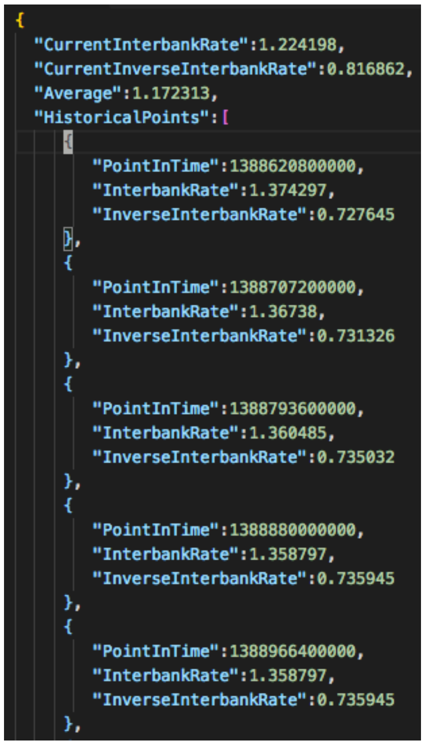
Besides historical price data, input features were required for machine learning models. If only timestamp and price data were plotted in the model, the result would be meaningless, as it would be equivalent to the result of plotting random points. For input

Figure 2: Data from Cryptodataset



```
2017-01-03 15:38:26,0.01,1020.9,BTC
2017-01-03 15:38:26,1.7904,1021.7,BTC
2017-01-03 15:38:26,0.146,1021.8,BTC
2017-01-03 15:38:26,2.9022,1021.9,BTC
2017-01-03 15:38:26,0.05,1022.0,BTC
2017-01-03 15:38:26,5.0,1022.0,BTC
2017-01-03 15:38:26,4.86,1022.0,BTC
2017-01-03 15:38:26,0.04974663,1022.1,BTC
2017-01-03 15:38:26,21.21020649,1022.1,BTC
2017-01-03 15:38:28,-0.102,1020.4,BTC
2017-01-03 15:38:28,-0.098,1020.4,BTC
2017-01-03 15:38:28,0.102,1020.6,BTC
```

Figure 3: Data from OFX



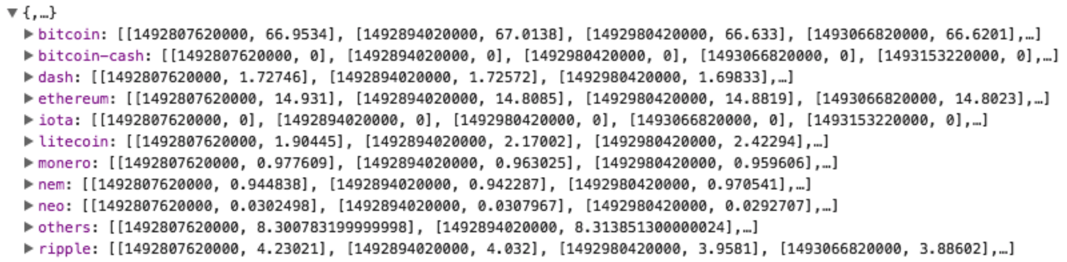
```
{
  "CurrentInterbankRate":1.224198,
  "CurrentInverseInterbankRate":0.816862,
  "Average":1.172313,
  "HistoricalPoints":[
      {
        "PointInTime":1388620800000,
        "InterbankRate":1.374297,
        "InverseInterbankRate":0.727645
      },
      {
        "PointInTime":1388707200000,
        "InterbankRate":1.36738,
        "InverseInterbankRate":0.731326
      },
      {
        "PointInTime":1388793600000,
        "InterbankRate":1.360485,
        "InverseInterbankRate":0.735032
      },
      {
        "PointInTime":1388880000000,
        "InterbankRate":1.358797,
        "InverseInterbankRate":0.735945
      },
      {
        "PointInTime":1388966400000,
        "InterbankRate":1.358797,
        "InverseInterbankRate":0.735945
      },
```

[1] http://api.bitcoincharts.com/v1/csv/
[2] https://www.cryptodatasets.com/platforms/Bitfinex/BTC/
[3] https://www.ofx.com/en-us/forex-news/historical-exchange-rates/

features, the total market cap and the Bitcoin dominance percent-
age in the cryptocurrency market were chosen. The market cap was chosen as Bitcoin price was highly correlated to the trend of the market cap. If majority of cryptocurrencies was performing bad and the total market cap decreases, the Bitcoin price would also decrease, and the relationship held true vice versa. The dominance percentage means the total volume of Bitcoin divided by that of entire cryptocurrency market. The reason for choosing this as an input feature is that other minor cryptocurrency are not readily available in banknotes, as exchange platforms do not support newly created, or unpopular cryptocurrency exchanges. Only method to purchase minor coins is to through Bitcoin or major coins, such as Ethereum. If there were several minor coins on exponentially rising trend and the dominance of Bitcoin was falling due to other cryptocurrencies taking the place of Bitcoin, the price of major coins would rise as people would purchase them to exchange them for minor coins.

