University of Malta

Department of Artificial Intelligence

Module Assignment

Financial Engineering (ARI5122)

Dylan Vasallo

dylan.vassallo.18@um.edu.mt

February 15, 2019

Contents

Chapter 1

Assignment – Part 1

The code used to answer the following questions can be found in the *'fintech_part_1.ipynb'* which is an IPython notebook. The notebook is in the following path *'src/fintech_part_1.ipynb'*. Also, the notebook references the 'fintech' library which was created for the purpose of this assignment. The library can be found in *'/src'*. All the code is well commented and easy to follow with.

1.1 Question 1

The objective of this question is to analyse the data for different instruments, based on last year's data, and provide your opinion in terms of their respective risk and return.

1.1.1 Q1 (i)

Download daily closing price data for S&P500, FTSE100 and Gold (SPDR) for the years 2014 to 2017.

Code to download the closing prices for the assets specified in this question can be found in 'Question 1 (i)' in the python notebook. The data was downloaded from Yahoo Finance and saved as .csv format in the following path *'src/data/part_1'*. The files were saved as follows *'FTSE100.csv'*, *'S&P500.csv'* and *'GOLD.csv'*. The data was then reloaded from the files and converted into pandas dataframes.

1.1.2 Q1 (ii)

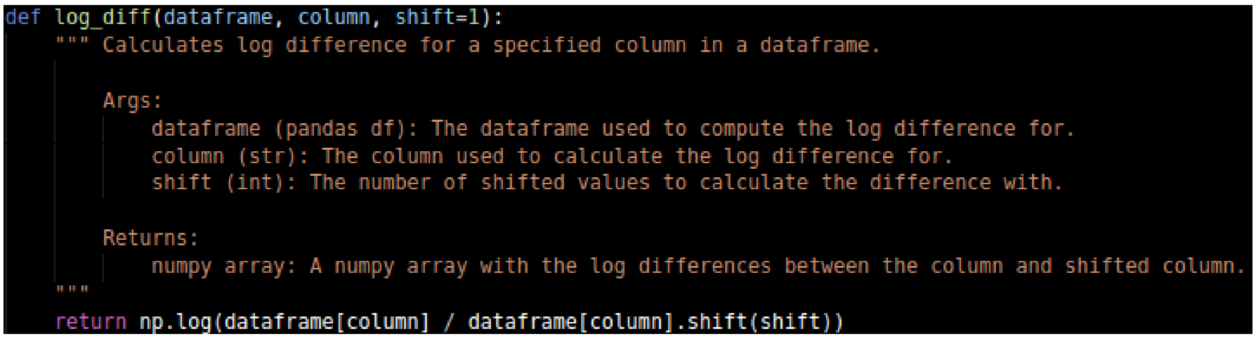
Why log returns are normally preferred from standard arithmetic returns?

Log returns are preferred over arithmetic returns for several reasons, one of which is that when using log returns you are inherently normalizing all the values. The process of normalization makes the returns easier to compare with and this is very useful in analytical situations or machine learning. Another advantage is time-additivity, meaning when using log-returns it is easier to compound returns since you only need to add the values unlike when using arithmetic returns [1]. Also, in theory prices are assumed to be distributed log normally (not always true for every price series) and transforming to log makes the values normally distributed. This is very useful in situations where it is assumed that the values are normally distributed, which is quite common in machine learning and statistics.

