Time Series Models and Object Clustering

Oamar Kanji & Ian MacKay

April 23, 2018

Contents

1 Introduction

The first section of this report will seek to show that the initialization step in k-means clustering is non-trivial, as the initial positions chosen for cluster means can significantly affect the final classification of the data set. The second section of this report will explore the use of ARIMA models in the analysis of Bitcoin prices, and some other methods of time series analysis.

2 Clustering

Clustering is the act of separating data into discrete groups to help analysis and prediction. It is also used for several compression and quantization algorithms, which are very important for image files. The resultant groups in k means clustering is highly dependent upon the method of centroid initialization.

2.1 k-means

In its simplest form, k-means assigns initial cluster means randomly. k data points are sampled from the data set and assigned as the initial positions for the centroids, thus sampling from a uniform distribution where every datapoint has an equal chance of being chosen.

2.2 k++means

Proposed in 2007 by David Arthur and Sergei Vassilvitskii[1], *k++*-means clustering seeks to reduce the inconsistency and increase the accuracy of the k-means algorithm with a more sophisticated initialization. The general idea of this algorithm is to initialize centroids far away from each other such that multiple centroids are unlikely to find themselves within a single cluster that would lead to a local optimum.

Unlike regular k-means where initial centroids are sampled from a uniform distribution, *k++* initializes centroids by sampling from a weighted distribution where the probability of a data point being chosen is proportionate to its distance from all other centroids.

2.3 Initialization

The sum of squared error (henceforth referred to as SSE) can be calculated as follows:
$SSE = \sum\limits_{i=1}^{k} \sum\limits_{p \in C_i} (p - m_i)^2$

- (k) number of clusters
- (C) set of objects in cluster
- (m) center point (mean) of cluster

Let us assume that for a particular value of k there exists ***k* final centroids** that will result in the data set being clustered in a way that will minimize the SSE. Let us refer to these k clusters as the ***k* best fitting clusters** as they give a SSE that is small for that particular value of k. An example of this best case classification is shown in Figure 2.4.1.

Sometimes, one or more centroids find what what is called a *local optimum*. This is usually brought about by centroids that are initialized close to each other, which influences the final classification of data points to an extent that the SSE is large, as seen in Figure 2.4.2. In these examples, two initialized centroids do not find their way to the best fitting clusters. Instead, the two centroids remain close to each other, partitioning what should be a single cluster into two. As these leaves one ideal cluster out, it forces one of the remaining centroids to group two clusters together, resulting in an enlarged SSE.

To observe the spread of results returned by k-means clustering, the algorithm was repeated 100 times on the data set and the resultant centroids were plotted on top. This is shown in Figure 2.4.3, and the histogram of SSE is shown in Figure 2.4.4. It can be observed that the areas of highest densities of plotted centroids lie over the four best clusters in the dataset. This may indicate that k-means achieves finding the four best clusters on most runs, but there is a noticeable amount of dispersion of centroids which shows that the algorithm is inconsistent. This is a direct consequence of the method of initialization used in plain k-means clustering.

2.4 Image Segmentation Experiments

3 Time Series Models

Time Series Models are used for a number of analytical and predictive purposes, such as modeling fluctuating inventory levels, commodity prices, and stock prices. They are highly useful for modeling real world phenomena, given that most are governed by time. [2]

3.1 Box-Jenkins Methodology

A time series can contain any of the following components:

- Trend - Long-term movement, similar to y=mx+b
- Seasonality - Fixed periodic fluctuation in y, due to calendar
- Cyclic - Periodic fluctuation in y, due to other influences
- Random - Noise + hidden influences

A summary of the Box-Jenkins Methodology is this:

1. Preprocess data and choose a model based on the components
    - Account for any trends or periodicity in the data.
    - Determine a suitable model from remaining data.
2. Estimate the model parameters
3. Assess the model and return to step one if necessary

3.2 ARIMA Models

ARIMA Models are a combination of Autoregression, Integration, and Moving Average models. Autoregression refers to how the resultant variable is dependent on its prior values, Moving Average refers to how the resultant variable is dependent on prior error terms, and Integration refers to the level of differencing on the dataset. They are denoted by *ARIMA(p,d,q)*, where p=autoregression factor, d=level of integration, and q=moving average factor. There is also a SARIMA (Seasonal Arima) model indicated by *ARIMA(p,d,q)(P,D,Q)s*, where the upper case variables are the same as previous, except for s, which indicates the time lag involved in seasonality.