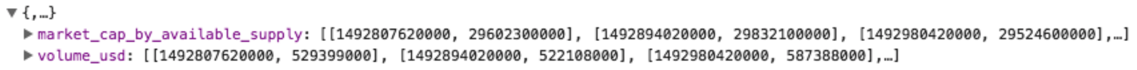
Data for the input features were obtained from Coinmarketcap website [4]. Figure 4 shows the snippet of data for Bitcoin dominance percentage. This dataset was available from 2013-04-28, with an interval of 5 minutes since then. Figure 5 shows the snippet of data for the total market cap of cryptocurrency market. The availability of this dataset was same as Bitcoin dominance dataset.

Figure 4: Data schema of Bitcoin dominance from Coinmarketcap



```
▼{,…}
  ▶bitcoin: [[1492807620000, 66.9534], [1492894020000, 67.0138], [1492980420000, 66.633], [1493066820000, 66.6201],…]
  ▶bitcoin-cash: [[1492807620000, 0], [1492894020000, 0], [1492980420000, 0], [1493066820000, 0], [1493153220000, 0],…]
  ▶dash: [[1492807620000, 1.72746], [1492894020000, 1.72572], [1492980420000, 1.69833],…]
  ▶ethereum: [[1492807620000, 14.931], [1492894020000, 14.8085], [1492980420000, 14.8819], [1493066820000, 14.8023],…]
  ▶iota: [[1492807620000, 0], [1492894020000, 0], [1492980420000, 0], [1493066820000, 0], [1493153220000, 0],…]
  ▶litecoin: [[1492807620000, 1.90445], [1492894020000, 2.17002], [1492980420000, 2.42294],…]
  ▶monero: [[1492807620000, 0.977609], [1492894020000, 0.963025], [1492980420000, 0.959606],…]
  ▶nem: [[1492807620000, 0.944838], [1492894020000, 0.942287], [1492980420000, 0.970541],…]
  ▶neo: [[1492807620000, 0.0302498], [1492894020000, 0.0307967], [1492980420000, 0.0292707],…]
  ▶others: [[1492807620000, 8.300783199999998], [1492894020000, 8.313851300000024],…]
  ▶ripple: [[1492807620000, 4.23021], [1492894020000, 4.032], [1492980420000, 3.9581], [1493066820000, 3.88602],…]
```

Figure 5: Data schema of the total market cap from Coinmarketcap



```
▼{,…}
  ▶market_cap_by_available_supply: [[1492807620000, 29602300000], [1492894020000, 29832100000], [1492980420000, 29524600000],…]
  ▶volume_usd: [[1492807620000, 529399000], [1492894020000, 522108000], [1492980420000, 587388000],…]
```

After all data were collected, there were uploaded to Google Cloud Storage. The reason for choosing Google Cloud Platform (GCP) over Amazon Web Service (AWS) was that AWS did not provide free credits readily.

Computation

Overview

Data retrieved from Bitcoinchart were cleaned and formatted appropriately already from Bitcoinchart. One reasonable thought on why Bitcoinchart has been providing cleaned data is to minimize the data transfer from the website, thereby minimizing the overall cost needed freely offer data to public. The cleaned data are not suitable particularly for advanced high frequency trading algorithm traders, who need other data points besides price of each transaction, but for this final project, Bitcoinchart basically did cleaning data portion, thereby decreasing the overall work required to complete the project.

For Bitfinex platform, data needed to be combined and unified into one file. There were 18 files for Bitfinex platform, one from Bitcoinchart and 17 from Cryptodataset. JavaScript was used to iterate through all 17 files, clean data, and append the data to the file obtained from Bitcoinchart. Finally, the combined file was uploaded to Storage.

[4]https://graphs2.coinmarketcap.com/global/dominance/

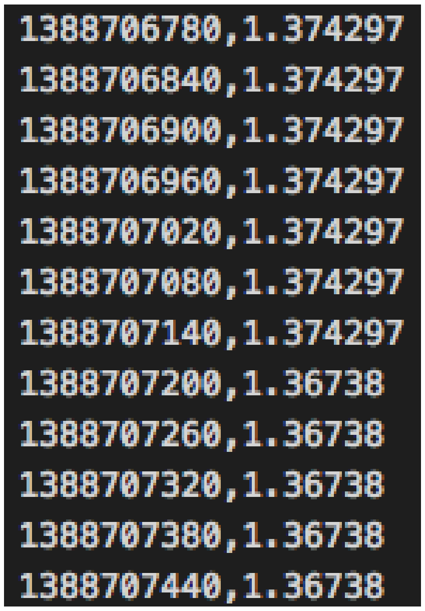
Data in the Storage were then fed into Scala program that ran inside a Spark Cluster created by Google Dataproc service. Scala program consists of aggregating data from all platforms for each region, apply exchange rates to data from EUR and JYP regions, average the prices minutely, join averaged price with market cap, and dominance percentage data, and produce a single CSV file. The produced CSV file was then fed into the Python program on local machine, which used Numpy, Pandas, and scikit-learn libraries to generate accuracy values of several machine learning models. The future price was also predicted using Python.

