Crypt Predict

Technical Manual

BSc (Hons) in Software Development

Name: Adam Eaton

Student ID: C00179859

Year: 4th Year

Supervisor: Lei Shi

Due-Date: 18 - 04 - 2018

Table of Contents

1. Accuracy_Methods.py

```python
# -*- coding: utf-8 -*-
"""
@author: Adam Eaton

Stores Functions used to add entries to the accuracy table, as well as a
Function to calculate the overall percentage accuracy of a given Cryptocurrency

"""
import Slack_Notify as SN

import csv
import os
import pandas as pd

def add_entries(crypto, act_vals, pred_vals):
    index = 0

    while index < len(pred_vals):
        if act_vals[index] == pred_vals[index]:
            acc = 0.0

        try:
            x = pred_vals[index] - act_vals[index]
            y = (pred_vals[index] + act_vals[index]) / 2
            acc = round((x / y) * 1000, 2)

            values = [crypto, act_vals[index], pred_vals[index], x, acc]

            with open(os.path.join('data',r'accuracy_data.csv'), 'a', newline='') as acc_csv:
                writer = csv.writer(acc_csv)
                writer.writerow(values)

            index += 1

        except:
            SN.send_notification("Accuracy_Methods.py - add_entries() -- ")
            pass


def calculate_accuracy(crypto):
    try:
        acc_data = pd.read_csv(os.path.join('data', 'accuracy_data.csv'))
        predicted_vals = []
        accuracy_vals = []

        for index, row in acc_data.iterrows():
            if row['crypto'] == crypto:
                predicted_vals.append(row['predicted'])
                accuracy_vals.append(row['accuracy'])

        avg_pred = sum(predicted_vals) / len(predicted_vals)
        avg_acc = sum(accuracy_vals) / len(accuracy_vals)

        x = avg_pred - avg_acc
        y = (avg_pred + avg_acc) / 2
        z = abs((x / y) * 10)
        overall_acc = round(100 - z, 2)

        return overall_acc

    except:
        SN.send_notification("Accuracy_Methods.py - calculate_accuracy() -- ")
        pass
```