3.3 ARIMA Variable Selection

Selection of ARIMA model variables can be difficult depending on the dataset. The first step is to remove any trends from the data by differencing, which is shown on Bitcoin data in Figure 3.3.1. The second step is to then analyze the Autocorrelation Factor (ACF) plot and Partial-ACF plot of the data, as shown in Table 3.3.1 and Figures 3.3.2-3.3.3.

3.4 Price Prediction Experiments

Bitcoin is a highly volatile cryptocurrency that is traded constantly worldwide. Thus, it presents an interesting, large, and highly detailed dataset that is perfect for experimenting with time series models on. The log of Volume Weighted Average Price over the period (01/01/13-04/01/17) can be seen in figure 3.4.1.

4 Conclusion

5 Figures & Tables

Figure 2.4.1:

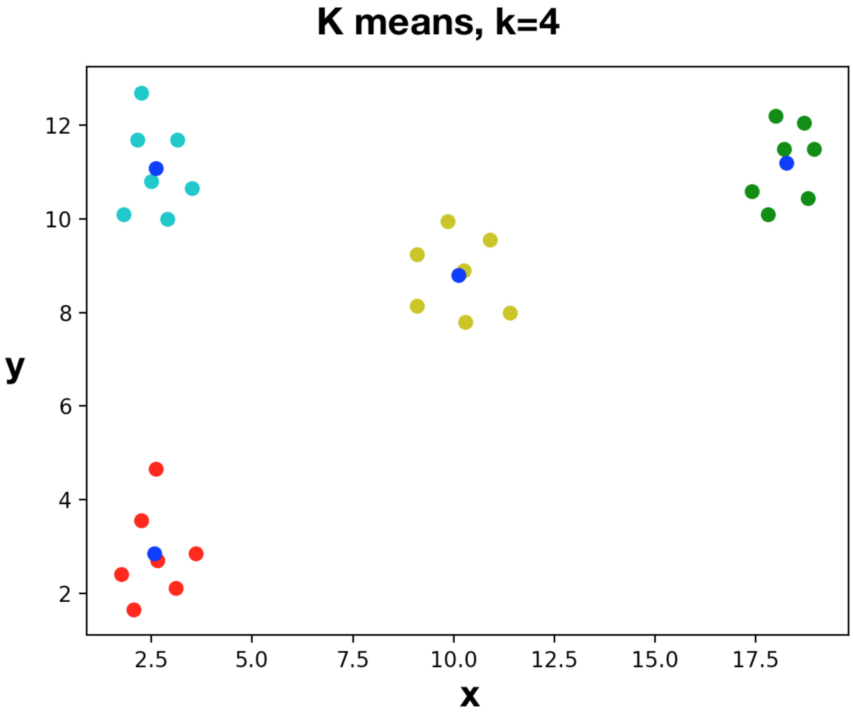


Figure 2.4.2:

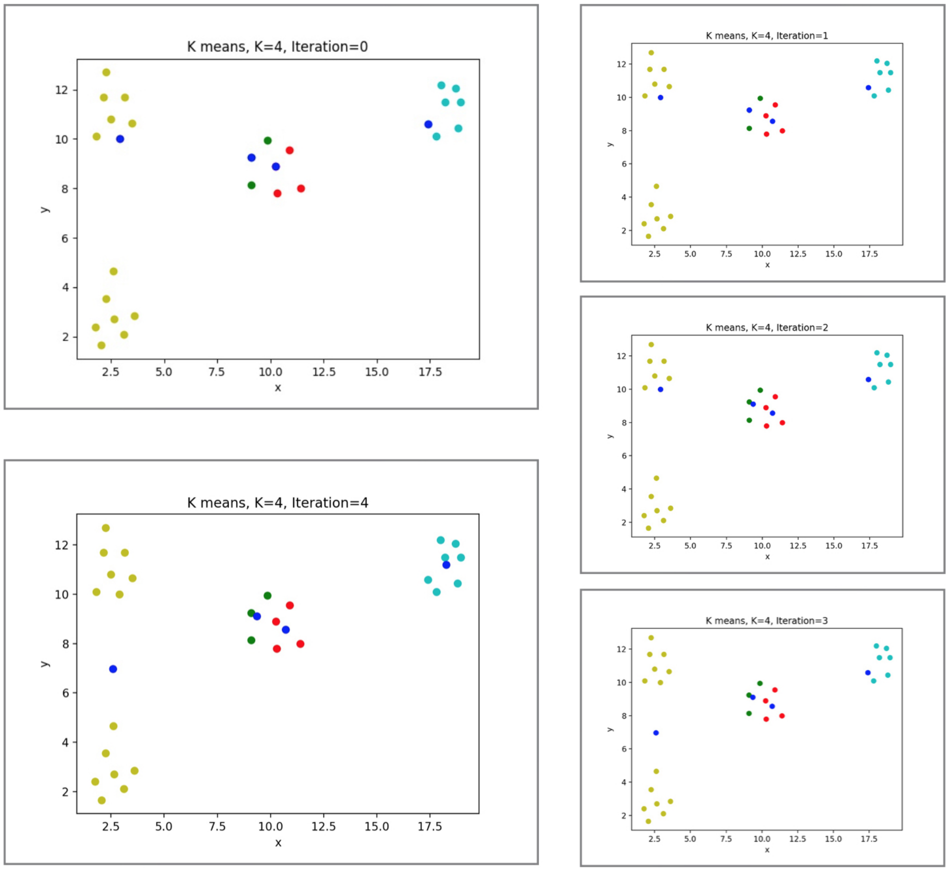


Figure 2.4.3:

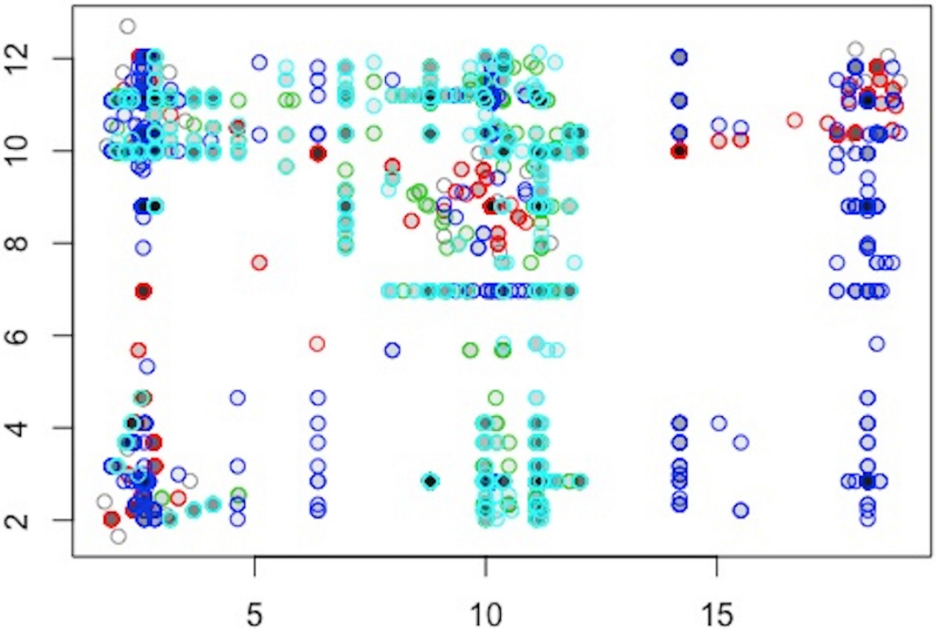


Figure 2.4.4:

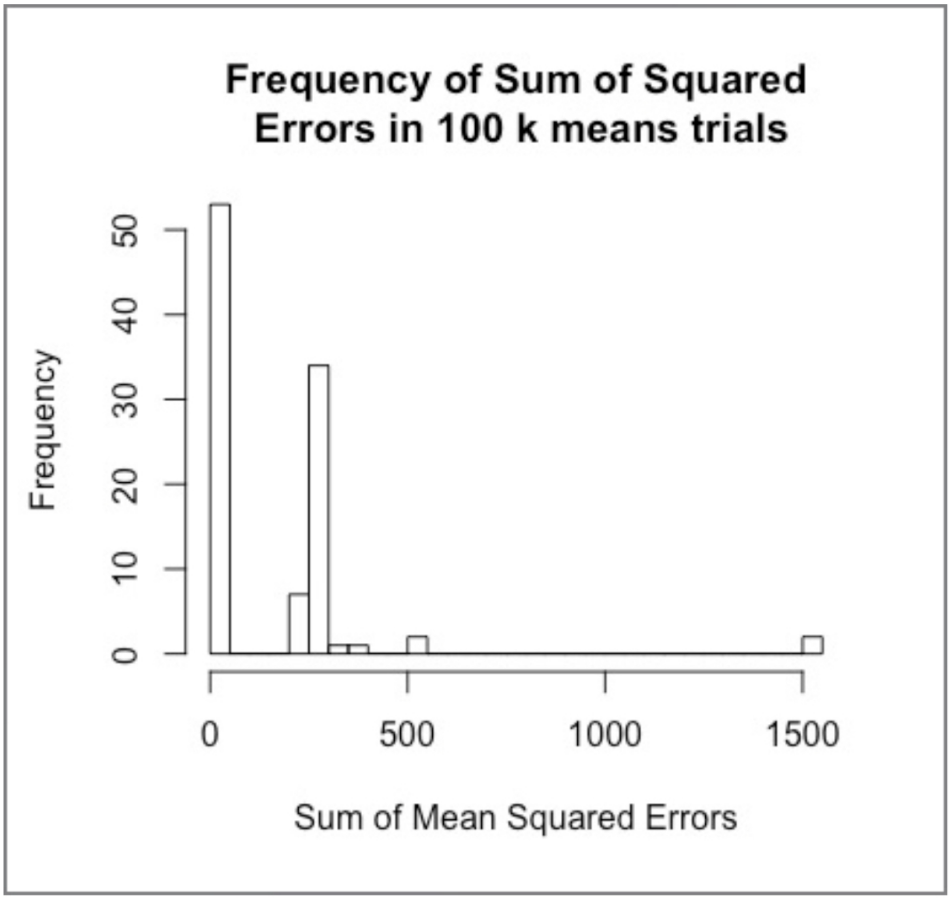


Figure 3.3.1:

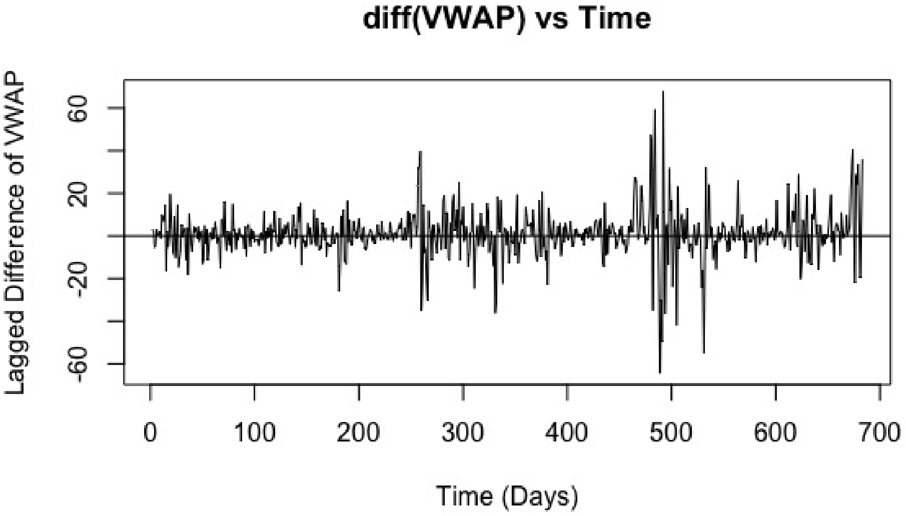


Figure 3.3.2:

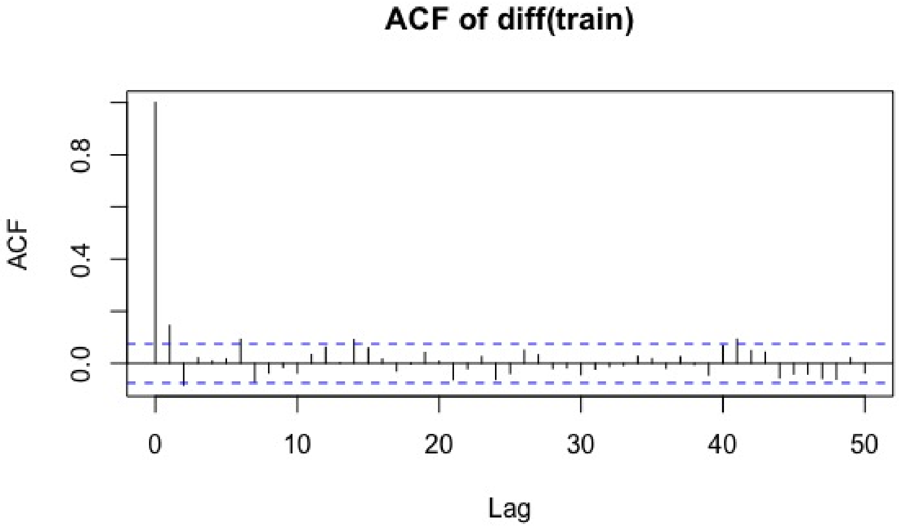


Figure 3.3.3:

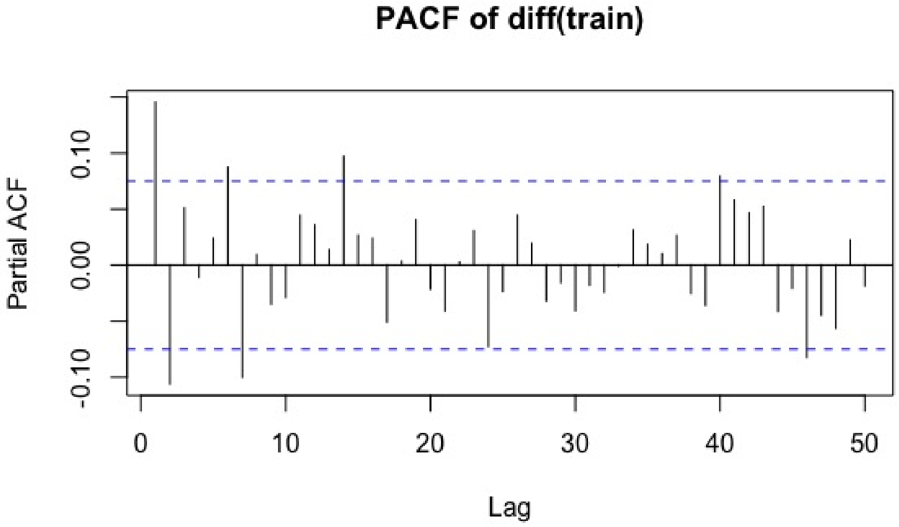


Figure 3.4.1:

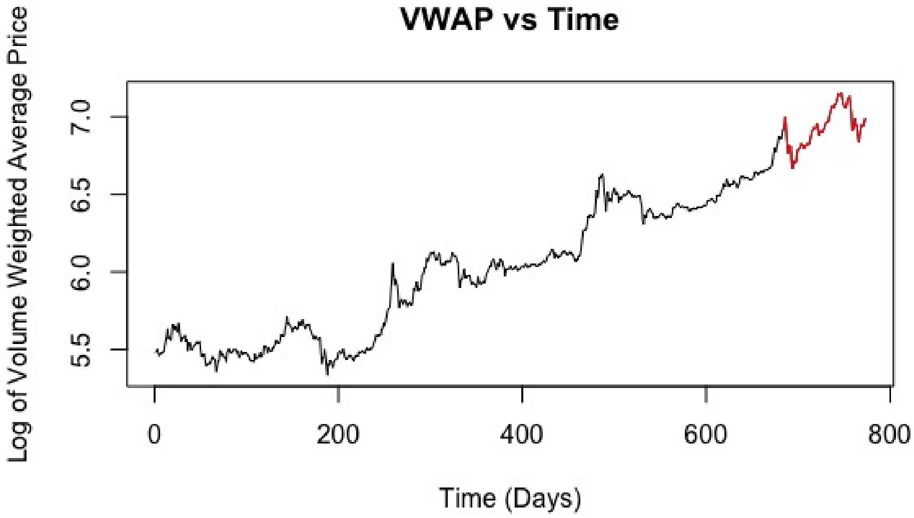


Figure 3.4.2:

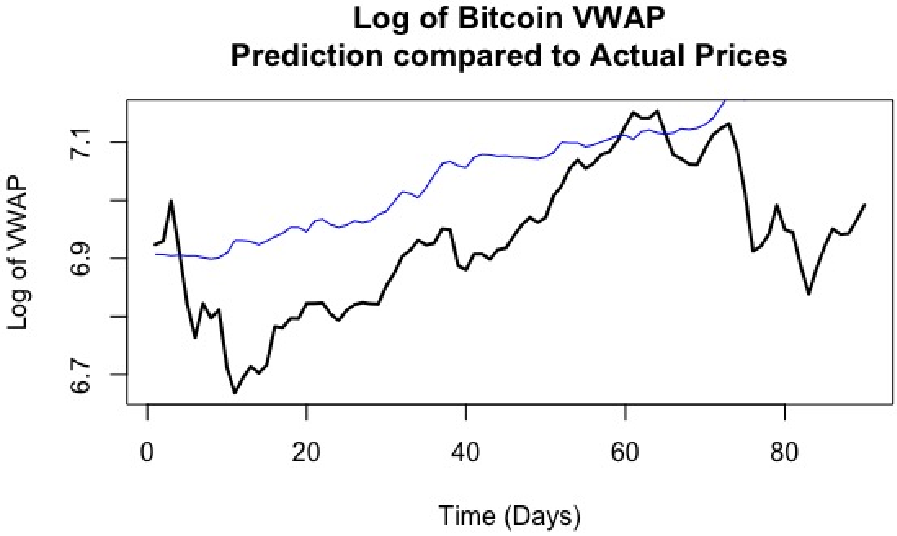
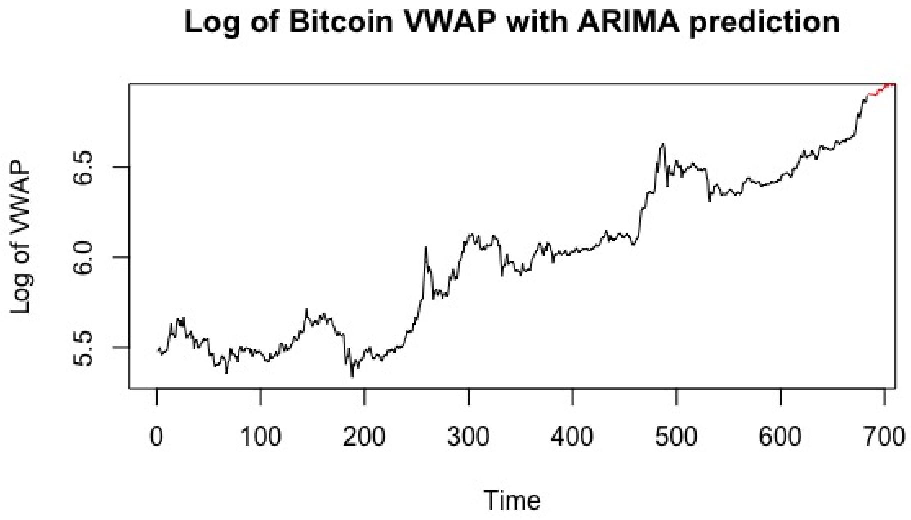


Figure 3.4.3:



6 Appendix

6.1 Bitcoin - Time Series Linear Regression

```
require(jsonlite)
require(plotly)

##############################################
# get.polo.url()
#
#   Creates a poloniex api call url
#   Either for training (train=T) or testing (train=F)
#   Testing end is 90 days past training end
#
# @param start, default=1
#   The start of your training dataset
# @param end, default=1
#   The end of your training dataset
# @param pair, default="USDT_BTC"
#   Token pairs, taken from the market list on https://poloniex.com/
# @param period, default=86400
#   Period of dataset in seconds, default=1 day
# @param train, default=TRUE
#   Whether you want the training or testing url
# @returns string
#   API URL for accessing poloniex data
##############################################
```

```
get.polo.url <- function(start=1, end=1, pair="USDT_BTC", period=86400, train=TRUE) {
  # Returns OHLC + Volume
  POLO_URL <- "https://poloniex.com/public?command=returnChartData&currencyPair=%s&start
  # Returns tick by tick trades (max 50000)
  # POLO_URL2 <- "https://poloniex.com/public?command=returnTradeHistory&currencyPair=

  START <- as.numeric(as.POSIXct(sprintf("2013-%d-01", start)), tz="GMT")
  END <- as.numeric(as.POSIXct(sprintf("2017-%d-01", end)), tz="GMT")
  PRED_START <- as.numeric(as.POSIXct(sprintf("2017-%d-01", end)), tz="GMT")
  PRED_END <- as.numeric(as.POSIXct(sprintf("2017-%d-01", end)), tz="GMT")+7776000
  # PERIOD_VALUES <- c(300, 7200, 86400)
  # 5min, 2hours, 24hours
  PREDICTION_LENGTH <- 90*86400/PERIOD # days
  if(train) {
    return(sprintf(POLO_URL, pair, START, END, period))
  }
  else {
    return(sprintf(POLO_URL, pair, PRED_START, PRED_END, period))
  }
}

# --------------------------------
# Data Creation and Visualization
# --------------------------------
df <- fromJSON(get.polo.url(start=1, end=1))
pred_df <- fromJSON(get.polo.url(start=1,end=1,train=F))
train <- ts(df[8])
test <- ts(pred_df[8])

plot(log(c(train, test)), type="l", xlab="Time (Days)", ylab="Log of Volume Weighted Ave
lines(x=seq(length(train)+1, length(train)+length(test)), y=log(test), col="red")

# -----------
# Differences
# -----------
plot(diff(train), ylab="Lagged Difference of VWAP", xlab="Time (Days)", main="diff(VWAP)
abline(h=0)

# --------------
# ACFs and PACFs
# --------------
acf(diff(train), lag.max=50, main="ACF of diff(train)")
pacf(diff(train), lag.max=50, main="PACF of diff(train)")

# --------------------
# Model and Prediction
# ---------------------
ARIMA <- arima(train, order=c(1,1,0), seasonal=list(order=c(1,1,0), period=41))
ARIMA.pred <- predict(ARIMA, n.ahead=PREDICTION_LENGTH)
error <- abs((c(test)-ARIMA.pred$pred))
error <- ((error/max(error))*(min(log(test))/max(log(test))))+min(log(test))
plot(log(test), lwd=2, xlab="Time (Days)", ylab="Log of VWAP", main="Log of Bitcoin VWAP
lines(x=seq(along=test), log(ARIMA.pred$pred), col="blue", lwd=1)
plot(log(ts(df2[8])), type='l', main="Log of Bitcoin VWAP with ARIMA prediction", ylab="
lines(x=seq(length(series)+1, length(series)+length(ARIMA.pred$pred)), y=log(ARIMA.pred$

# long
#errors <- array(dim=c(4,30))
#for(i in 1:4) {
#  df <- fromJSON(get.polo.url(start=i,end=i))
#  pred_df <- fromJSON(get.polo.url(start=i,end=i,train=F))
#  train <- ts(df[8])
#  test <- ts(pred_df[8])
#  for(j in 9:38) {
#    ARIMA <- arima(train, order=c(1,1,0), seasonal=list(order=c(1,1,1), period=j))
#    ARIMA.pred <- predict(ARIMA, n.ahead=PREDICTION_LENGTH)
#    error <- sum(abs(c(test)-ARIMA.pred£pred))
#    errors[i,(j-21)] <- error
#  }
#}

# install.packages('rnn')
#library(rnn)
#TRAIN_TIME = df[1]
#TRAIN_VOLUME = df[6]
#TRAIN_PRICE = df[4]
#TEST_TIME = pred_df[1]
#TEST_VOLUME = pred_df[6]
#TEST_PRICE = pred_df[4]
\end {minted}

% ------------
% Python Codes
% ------------
\subsection{\textit{k} and \textit{k}++ means clustering}
\begin{minted}{python}
import numpy as np
import scipy
import matplotlib.pyplot as plt
```

```
class Kmeans(object):
    def __init__(self, data_as_txt, k, seed=None):
        self.k = k
        self.data_as_txt = data_as_txt
        self.seed = seed

    def load_dataset(self, name):
        return np.loadtxt(name)

    def euclidian(slef, a, b):
        return np.linalg.norm(a-b)  # magnitude between two points

    def plot(self, dataset, history_centroids, belongs_to):
        colors = ['r', 'g', 'y', 'c']

        fig, ax = plt.subplots()


        for index in range(dataset.shape[0]):
            instances_close = [i for i in range(len(belongs_to)) if belongs_to[i] == ind
            for instance_index in instances_close:
                ax.plot(dataset[instance_index][0], dataset[instance_index][1], (colors[

        history_points = []
        for index, centroids in enumerate(history_centroids):
            for inner, item in enumerate(centroids):
                if index == 0:
                    history_points.append(ax.plot(item[0], item[1], 'bo')[0])
                else:
                    history_points[inner].set_data(item[0], item[1])
                    plt.pause(0.8)


    ''' num_instances: number of rows in data (i.e how many points)
        dataset: the data
        return: list of k randomly selected data points in a list: [[x1 y1] [x2 y2] ...
    '''
    def initialization(self, dataset):

        if self.seed is not None:
            np.random.seed(self.seed)

        num_instances, num_features = dataset.shape  # 45, 2
        return dataset[np.random.choice(num_instances - 1, self.k, replace=False)]
        #return dataset[np.random.randint(0, num_instances - 1, size = self.k)]

    def kmeans(self, k, epsilon=0, distance='euclidian'):
        history_centroids = []
        if distance == 'euclidian':
            dist_method = self.euclidian
        dataset = self.load_dataset(self.data_as_txt)
        # dataset = dataset[:, 0:dataset.shape[1] - 1]
        num_instances, num_features = dataset.shape  # 45, 2


        #prototypes = dataset[np.random.randint(0, num_instances - 1, size=k)]
        prototypes = self.initialization(dataset)
        history_centroids.append(prototypes)
        prototypes_old = np.zeros(prototypes.shape)
        belongs_to = np.zeros((num_instances, 1))
        norm = dist_method(prototypes, prototypes_old)
        iteration = 0

        while norm > epsilon:
            iteration += 1
            norm = dist_method(prototypes, prototypes_old)
            prototypes_old = prototypes
            for index_instance, instance in enumerate(dataset):
                dist_vec = np.zeros((k, 1))
                for index_prototype, prototype in enumerate(prototypes):
                    dist_vec[index_prototype] = dist_method(prototype,
                                                            instance)

                belongs_to[index_instance, 0] = np.argmin(dist_vec)

            tmp_prototypes = np.zeros((k, num_features))

            for index in range(len(prototypes)):
                instances_close = [i for i in range(len(belongs_to)) if belongs_to[i] ==
                prototype = np.mean(dataset[instances_close], axis=0)
                # prototype = dataset[np.random.randint(0, num_instances, size=1)[0]]

                tmp_prototypes[index, :] = prototype

            prototypes = tmp_prototypes

            history_centroids.append(tmp_prototypes)

        #self.plot(dataset, history_centroids, belongs_to)
        return prototypes, history_centroids, belongs_to

    def execute(self, graph=True):
        dataset = self.load_dataset(self.data_as_txt)
        centroids, history_centroids, belongs_to = self.kmeans(self.k)
        if(graph):
            self.plot(dataset, history_centroids, belongs_to)

        return centroids


class Kpp(Kmeans):

    def initialization(self, dataset):
        X = dataset
        print(X)

        C = [X[0]]
        print(C)
        for k in range(1, self.k):
            #  find shortest distantce squared to center
            D2 = scipy.array([min([scipy.inner(c-x, c-x) for c in C]) for x in X])
            probs = D2/D2.sum()
            cumprobs = probs.cumsum()
            print(cumprobs)
            r = scipy.rand()
            for j, p in enumerate(cumprobs):
                if r < p:
                    i = j
                    break
            C.append(X[i])
        return np.array(C)

if __name__ == "__main__":
    kpp_means = Kpp("pts2.txt", 4)
    #centroids = kpp_means.execute(graph=True)

    kmeans = Kmeans("pts2.txt", 4)
    #centroids = kmeans.execute(graph=True)

    all_kmeans_centroids = []
    all_kpp_means_centroids = []

    for _ in range(1, 1000):
        kmeans_centroids = kmeans.execute(graph=False)
        all_kmeans_centroids.append(kmeans_centroids)

    for _ in range(1, 1000):
        kpp_centroids = kpp_means.execute(graph=False)
        all_kpp_means_centroids.append(kpp_centroids)

    with open("KmeansCentroids.txt", mode='w') as File:
        for i in all_kmeans_centroids:
            for j in i:
                File.write("{} {} ".format(j[0], j[1]))
            File.write("\n")

    with open("KppCentroids.txt", mode='w') as File:
        for i in all_kpp_means_centroids:
            for j in i:
                File.write("{} {} ".format(j[0], j[1]))
            File.write("\n")
```