Cleaning Data

For Bitfinex data, 18 files had to be combined into one file for reduce confusion during the actual computation. This process was done purely with JavaScript. The files fetched from Cryptodataset are named as bitfinexRaw$i$.csv, where i is from 1 to 17, and the file responsible for accumulating all of data is named as bitfinexUSD.csv. Using fs built-in library for reading files in Node run-time environment and CSV parsing library for easier streaming, data schema shown in figure 2 were converted into same format as schema shown in figure 1. One tricky part about cleaning the data was the timestamp. *moment* library was used to convert calendar date to UNIX epoch timestamp. Initially, *moment* library $valueOf$ function was converting the calendar date into microseconds, resulting too many digits for the epoch timestamp. The difference was not realized until the data were fed into Scala program, in which aggregation by date was not working as intended, and incorrect dates were showing. The problem was simply solved by dividing the value by 1000 to make the timestamp value a true epoch timestamp. JavaScript code is available in the following directory inside the Github repository: *src/main/node*. The file name is *cleanData.js*. The aggregation of data did not require a Spark cluster, and was performed inside a local machine.

Cleaning conversion rate data shown in figure 3 was also done in JavaScript. The downside of the collected dataset was its huge interval. Data are planned to be aggregated in minute interval, but because the smallest interval available for exchange rates is daily, some deviation from detailed and perfect calculation had to be made. Fortunately, based on historical data, the exchange rate data rarely fluctuated, meaning using daily exchange rates, under reasonable assumptions, would not be detrimental to the results. However, when minutely aggregated Bitcoin price data were to be joined the daily exchange rates, there would be gaps. Therefore, JavaScript was once again utilized to duplicate daily exchange rates 1440 times, so that every minute of a particular day would have a datum. The cleaned data is shown in figure 6, first column indicating timestamp, and the second column indicating the exchange rate.

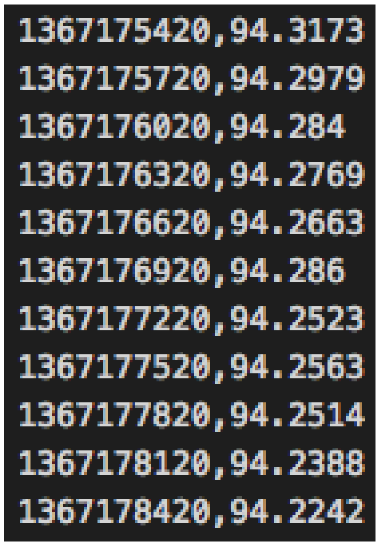
Figure 6: Cleaned exchange rate data



```
1388706780,1.374297
1388706840,1.374297
1388706900,1.374297
1388706960,1.374297
1388707020,1.374297
1388707080,1.374297
1388707140,1.374297
1388707200,1.36738
1388707260,1.36738
1388707320,1.36738
1388707380,1.36738
1388707440,1.36738
```

For input features, there was no static file that provided historical data. API had to be utilized in order to fetch data. JavaScript was used to fetch data from beginning of April 28th 2013 up to the time the data were fetched, which was first week of April. Each API request only provided a day worth of data − 288 points − in five-minute interval. The overall time to fetch both the market cap and dominance data was approximately 50 minutes, and the total size of data was 20 MB. The raw data are shown in figure 4 and figure 5, and the cleaned data schema is shown in figure 7. Figure 7 specifically shows first column as timestamp, and second column as dominance percentage. The cleaned data for the market cap are almost identical, except the second column having different data.

All of CSV files containing cleaned data had a column of epoch timestamp, and other columns of only necessary information that would be used in calculations.



```
1367175420,94.3173
1367175720,94.2979
1367176020,94.284
1367176320,94.2769
1367176620,94.2663
1367176920,94.286
1367177220,94.2523
1367177520,94.2563
1367177820,94.2514
1367178120,94.2388
1367178420,94.2242
```

Figure 7: Cleaned exchange rate schema

Aggregation

In Scala program with Spark framework, Dataframes are used instead of RDD due to clearly known schema of individual CSV file being fed into the program, and by utilizing Catalyst and Tungsten, the optimization of queries was guaranteed. In addition, errors in data schema, calculation, or aggregation were able to be picked up prior to the actual computation, which saved tons of development time. Three parts that will be discussed in this report are:

1. Aggregating price in each region
2. Joining all of regions after unifying currency and with the averaged price with other input feature values
3. Python code for applying machine learning model