2. app.py

```python
# -*- coding: utf-8 -*-
"""
@author: Adam Eaton

Main Flask Application for Crypt-Predict. Contains the different possible routes
and their associated function calls.

"""

from flask import Flask, render_template, request, Markup

import Generation_Methods as GM
import Data_Check as DC
import Accuracy_Methods as AM

app = Flask(__name__)


@app.route('/')
@app.route('/index')
def return_index():
    return render_template('index.html',
                           btc_acc = AM.calculate_accuracy("btc"),
                           eth_acc = AM.calculate_accuracy("eth"),
                           ltc_acc = AM.calculate_accuracy("ltc"))


@app.route('/about')
def return_about():
    return render_template('about.html')


@app.route('/submit', methods=['GET', 'POST'])
def submit_form():
    btc = request.form.get('btc-check')
    eth = request.form.get('eth-check')
    ltc = request.form.get('ltc-check')
    method = str(request.form.get('method'))
    time_interval = int(request.form.get('interval'))
    fill_missing = False

    btc_graph = False
    eth_graph = False
    ltc_graph = False

    if method == "predict":
        if btc == "on":
            data = GM.select_data("btc", time_interval)
            data = DC.run_check(data, time_interval, fill_missing)
            btc_pred = GM.generate_prediction(data)
            btc_time = GM.generate_dates(btc_pred, time_interval)
            btc_graph = GM.generate_graph(method, None, btc_pred, btc_time)
            btc_graph = Markup(btc_graph)

        if eth == "on":
            data = GM.select_data("eth", time_interval)
            data = DC.run_check(data, time_interval, fill_missing)
            eth_pred = GM.generate_prediction(data)
            eth_time = GM.generate_dates(eth_pred, time_interval)
            eth_graph = GM.generate_graph(method, None, eth_pred, eth_time)
            eth_graph = Markup(eth_graph)

        if ltc == "on":
            data = GM.select_data("ltc", time_interval)
            data = DC.run_check(data, time_interval, fill_missing)
            ltc_pred = GM.generate_prediction(data)
            ltc_time = GM.generate_dates(ltc_pred, time_interval)
            ltc_graph = GM.generate_graph(method, None, ltc_pred, ltc_time)
            ltc_graph = Markup(ltc_graph)

        return render_template('results.html',
                               btc_div = btc_graph,
                               eth_div = eth_graph,
                               ltc_div = ltc_graph)

    elif method == "test":
        if btc == "on":
            data = GM.select_data("btc", time_interval)
            data = DC.run_check(data, time_interval, fill_missing)
            btc_actual, btc_pred = GM.generate_test(data)
            AM.add_entries("btc", btc_actual, btc_pred)
            btc_time = GM.generate_dates(btc_pred, time_interval)
            btc_graph = GM.generate_graph(method, btc_actual, btc_pred, btc_time)
            btc_graph = Markup(btc_graph)

        if eth == "on":
            data = GM.select_data("eth", time_interval)
            data = DC.run_check(data, time_interval, fill_missing)
            eth_actual, eth_pred = GM.generate_test(data)
            AM.add_entries("eth", eth_actual, eth_pred)
            eth_time = GM.generate_dates(eth_pred, time_interval)
            eth_graph = GM.generate_graph(method, eth_actual, eth_pred, eth_time)
            eth_graph = Markup(eth_graph)

        if ltc == "on":
            data = GM.select_data("ltc", time_interval)
            data = DC.run_check(data, time_interval, fill_missing)
            ltc_actual, ltc_pred = GM.generate_test(data)
            AM.add_entries("ltc", ltc_actual, ltc_pred)
            ltc_time = GM.generate_dates(ltc_pred, time_interval)
            ltc_graph = GM.generate_graph(method, ltc_actual, ltc_pred, ltc_time)
            ltc_graph = Markup(ltc_graph)

        return render_template('results.html',
                               btc_div = btc_graph,
                               eth_div = eth_graph,
                               ltc_div = ltc_graph)


if __name__ == '__main__':
    app.run(debug = True)
```

3. Bayesian_Methods.py

```python
# -*- coding: utf-8 -*-
"""
@author: Adam Eaton

Stores the functions used to split data, creating dataframes as well as
make the mathematical calculations involved in the algorithm.

"""
import Slack_Notify as SN

import pandas as pd
import numpy as np
import math

def split_prices(df):
    prices_1 = []
    prices_2 = []
    prices_3 = []

    dfVal = df["price"].values
    count = 0

    try:
        for x in dfVal:
            if(count < 9001):
                prices_1.append(x)
            if(count > 9000 and count <= 18001):
                prices_2.append(x)
            if(count > 18000 and count < 27002):
                prices_3.append(x)
            count = count+1

    except:
        SN.send_notification("Bayesian_Methods.py - split_prices() -- ")
        pass

    return prices_1, prices_2, prices_3


def create_dataframe(l):
    index_list= []
    column_list = []

    for x in range(50):
        index_list.append(x+1)

    for x in range(l):
        column_list.append("priceDiff"+str(x+1))
    column_list.append("Var")

    df = pd.DataFrame(index=index_list, columns=column_list)
    df = df.fillna(0.0)

    return df


def populate_dataframe(data_frame, data_source):
    col_num = len(data_frame.columns)
    value_count = 1

    if(col_num == 46):
        value_count = 181
    if(col_num == 91):
        value_count = 91
    for index, row in data_frame.iterrows():
        row_contents = []
        counter = 1
        while(counter <= col_num):
            if(counter == col_num):
                row['Var'] = np.mean(row_contents)
                break

            price_diff = data_source[value_count] - data_source[value_count-1]


            row["priceDiff"+str(counter)] = price_diff

            row_contents.append(price_diff)
            value_count = value_count+1
            counter = counter+1

    return col_num


def calculate_similarity(x, y):
    similarity = 0
    mean_1 = np.mean(x)
    mean_2 = np.mean(y)
    sum_m = 0

    for i in range(0, len(x)):
        m_x = (x[i] - mean_1)
        m_y = (y[i] - mean_2)
        sum_m += m_x * m_y

    m_std = float(len(x) * np.std(x) * np.std(y))

    try:
        similarity = float(sum_m) / float(m_std)
    except ZeroDivisionError:
        similarity = 0.1

    return similarity


def calculate_delta(row, df, weight):
    x0 = 0
    x1 = 0

    for i in range(0,len(df)):
        y = df.iloc[i][-1]
        z = df.iloc[i][:-1]
        sli = row[:-1]

        similarity = calculate_similarity(sli, z)

        x1 += y * math.exp(similarity * weight)
        x0 += math.exp(similarity * weight)

    return float(x1 / x0)


def set_delta(weight, train_1, train_2):
    t_delta = np.empty(0)
    index = len(train_2.index)

    for x in range(0, index):
        delta = calculate_delta(train_1.iloc[x], train_2, weight)
        t_delta = np.append(t_delta, delta)

    return t_delta
```

```python
def combine_data(d1, d2, d3, d4):
    data = {'delta': d1, 'delta_45': d2,
            'delta_90': d3, 'delta_180': d4}

    data_df = pd.DataFrame(data)

    return data_df
```