1.1.3 Q1 (iii)

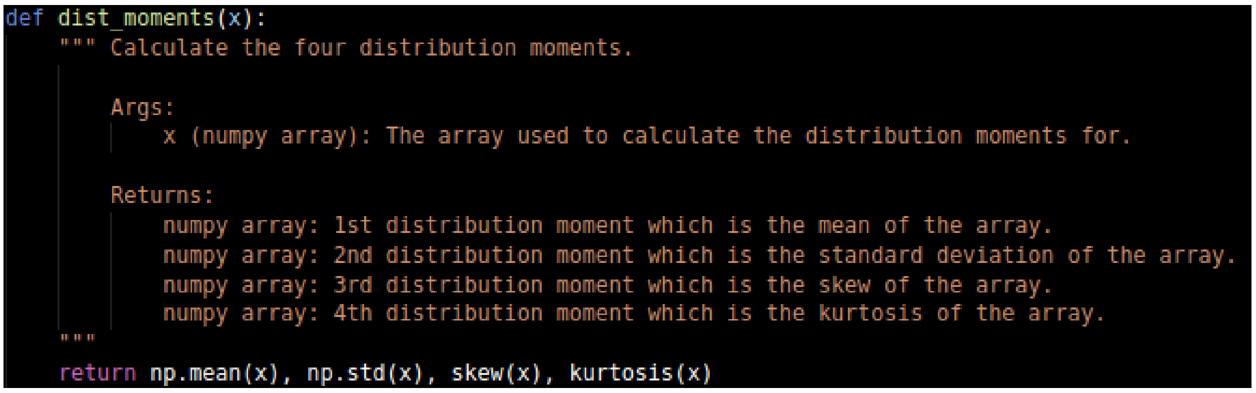
Identify the first 4 distribution moments for each index/product mentioned in part (i). For your calculations utilise daily log returns. In your answer describe the calculations/steps performed.

First the log returns were calculated for each index/product using the adjusted closing price. Once the log returns were computed a new column was added in each pandas dataframe called "Log Returns", so it can be utilised when calculating the distribution moments. A function to compute the log returns was created in the 'fintech' library and another function was also created to compute the four distribution moments. These functions can be shown in Fig. 1.1 and Fig. 1.2. The first and second distribution moments were calculated using numpy [2] functions. The third and the forth distribution moments were calculated using scipy [3].



```python
def log_diff(dataframe, column, shift=1):
    """ Calculates log difference for a specified column in a dataframe.

        Args:
            dataframe (pandas df): The dataframe used to compute the log difference for.
            column (str): The column used to calculate the log difference for.
            shift (int): The number of shifted values to calculate the difference with.

        Returns:
            numpy array: A numpy array with the log differences between the column and shifted column.
    """
    return np.log(dataframe[column] / dataframe[column].shift(shift))
```

Figure 1.1: Function to compute log returns.



```python
def dist_moments(x):
    """ Calculate the four distribution moments.

        Args:
            x (numpy array): The array used to calculate the distribution moments for.

        Returns:
            numpy array: 1st distribution moment which is the mean of the array.
            numpy array: 2nd distribution moment which is the standard deviation of the array.
            numpy array: 3rd distribution moment which is the skew of the array.
            numpy array: 4th distribution moment which is the kurtosis of the array.
    """
    return np.mean(x), np.std(x), skew(x), kurtosis(x)
```

Figure 1.2: Function to compute four distribution moments.

To further explain these functions used in the code, the following equations are presented. Eq. 1.1 is used to compute the log returns, which basically takes the natural log of the adjusted price at t divided by the adjusted price at $t-1$.

$$Log\ Returns = \ln(\frac{S_t}{S_{t-1}}) \tag{1.1}$$

The first distribution moment is calculated using the Eq. 1.2 which basically is the mean of the computed log returns. By finding the mean of the log returns we get a value of the expected return and it gives us the centre point under the distribution. The second distribution moment as shown in Eq. 1.3 is the standard deviation of the log returns. Such value shows us the volatility of the index/product and measures the disperation in the distribution.

The third moment is the computed Skew of the log returns and this is shown in Eq. 1.4. Skew is calculated by taking the sum of the log return Y_i subtracted by the log return mean $\overline{Y}$ and raised to the power of 3. Then this summation is divided by the number of the log return values (N). After doing so this result is divided by the standard deviation raised to the power of 3 which is shown as s^3. This gives us the measure of symmetry for the distribution and show us how skewed the log returns are. Similarly the fourth moment, which is also know as Kurtosis, is calculated in the same way but raised to the power of 4 as shown in Eq. 1.5. This measures the shape of our distribution for the log returns (tails, tall or flat) and the distribution is set to be normally distributed if it is close to 3.