Each region included historic data from at least one platform, and they need to be appended, then be aggregated in a specific interval. The functions responsible for aggregating price for each region are shown below. The parameter *name* for *aggPrice* is among *usd*, *jyp*, and *eur*. Its helper function, *fetchDataAndConvertToDf*, reads a file based on *name* straight from Google Cloud Storage, converts RDD[String] to RDD[Row] for conversion to Dataframe. After RDD[Row] is converted to Dataframe, the first column, which should be timestamp column for all of CSV file, is casted as StringType, while any other columns are casted as FloatType. The timestamp then is proceeded to be casted as LongType to be converted from UNIX timestamp to a human-readable format, using "yyyy-MM-dd HH:mm" format, to group data by one minute interval. Utilizing RelationalGroupedDataset object, grouping is done easily. After *fetchDataAndConvertToDf* function, Dataframe with column *name* specified by the third input parameter, column "count", and column "Time" will be available. If *name* is *usd*, then there is no need to apply exchange rate, so return whatever returned from *fetchDataAndConvertToDf*. If *name* is not *usd*, then exchange rate should be applied. The file name for EUR-USD exchange rate data is eur_usd, and for JPY-USD data is jpy_usd. These names are specified as a semi-global variable outside of the main function. The exchange rate data are fetched using *fetchDataAndConvertToDf* again in line 7, and averaged ratio is obtained in line 8. The columns that serve no purpose are dropped as soon as possible throughout the program. Now, in line 9, minutely accumulated price data is joined with exchange rate data with leftouter technique, which guarantees all of minutes for price data to remain. One might ask, "What if there is a row with NULL value for exchange rate, which may occur when the exchange rate data do not have the particular minute in its Dataframe?" This situation will never occur because when the datasets for exchange rate data were fetched, their time intervals were larger than that of the historical price datasets. Once joined, the price column and ratio column are multiplied, and *aggPrice* finishes. By the end of *aggPrice*, Dataframe with columns "Time", "count_*name*", and "Price_*name*" is outputted. The reason for having specific names will become clear once all Dataframe from each region are congregated.

```scala
1  // first aggregation
2  def aggPrice(name: String): DataFrame = {
3      val df = fetchDataAndConvertToDf(name, List("Price", "Amount"), "Price")
4      println("Date column added " + name)
5
```

```scala
    if (name != "usd") {
      val exchangeDf = fetchDataAndConvertToDf(name + "usd", List("Ratio"), "Ratio")
        .withColumn("Ratio", (col("Ratio") / col("count"))).drop("count")
      df.join(exchangeDf, Seq("Time"), "leftouter")
        .withColumn("Price_" + name, col("Price") * col("Ratio")).drop("Ratio", "Price")
        .withColumnRenamed("count", "count_" + name)
    } else {
      df.withColumnRenamed("Price", "Price_" + name)
        .withColumnRenamed("count", "count_" + name)
    }
}

def fetchDataAndConvertToDf(name: String, fields: List[String], sumBy: String): DataFrame = {
    println("Fetching ..." + name)
    val rdd = parseData(spark.sparkContext.textFile(fileName(name)))
    println("Data parsed for " + name)
    val df = applyTimeFormat(spark.createDataFrame(rdd, schema(fields))).cache()

    println("Grouping and averaging " + name)
    val count = df.groupBy("Time").count()
    val sum = df.groupBy("Time").sum(sumBy)
    count.join(sum, Seq("Time"), "leftouter")
      .withColumnRenamed("sum(" + sumBy + ")", sumBy)
}
```

After Dataframe has shaped up for each region, it is time to join all of them together to be averaged. Fullouter technique is used for joining to account for days when there were only a few transactions in only in one region, and the days when JPY platforms did not start service years after USA platforms had started. One mistake was discovered at this stage of code, where some values did not show up. The knowledge of NULL object in Scala was not fully studied; therefore, the fact that adding any number to NULL ends up as NULL was not realized until printing to console for debugging. The solution was to use $.na.fill$ function to replace all NULL values with 0.0. After two fullouter joins, Dataframe with time column, count and price columns from each region is generated. One inconvenience found when dealing with Dataframe is that there is no way to add or perform any arithmetic operation on columns with same name. Initially, there were multiple columns named "Price" and "Count." However, due to limited API offered by Dataframe, those columns had to be named differently, hence specific names for price and count were generated from *aggPrice* function. The price and count are added, then divided to obtain the average price for each minute interval. Used columns are dropped to minimize the size of data transfers.

Now the averaged price data are ready, it is time to finalize the computation by joining the data for input features. Input features are converted to Dataframe similarily as price data. Because input features do not need to be combined with any other dataframe, they are grouped then averaged immediately with RelationalGroupedDataset API. The input features have same time interval, same start date, and end date. Any join operation would have outputted same result. Here in line 34, dominance dataset is joined with market cap dataset using leftouter, and then the joined dataframe is joined again with averaged price dataset also using leftouter technique. The earliest timestamp for averaged price data is Mar 31st, 2013, which is from Bitfinex platform, while the earliest timestamp for both input features is April 28th, 2013. Fullouter technique could have been used, but the end result would have outputted more rows with NULL values. The only reason for using leftouter is to remove rows with empty fields.

```scala
def addAndAverage(usd: DataFrame, eur: DataFrame, jpy: DataFrame): DataFrame = {
    val joined = usd.join(eur, Seq("Time"), "fullouter").join(jpy, Seq("Time"), "fullouter")
      .na.fill(0.0, Seq("Price_usd", "count_usd", "Price_eur", "count_eur", "Price_jpy", "count_jpy"))

    joined
      .withColumn("Price", joined("Price_usd") + joined("Price_eur") + joined("Price_jpy"))
      .withColumn("Count", (joined("count_usd") + joined("count_eur") + joined("count_jpy")))
      .withColumn("Price", (col("Price") / col("Count")))
      .drop("Count", "Price_usd", "Price_eur", "Price_jpy", "count_usd", "count_eur", "count_jpy")
    }


def joinWithOtherData(agg: DataFrame): DataFrame = {
    val domRdd = parseData(spark.sparkContext.textFile(fileName("dominance")))

    val dominanceDF = applyTimeFormat(spark.createDataFrame(domRdd, schema(List("Percent"))))
      .groupBy(col("Time")).avg("Percent").withColumnRenamed("avg(Percent)", "Dominance %")

    val capRdd = parseData(spark.sparkContext.textFile(fileName("cap")))

    val capDF = applyTimeFormat(spark.createDataFrame(capRdd, schema(List("Cap"))))
      .groupBy(col("Time")).avg("Cap").withColumnRenamed("avg(Cap)", "Cap")

    dominanceDF
      .join(capDF, Seq("Time"), "leftouter")
      .join(agg, Seq("Time"), "leftouter")
}
```