4. Data_Check.py

```
# -*- coding: utf-8 -*-
"""
@author: Adam Eaton

Contains the fucntions used to perform the Data Check on the selected Data
whether that be to drop rows with missing data or to interpolate it based on
previous data.

"""
import Slack_Notify as SN

import numpy as np
import pandas as pd

# Interpolate np.NaN values within the supplied dataframe
def interp_values(data):
    interp_data = data.interpolate(method='values')
    return interp_data


# Check for missing values based on the epoch values in the 'date' column
def fill_data(data, freq):
    data_len = len(data.index)
    count = 1
    missing_count = 0

    # For each element in DataFrame, check its date against the predecessor
    # If difference is greater than freq, insert new row with appropriate time
    # and initialise NaN values for other columns
    while(count < data_len):
        curr_row = data.iloc[count]['date']
        prev_row = data.iloc[count-1]['date']
        diff = curr_row - prev_row

        # Fault tolerant up to freq seconds
        # if the difference in values is greater than freq seconds then mark is as missing
        # otherwise ignore it and move on
        if(int(diff-freq) > freq):
            missing_count = missing_count+1
            missing_vals = int(diff/freq)

            while(missing_vals >= 0):
                line = pd.DataFrame({"date": prev_row+freq, "price": np.NaN,
                                     "vAsk": np.NaN, "vBid": np.NaN}, index=[count])
                data = pd.concat([data.ix[:count-1], line, data.ix[count:]]).reset_index(drop=True)
                missing_vals = missing_vals - 1

        count = count + 1

    if(missing_count != 0):
        data = interp_values(data)

    return data


# Drops any rows containing a NaN value
def drop_data(data):
    drop = data.dropna(axis=1, how='any')
    return drop


def run_check(data, freq, fill):
    if fill == True:
        data = fill_data(data, freq)
    elif fill == False:
        data = drop_data(data)

    else:
        SN.send_notification("Data_Check.py - run_check() -- ")
        pass
    return data
```