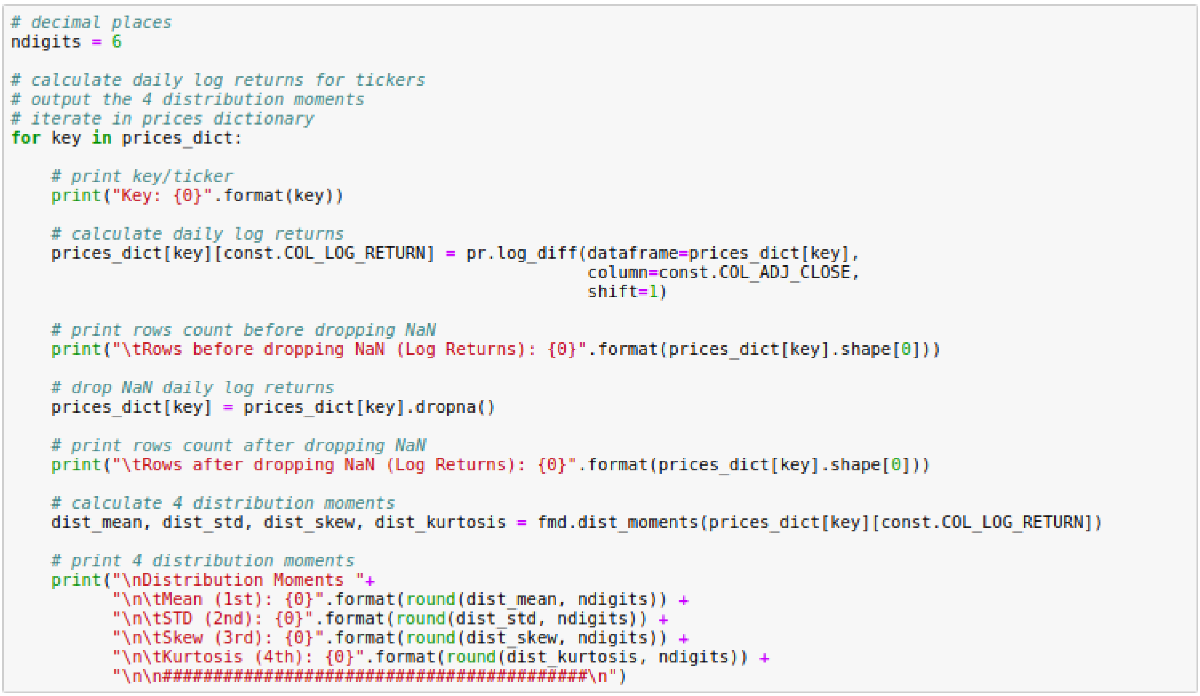
$$1st\ Moment = \frac{\Sigma x}{N} \tag{1.2}$$

$$2nd\ Moment = \sqrt{\frac{\Sigma(x-\overline{x})^2}{N}} \tag{1.3}$$

$$3rd\ Moment = \frac{\sum_{i=1}^{N}(Y_i-\overline{Y})^3/N}{s^3} \tag{1.4}$$

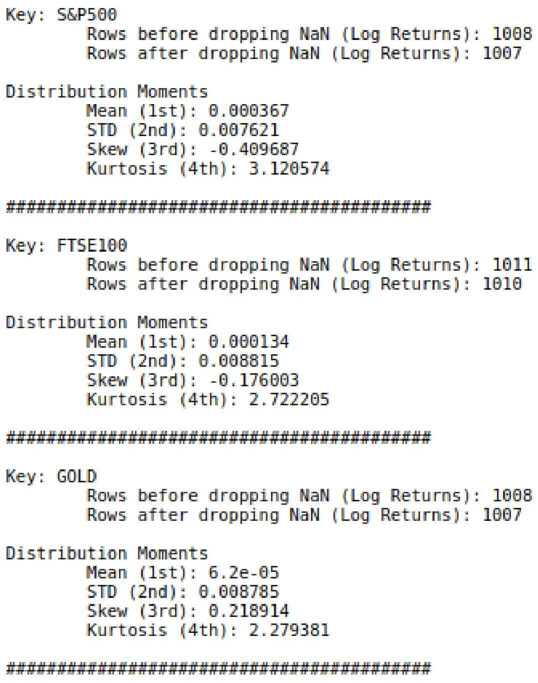
$$4th\ Moment = \frac{\sum_{i=1}^{N}(Y_i-\overline{Y})^4/N}{s^4} \tag{1.5}$$