After all computation, Dataframe is saved as a CSV file back to Google Cloud Storage for Python program to consume.

The final portion of this final project is the python program. Below is the code for building the regression model. First, the data are read from Google Cloud Storage and transformed to Dataframe using Pandas library. The test dataset for linear regression model is 20%, and the training dataset is 80%. Using scikit-learn library, classifier is declared as LinearRegression model. line 9 to 12 separate input features from Bitcoin price for both test and train datasets. Train datasets are then fed into classifier in line 14 to get the accuracy score. Variance is also calculated in line 16. The rest of code is organizing the dataset for plotting using Matplot library.

```
1  file = "final.csv"
2  df = pd.read_csv(file, names=["Time", "BitDom", "MarketCap", "Price"])
3  train_perc = 0.8
4  test_perc = 0.2
5  train_data, test_data = train_test_split(df, test_size=test_perc)
6  train_data['Time'] = train_data['Time'].apply(convertToUnix).astype(int)
7  test_data['Time'] = test_data['Time'].apply(convertToUnix).astype(int)
8  clf = LinearRegression()
9  x_train_data = train_data.drop('Price', axis=1)
10 y_train_data = train_data[['Price']]
11 x_test_data = test_data.drop('Price', axis=1)
12 y_test_data = test_data[['Price']]
13
14 clf.fit(x_train_data, y_train_data)
15 print("Linear Regression Accuracy: ", clf.score(x_test_data, y_test_data))
16 y_pred = clf.predict(x_test_data)
17 print("Linear Regression r2_score: ", r2_score(y_pred, y_test_data))
18 x_test_data_lr = x_test_data
19 x_test_data_lr['Price'] = pd.DataFrame(y_pred, index=x_test_data_lr.index)
20 sorted_x_test_data_lr = x_test_data_lr.sort_values(by="Time")
21 plt.xlabel("Time")
22 plt.ylabel("Price")
23 plt.title("Linear Regression - Bitcoin(USD) Price Predictions")
24 plt
25   .plot(sorted_x_test_data_lr['Time'][400000:]
26   .apply(pd.Timestamp.utcfromtimestamp), sorted_x_test_data_lr['Price'][400000:])
27 plt.gcf().autofmt_xdate()
28 plt.savefig("lr_rice_predictions")
```

Challenges

Initially Google BigQuery was decided to be used to fetch data rather than reading the data straight from Storage. BigQuery enabled specifying schema on CSV files, fast querying for large data, and SQL syntax to avoid troubles converting timestamp to human-readable format and filtering data to desired timeframe. Overall, BigQuery provided convenience while the program was being developed. However, once Google Dataproc was established and the job ran, the program complained that there was NoSuchMethod exception, which occurs only when library is missing the functions that are being utilized within the program. Everything worked completely fine on local machien, but as soon as the program was transferred to Google VM machines, things started to tear apart. The source code for Google BigQuery library was explored to investigate where the issue might rise from. All versions, including beta versions, for BigQuery library were tried. Interestingly, one version below the latest version worked for one day, and the next day everything failed again. After countless StackOverflow posts were read, the issue was due to dependency conflicts between the library being provided by the program and the pre-installed Google libraries inside Google VM. Google VM established by Dataproc installed various packages which came to conflict with the packages specified in sbt file. One StackOverflow suggested using a more general Google Cloud library that contains all necessary functions for all sorts of Google Cloud services, rather than using a very specific library for a particular service. The library was replaced, and everything began to work on Dataproc.

Unfortunately, the dependency issue was resolved when only small subset of BigQuery data was being fetched. When the code for program was changed to allow all of data to be fetched, the program hung in the beginning of the program. Outputs from Catalyst and Tungsten did not appear, and the program eventually died with Memory exception after 4 hours. Different set of Spark cluster with higher memory capacity was set up to run the program, but the result was hopeless. Google BigQuery was causing too many issues that were unrelated to the purpose of this project; therefore, it was abandoned and the Scala code was modified to read straight from Google Cloud Storage. Every computation ran very smooth afterwards.