5. Data_Collection.py

```python
# -*- coding: utf-8 -*-
"""
@author: Adam Eaton

Contains functions to retrieve and store data for the three Cryptocurrencies.
Doesn't run as part of the main application but instead runs in it's own instance.
"""
import Slack_Notify as SN

import time
import requests
import csv

BTC_Ticker_Add = "https://www.okcoin.com/api/v1/ticker.do?symbol=btc_usd"
BTC_Depth_Add = "https://www.okcoin.com/api/v1/depth.do?symbol=btc_usd&size=60"

ETH_Ticker_Add = "https://www.okcoin.com/api/v1/ticker.do?symbol=eth_usd"
ETH_Depth_Add = "https://www.okcoin.com/api/v1/depth.do?symbol=eth_usd&size=60"

LTC_Ticker_Add = "https://www.okcoin.com/api/v1/ticker.do?symbol=ltc_usd"
LTC_Depth_Add = "https://www.okcoin.com/api/v1/depth.do?symbol=ltc_usd&size=60"

currency_Add = "https://api.fixer.io/latest?base=USD"


def run_queries():
    btc_data = BTC_query()
    eth_data = ETH_query()
    ltc_data = LTC_query()

    btc_str = str(btc_data)
    eth_str = str(eth_data)
    ltc_str = str(ltc_data)

    if(btc_str[:1] == "[" and eth_str[:1] == "[" and ltc_str[:1] == "[" ):
        with open(r'btc_data.csv', 'a', newline = '') as btc:
            writer = csv.writer(btc)
            writer.writerow(btc_data)

        with open(r'eth_data.csv', 'a', newline = '') as eth:
            writer = csv.writer(eth)
            writer.writerow(eth_data)

        with open(r'ltc_data.csv', 'a', newline = '') as ltc:
            writer = csv.writer(ltc)
            writer.writerow(ltc_data)

        print("Tick")
    else:
        SN.send_notification("Data_Collection.py - run_queries() -- ")
    return

# Possibly look into re-implementing later on, as of now (18/01/18) it's providing innacurate
results
"""
```

```
def currency_query(val_usd):
    currency_response = requests.get(currency_Add).json()
    currency_value = currency_response['rates']['EUR']
    return float(val_usd) * float(currency_value)
"""

def BTC_query():
    BTC_Ticker = requests.get(BTC_Ticker_Add).json()
    BTC_Depth = requests.get(BTC_Depth_Add).json()

    BTC_Price_USD = float(BTC_Ticker['ticker']['last'])
   # BTC_Price_EUR = currency_query(BTC_Price_USD)

    BTC_Date = BTC_Ticker['date']
    BTC_vBid = sum([bid[1] for bid in BTC_Depth['bids']])
    BTC_vAsk = sum([ask[1] for ask in BTC_Depth['asks']])
    values = [BTC_Date, BTC_Price_USD, BTC_vBid, BTC_vAsk]
    return values


def ETH_query():
    ETH_Ticker = requests.get(ETH_Ticker_Add).json()
    ETH_Depth = requests.get(ETH_Depth_Add).json()

    ETH_Price_USD = float(ETH_Ticker['ticker']['last'])
    #ETH_Price_EUR = currency_query(ETH_Price_USD)

    ETH_Date = ETH_Ticker['date']
    ETH_vBid = sum([bid[1] for bid in ETH_Depth['bids']])
    ETH_vAsk = sum([ask[1] for ask in ETH_Depth['asks']])
    values = [ETH_Date, ETH_Price_USD, ETH_vBid, ETH_vAsk]
    return values


def LTC_query():
    LTC_Ticker = requests.get(LTC_Ticker_Add).json()
    LTC_Depth = requests.get(LTC_Depth_Add).json()

    LTC_Price_USD = float(LTC_Ticker['ticker']['last'])
    #LTC_Price_EUR = currency_query(LTC_Price_USD)

    LTC_Date = LTC_Ticker['date']
    LTC_vBid = sum([bid[1] for bid in LTC_Depth['bids']])
    LTC_vAsk = sum([ask[1] for ask in LTC_Depth['asks']])
    values = [LTC_Date, LTC_Price_USD, LTC_vBid, LTC_vAsk]
    return values


def main():
    start_time = time.time()

    while True:
        try:
            run_queries()
            time.sleep(20.0 - ((time.time() - start_time) % 20.0))
        except:
            SN.send_notification("Data_Collection.py - run_queries() -- ")
            pass

if __name__ == '__main__':
    main()
```