The functions described above were called from the notebook as shown in Fig. 1.3 and the other Fig. 1.4 shows the output of the distribution moments for each index/product.



```python
# decimal places
ndigits = 6

# calculate daily log returns for tickers
# output the 4 distribution moments
# iterate in prices dictionary
for key in prices_dict:

    # print key/ticker
    print("Key: {0}".format(key))

    # calculate daily log returns
    prices_dict[key][const.COL_LOG_RETURN] = pr.log_diff(dataframe=prices_dict[key],
                                                        column=const.COL_ADJ_CLOSE,
                                                        shift=1)

    # print rows count before dropping NaN
    print("\tRows before dropping NaN (Log Returns): {0}".format(prices_dict[key].shape[0]))

    # drop NaN daily log returns
    prices_dict[key] = prices_dict[key].dropna()

    # print rows count after dropping NaN
    print("\tRows after dropping NaN (Log Returns): {0}".format(prices_dict[key].shape[0]))

    # calculate 4 distribution moments
    dist_mean, dist_std, dist_skew, dist_kurtosis = fmd.dist_moments(prices_dict[key][const.COL_LOG_RETURN])

    # print 4 distribution moments
    print("\nDistribution Moments "+
          "\n\tMean (1st): {0}".format(round(dist_mean, ndigits)) +
          "\n\tSTD (2nd): {0}".format(round(dist_std, ndigits)) +
          "\n\tSkew (3rd): {0}".format(round(dist_skew, ndigits)) +
          "\n\tKurtosis (4th): {0}".format(round(dist_kurtosis, ndigits)) +
          "\n\n##########################################\n")
```

Figure 1.3: Code used in the notebook to get distribution moments.



```
Key: S&P500
        Rows before dropping NaN (Log Returns): 1008
        Rows after dropping NaN (Log Returns): 1007

Distribution Moments
        Mean (1st): 0.000367
        STD (2nd): 0.007621
        Skew (3rd): -0.409687
        Kurtosis (4th): 3.120574

##########################################

Key: FTSE100
        Rows before dropping NaN (Log Returns): 1011
        Rows after dropping NaN (Log Returns): 1010

Distribution Moments
        Mean (1st): 0.000134
        STD (2nd): 0.008815
        Skew (3rd): -0.176003
        Kurtosis (4th): 2.722205

##########################################

Key: GOLD
        Rows before dropping NaN (Log Returns): 1008
        Rows after dropping NaN (Log Returns): 1007

Distribution Moments
        Mean (1st): 6.2e-05
        STD (2nd): 0.008785
        Skew (3rd): 0.218914
        Kurtosis (4th): 2.279381

##########################################
```

Figure 1.4: Function to compute log returns

1.1.4 Q1 (iv)

Comment on the measured statistics from the perspective of risk and return. In your answer compare the results obtained.

As described the first moment is the expected return of an asset, while the second moment gives as the expected volatility of an asset. This means that the volatility gives as the dispersion of where the price might go. The higher the volatility the larger the swings for the price over time, which can make an asset quite risky since the price can move in an upward or downward direction. The Skew measure will help us determine the extremes of where the price might go. The more skewed an asset is the less accurate financial models will be, since most of them rely on normally distributed data. When having a positively skewed returns, means that there were frequent small losses and a few large gains, while a negatively skewed returns, means that there were frequent small gains and a few large loses. In terms of risk and reward an attractive asset would be an asset with the following traits; high expected returns, low volatility, a positive skew and a Kurtosis measure close to 3 (normal distribution).

Looking at the S&P500 statistics the expected returns (0.036%) and volatility (0.7621%) is more attractive in terms of risk and return than the FTSE100 index with expected returns (0.0134%) and volatility (0.8815%). This is because S&P500 has higher expected return with less volatility making it a safer bet according to historic data. Both assets have a negative Skew and the Kurtosis measure for both assets are close to 3 with a difference of +0.120574 and –0.2278 respectively.