Besides the library conflict, one minor challenge was that when Dataframe was about to be saved to CSV file, the program was partitioning Dataframe, no matter how small it is, to 200, making the entire program for a trivial amount of data very long. In addition, the program was written to output one CSV file after all computation, but one CSV file was outputted from each partition, making a total of 200 CSV files, most of them 0 byte, saved in Google Storage. This problem was resovled by repartitioning Dataframe to 20 at the end of computation, and using Unix commands on terminal to combine the data.

Tools used

The libraries and languages utilized in this project were thoroughly discussed in other sections.

At the end, a Spark cluster of one master node and three slaves were chosen to perform computation. The master node was $n1 - standard - 2$, and each slave node was $n1 - highmem - 2$, resulting total of 8 cores and 46.5 GB of memory. The overall computation took 18 minutes. The overall size of cleaned data was approximately 18 GB, and the size of final CSV file was 154 MB. The amount of cleaned data forced this project to be done in a Spark cluster, while applying machine learning models was done in local machine. After all development and computation, total cost was less than $5.

Result

The first question for this final project was how accurate are the mathematical models. In the computation section, only linear regression code was shown, but decision tree and neural network modeling were also applied to see how models differ in terms of result.

Figure 8 shows the graph of actual Bitcoin price for past year. The data prior to July 2017 are truncated for clarity of the graph. Figure 8 is the result of all computation done in Scala. The spikes within the graph indicate that the data obtained from Bitcoinchart and Cryptodataset were indeed every transaction happened in each platform. The graph is not normalized; therefore, the spikes are present.

Figure 8: Actual Bitcoin price graph, aggregated

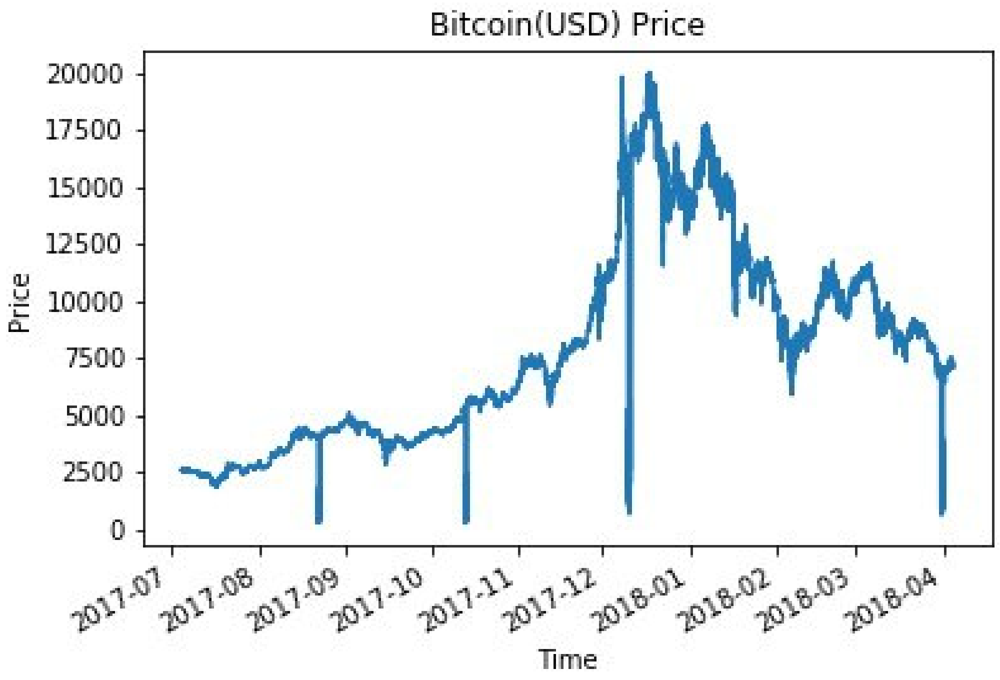


Figure 9 shows the graph of linear regression model. The result is normalized. Figure 10 shows the accuracy of the linear regression model.