6. Generation_Methods.py

```python
# -*- coding: utf-8 -*-
"""
@author: Adam Eaton

Stores the Functions used to perform tasks related to selecting the data,
generating the prediction as well as graphing the end result.
However the actual maths based functions are stored in Bayesian_Method.py

"""
import Bayesian_Methods as BM

from plotly.offline import plot
from plotly.graph_objs import Scatter, Layout, Margin

import statsmodels.formula.api as smf
import datetime
import numpy as np
import pandas as pd
import os


def select_data(coin, interval):
    if coin == "btc":
        dataset = pd.read_csv(os.path.join('data', 'btc_data.csv'))
    elif coin == "eth":
        dataset = pd.read_csv(os.path.join('data', 'eth_data.csv'))
    elif coin == "ltc":
        dataset = pd.read_csv(os.path.join('data', 'ltc_data.csv'))

    if interval == 20:
        data = dataset.tail(27002)

    elif interval == 40:
        data = dataset.tail(54004)
        data = data.iloc[::2, :]

    elif interval == 60:
        data = dataset.tail(81006)
        data = data.iloc[::3, :]

    elif interval == 80:
        data = dataset.tail(108008)
        data = data.iloc[::4, :]

    data = data.reset_index(drop=True)
    return data


def conv_prices(start, data, col):
    values = []
    values.append(start)
    count = 1

    for index, row in data.iterrows():
        val = values[count - 1] + row[col]
        values.append(round(val, 2))
        count = count + 1
    return values


def generate_dates(data, interval):
    index = 0
    dates = []
    ts = datetime.datetime(2018, 1,1, 0,0,0)
    while index < len(data):
        dates.append(datetime.datetime.strftime(ts, '%H:%M:%S'))
        ts += datetime.timedelta(seconds=interval)
        index += 1
    return dates


def generate_prediction(data):
    weight = 2
    prices_1, prices_2, prices_3 = BM.split_prices(data)

    # Initialising series of dataframes
    train_1_45 = BM.create_dataframe(45)
    train_1_90 = BM.create_dataframe(90)
    train_1_180 = BM.create_dataframe(180)

    train_2_45 = BM.create_dataframe(45)
    train_2_90 = BM.create_dataframe(90)
    train_2_180 = BM.create_dataframe(180)

    train_3_45 = BM.create_dataframe(45)
    train_3_90 = BM.create_dataframe(90)
    train_3_180 = BM.create_dataframe(180)


    #Populating the previously created dataframes
    BM.populate_dataframe(train_1_45, prices_1)
    BM.populate_dataframe(train_1_90, prices_1)
    BM.populate_dataframe(train_1_180, prices_1)

    BM.populate_dataframe(train_2_45, prices_2)
    BM.populate_dataframe(train_2_90, prices_2)
    BM.populate_dataframe(train_2_180, prices_2)

    BM.populate_dataframe(train_3_45, prices_3)
    BM.populate_dataframe(train_3_90, prices_3)
    BM.populate_dataframe(train_3_180, prices_3)

    train_delta_45 = BM.set_delta(weight, train_3_45, train_2_45)
    train_delta_90 = BM.set_delta(weight, train_3_90, train_2_90)
    train_delta_180 = BM.set_delta(weight, train_3_180, train_2_180)

    train_1_180_Var = train_2_180[['Var']]
    train_2_180_Var = train_3_180[['Var']]
    tr_delta = []

    for index, row in train_1_180_Var.iterrows():
        tr_delta.append(row['Var'])

    for index, row in train_2_180_Var.iterrows():
        tr_delta[index-1] += row['Var']

    train_delta = np.asarray(tr_delta)
    train_delta = np.reshape(tr_delta, -1)

    # Combine all the training data
    training_data = BM.combine_data(train_delta, train_delta_45,
                              train_delta_90, train_delta_180)

    # Using the delta_X data to train the linear model
    formula_s = 'delta ~ delta_45 + delta_90 + delta_180'
    model = smf.ols(formula = formula_s, data = training_data).fit()

    # Predict price variation on the training data set.
    predict = model.predict(training_data)

    predictions = { 'Predicted Values': predict }
    predictions_DF = pd.DataFrame(predictions)
    # start denotes the point at which the prices should be converted
    start = prices_3[len(prices_3) - 1]
    predicted_vals = conv_prices(start, predictions_DF, 'Predicted Values')

    return predicted_vals


def generate_test(data):
    weight = 2