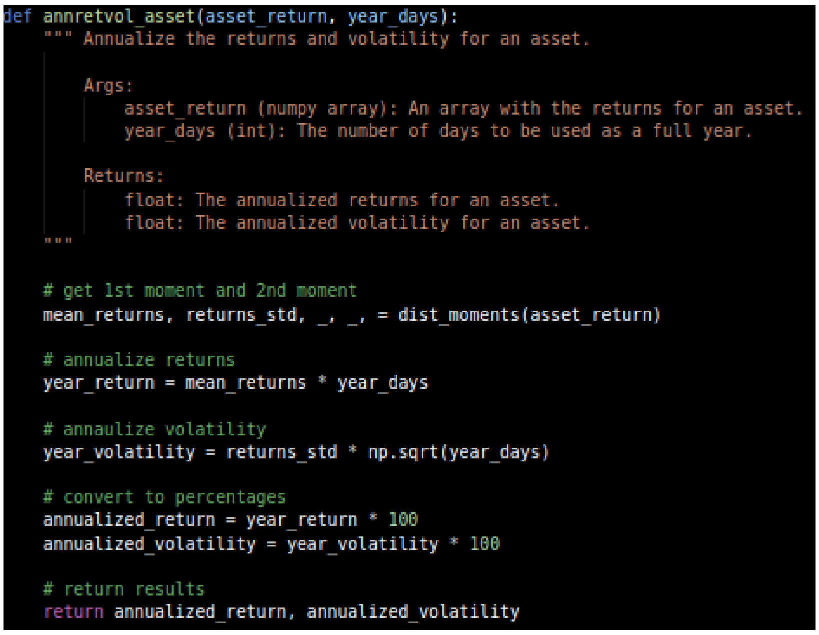
On the other hand, the gold asset has an expected return (0.0062%) and volatility (0.8785%), which although it is less volatile than the FTSE100, the expected return is significantly lower. Unlike the other assets, the FTSE100 has a positively skewed return which is preferred over negatively skewed values. Also, the Kurtosis is close to 3 with a difference of -0.72061.

In terms of risk and return, the most attractive asset from the results obtained is the S&P500 index.

1.1.5 Q1 (v)

Annualize daily return (first moment) and volatility (second moment). In your scaling process assume 250 days for the year. In your answer describe the calculations/steps performed.

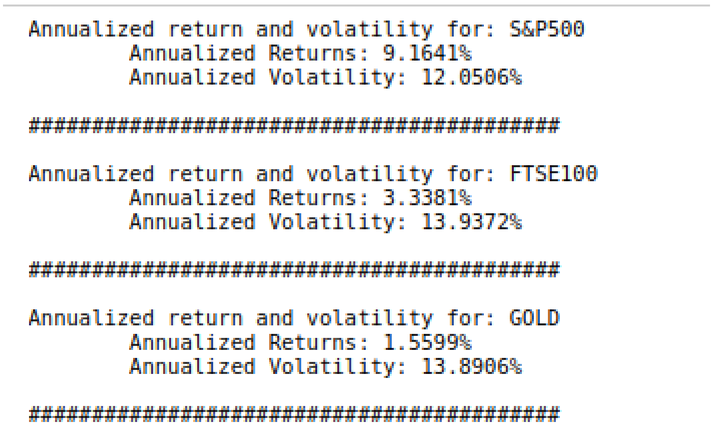
The code for this task can be found in ‘Question 1 (v)’ in the python notebook. A function called ‘annretvol_asset’ was created in the ‘fintech’ library to annualize the log returns and volatility. This function is shown in Fig. 1.5.



```python
def annretvol_asset(asset_return, year_days):
    """ Annualize the returns and volatility for an asset.

        Args:
            asset_return (numpy array): An array with the returns for an asset.
            year_days (int): The number of days to be used as a full year.

        Returns:
            float: The annualized returns for an asset.
            float: The annualized volatility for an asset.
    """

    # get 1st moment and 2nd moment
    mean_returns, returns_std, _, _, = dist_moments(asset_return)

    # annualize returns
    year_return = mean_returns * year_days

    # annaulize volatility
    year_volatility = returns_std * np.sqrt(year_days)

    # convert to percentages
    annualized_return = year_return * 100
    annualized_volatility = year_volatility * 100

    # return results
    return annualized_return, annualized_volatility
```

Figure 1.5: Function to annualize returns and volatility for an asset.

In the first line of the function the first moment and second moment are computed for the specific asset. Once these are computed the daily log return is annualized by simply using ($First\ moment \times 250$) and the volatility is annualized using ($Second\ moment \times \sqrt{250}$). The formulas described were used since returns scale with time, while volatility scales with square root of time. The computed values are then converted to percentages and returned by the function. To calculate the annualized returns and volatility for each asset, this function was called from a loop and the daily log returns found in (iii) were passed as a parameter. Results for these two measurements are shown in Fig. 1.6.



```
Annualized return and volatility for: S&P500
        Annualized Returns: 9.1641%
        Annualized Volatility: 12.0506%

##########################################

Annualized return and volatility for: FTSE100
        Annualized Returns: 3.3381%
        Annualized Volatility: 13.9372%

##########################################

Annualized return and volatility for: GOLD
        Annualized Returns: 1.5599%
        Annualized Volatility: 13.8906%

##########################################
```

Figure 1.6: Annualized returns and volatility for the three assets.