6.2 Centroid Plots and Distribution of SSE

```
# --------------
# Plot centroids
# --------------
points <- read.table("pts2.txt")
kmeansCentroids <- read.csv("KmeansCentroids.txt", sep=" ", header=FALSE, na.strings="na
kppCentroids <- read.csv("KppCentroids.txt", sep=" ", header=FALSE, na.strings="nan")[1:

plot(points, col=rgb(0,0,0,0.5))
for(i in 1:4) {
  points(x=kmeansCentroids[,1*i], y=kmeansCentroids[,2*i], pch=21, col=i+1, bg=rgb(0.1,0
}
plot(points, col=rgb(0,0,0,0.5))
for(i in 1:4) {
  points(x=kppCentroids[,1*i], y=kppCentroids[,2*i], pch=25, col=i+1, bg=rgb(0.1,0.1,0.1

# ------------------------
# Show distribution of SSE
# ------------------------
x <- read.csv("kmeans_sse.csv", sep=" ", header=FALSE)
x <- unlist(x, use.names=FALSE)
hist(x, main="Frequency of Sum of Mean Squared\n Errors in 100 k means trials", xlab="Su
mu <- mean(x)
variance <- var(x)
print(mu)
print(variance)
print(IQR(x))

x <- read.csv("kpp_sse.csv", sep=" ", header=FALSE)
x <- unlist(x, use.names=FALSE)
hist(x, main="Frequency of Sum of Mean Squared\n Errors in 100 k++ trials", xlab="Sum of
mu <- mean(x)
variance <- var(x)
print(mu)
print(variance)
print(IQR(x))
```

https://github.com/immackay/4990-Final-ProjectGitHub repository for code

References

[1] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[2] Unknown. Time series - wikipedia.