    prices_1, prices_2, prices_3 = BM.split_prices(data)

    # Initialising series of dataframes
    train_1_45 = BM.create_dataframe(45)
    train_1_90 = BM.create_dataframe(90)
    train_1_180 = BM.create_dataframe(180)

    train_2_45 = BM.create_dataframe(45)
    train_2_90 = BM.create_dataframe(90)
    train_2_180 = BM.create_dataframe(180)

    test_45 = BM.create_dataframe(45)
    test_90 = BM.create_dataframe(90)
    test_180 = BM.create_dataframe(180)


    #Populating the previously created dataframes
    BM.populate_dataframe(train_1_45, prices_1)
    BM.populate_dataframe(train_1_90, prices_1)
    BM.populate_dataframe(train_1_180, prices_1)

    BM.populate_dataframe(train_2_45, prices_2)
    BM.populate_dataframe(train_2_90, prices_2)
    BM.populate_dataframe(train_2_180, prices_2)

    BM.populate_dataframe(test_45, prices_3)
    BM.populate_dataframe(test_90, prices_3)
    BM.populate_dataframe(test_180, prices_3)


    # Populating training sets
    train_delta_45 = BM.set_delta(weight, train_2_45, train_1_45)
    train_delta_90 = BM.set_delta(weight, train_2_90, train_1_90)
    train_delta_180 = BM.set_delta(weight, train_2_180, train_1_180)

    test_delta_45 = BM.set_delta(weight, test_45, train_1_45)
    test_delta_90 = BM.set_delta(weight, test_90, train_1_90)
    test_delta_180 = BM.set_delta(weight, test_180, train_1_180)


    # Calculating train_delta by combining the variance values of both sets
    train_1_180_Var = train_1_180[['Var']]
    train_2_180_Var = train_2_180[['Var']]
    tr_delta = []

    for index, row in train_1_180_Var.iterrows():
        tr_delta.append(row['Var'])

    for index, row in train_2_180_Var.iterrows():
        tr_delta[index-1] += row['Var']

    # Actual delta values for training data
    train_delta = np.asarray(tr_delta)
    train_delta = np.reshape(tr_delta, -1)

    # Actual delta values for test data.
    test_delta = np.asarray(test_180[['Var']])
    test_delta = np.reshape(test_delta, -1)

    # Combine all the training data
    training_data = BM.combine_data(train_delta, train_delta_45,
                              train_delta_90, train_delta_180)


    # Combine all the test data
    testing_data = BM.combine_data(test_delta, test_delta_45,
                                   test_delta_90, test_delta_180)

    # Using the delta_X data to train the linear model
    formula_s = 'delta ~ delta_45 + delta_90 + delta_180'
    model = smf.ols(formula = formula_s, data = training_data).fit()

    predict = model.predict(testing_data)

    comparison = { 'Actual Values': test_delta,
                  'Predicted Values': predict }

    comparison_DF = pd.DataFrame(comparison)
    start = prices_3[len(prices_3) - 1]
    actual_vals = conv_prices(start, comparison_DF, 'Actual Values')
    predicted_vals = conv_prices(start, comparison_DF, 'Predicted Values')

    return actual_vals, predicted_vals



def generate_graph(method, act_data, pred_data, time):
    if method == "predict":
        graph = plot({"data": [Scatter(x=time, y=pred_data)],
                    "layout": Layout(width="800", height="500", margin=Margin(t="20"))},
                    output_type='div')

    elif method == "test":
        graph = plot({"data": [Scatter(showlegend=True, name="Predicted", x=time,
y=pred_data),Scatter(showlegend=True, name="Actual", x=time, y=act_data) ],
                    "layout": Layout(width="800", height="500",
margin=Margin(t="20"),showlegend=True)},
                    output_type='div')
    return graph
```

7. Slack_Notify.py

```python
# -*- coding: utf-8 -*-
"""
@author: Adam Eaton

Used to send notifications to the designated Slack channel regarding errors
that occur during runtime.

"""

from slackclient import SlackClient
import traceback
import sys

slack_token = "#"

def send_notification(msg):
    sc = SlackClient(slack_token)

    stack = traceback.format_exception_only(sys.last_type, sys.last_value)
    error = str(stack[len(stack)-1])
    message = msg + error

    sc.api_call(
            "chat.postMessage",
            channel = "#notifications",
            text = message)

    return
```