Figure 9: Bitcoin Price using Linear Regression

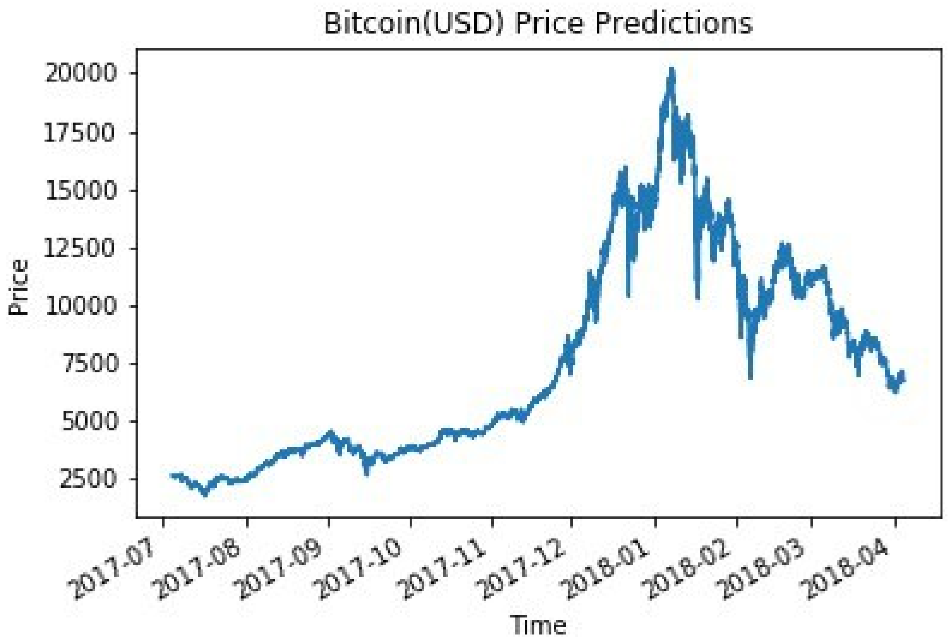
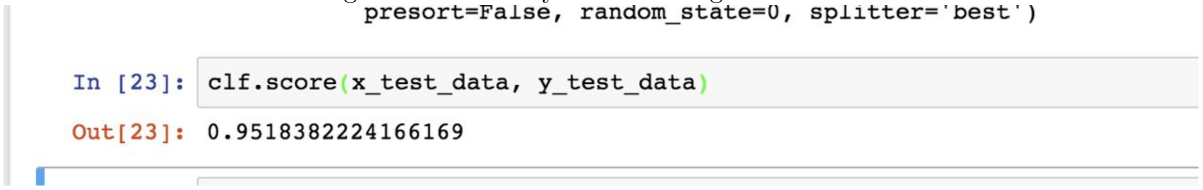


Figure 10: Accuracy of Linear Regression Model



```
            presort=False, random_state=0, splitter='best')

In [23]: clf.score(x_test_data, y_test_data)

Out[23]: 0.9518382224166169
```

0.95 is much higher than anticipated accuracy, and reasons for this number were investigated. One reason might be the trend of market cap data are too highly correlated to the price of Bitcoin, hence having the market cap as an input feature might have forced the accuracy to be high. However, one fact to note is that the market cap data is not minutely data. The dataset is 5 minute interval duplicated 5 times to mimic as if it is a minutely retrieved data. The conclusion is that the price after 5 minutes can be predicted with 95% accuracy using the linear regression model.

Figure 11 shows the result of decision tree model. Figure 12 shows the accuracy of the model.

Figure 11: Bitcoin Price using Decision Tree Model

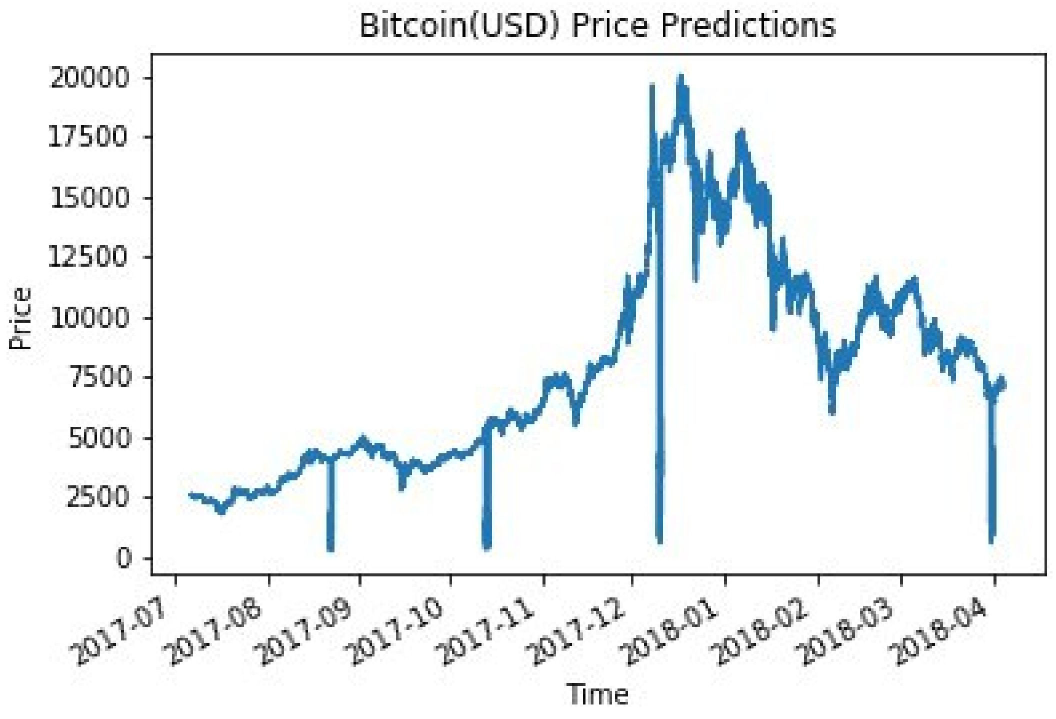
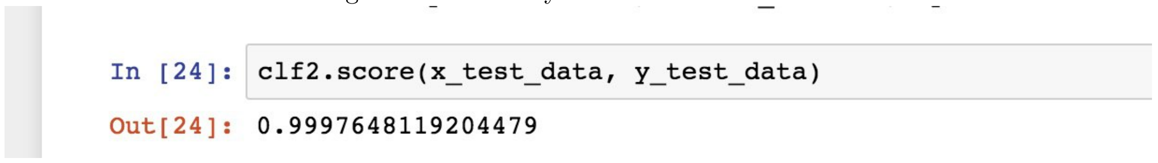


Figure 12: Accuracy of Decision Tree Model



```
In [24]: clf2.score(x_test_data, y_test_data)

Out[24]: 0.9997648119204479
```

Decision tree model even picked up the outliers within the price data, and has resulted in an overfit. For prediction purpose, decision tree model is not a worse candidate than the regression model.

Figure 13 shows the result of neural network model. Figure 14 shows the accuracy of the model.

Figure 13: Bitcoin Price using Decision Tree Model

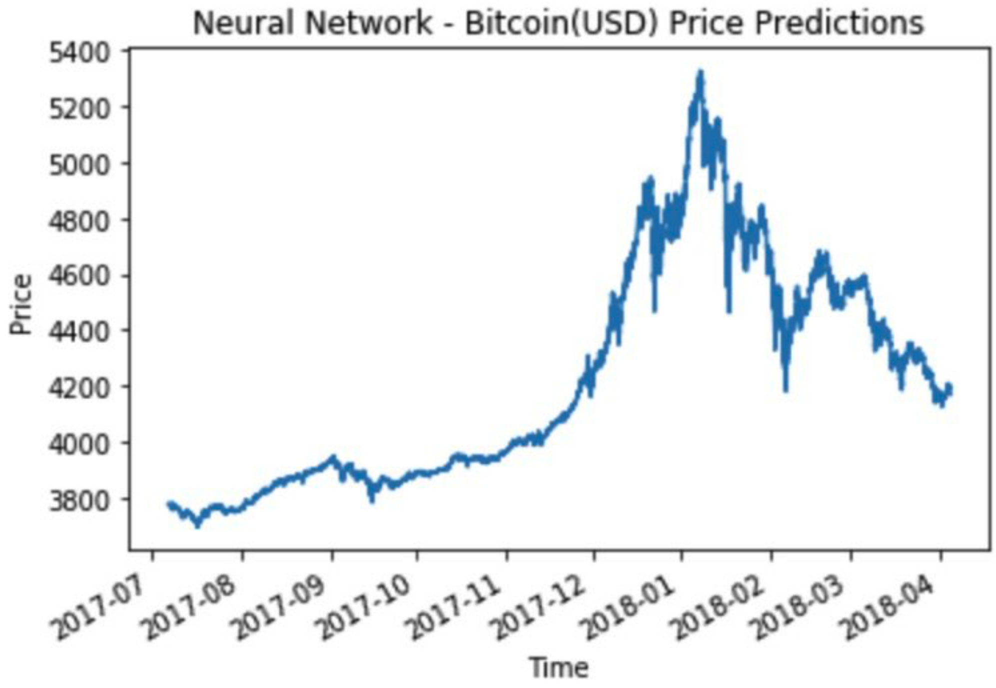
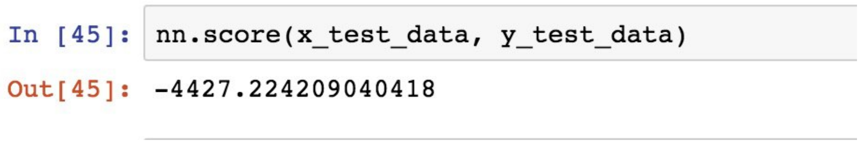


Figure 14: Accuracy of Decision Tree Model



```
In [45]: nn.score(x_test_data, y_test_data)

Out[45]: -4427.224209040418
```

The graphs look similar, but there is a huge difference in y-axis, where neural network model predicted the peak value to be 5.4k, on the other hand the actual price was nearing 20k. Neural network is not suitable to be used in price predicting scenario.

The purpose of trying multiple machine learning models was to see how each model differ when given same datasets. The purpose of this project was not to accurately predict and conclude which model is the best for price prediction, but to preprocess large data in a cluster and to play around with the output using machine learning model to output something interesting. Based on the result and computation, the goal was achieved.

The second objective of this project was to see how accurately can the models predict the future price value. To make the result interesting, a day was chosen where there was a sudden spike in actual Bitcoin price. The day was April 12th, 2018, where from 10:30 AM to 11:30 AM, in UTC time, the price jumped from 6.8k to 7.8k. The inputs for the regression and decision tree model were 1523550697 for timestamp, 43% for dominance, and 297 Billion for market cap, which were the values 5 minutes before the spike. The actual price at 1523550697 was $7,597.78. The regression model predicted the price to be around $7,477.18, while the decision tree predicted $7,345.48. The error rate was 1.44%, which was surprisingly low. The aggregated dataset only had data points until first week of April, when the price was around 6.6k. In conclusion, the prediction model was able to predict surprisingly close for one time point.