1.1.6 Q1 (vi)

By considering the last closing price at the end of 2017, and the annualized volatility from question (v), what would be the price level of SP 500 after 1

month, that according to normal probability, there is a 32% chance that the actual price will be above it. Show your workings.

Code for this task can be found in 'Question 1 (vi)' in the python notebook. To find the price level for the asset after one month the last closing price was fetched, which was equal to \$2,673.61. After doing so the annualized volatility (12.05%) was scaled to one month using the following formula ($\frac{12.05}{100} \times \sqrt{\frac{20}{250}}$) and it was assumed that a one month period contains 20 days. Using these two values the price deviation could be computed using ($\$2,673.61 \times (\frac{12.05}{100} \times \sqrt{\frac{20}{250}})$) which outputs \$91.12. So the price levels after one month is in the range of ($\$2,673.61 - \$91.12$) to ($\$2,673.61 + \$91.12$) which is equal to the range of \$2,582.49 - \$2,764.73.

Using the Z-score equation as shown in Eq. 1.6, we can find the number of standard deviations a value is from the mean. In this case the mean is set to be the last closing price for the asset, which is equal to \$2,673.61. We found that the price has a 16% chance that it will be above of \$2,764.73. To find a price which has a 32% chance of being above the actual price, we rearrange the formula to find X. Using a z-score table we know that a z-score of 0.47 has 68% area under the distribution. So, by finding this value we would be able to find a price which has 32% chance that the actual price will be above it.

$$Z = \frac{X - \mu}{\sigma} \tag{1.6}$$

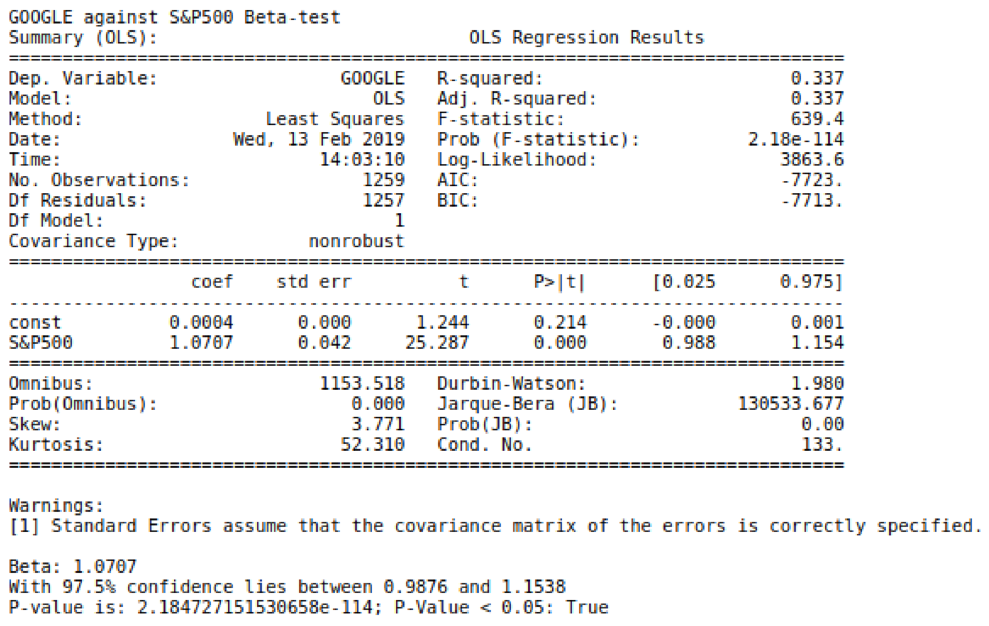
The price which has 32% chance of being above the actual price is ($0.47 \times \$91.12 + \$2,673.61$), which is equal to \$2716.44.

1.1.7 Q1 (vii)

Download the Google and Amazon daily prices for the last 5 years (till 31/12/2017). By utilizing a regression model, perform the Beta-test against the S&P 500 index. Comment on your findings.

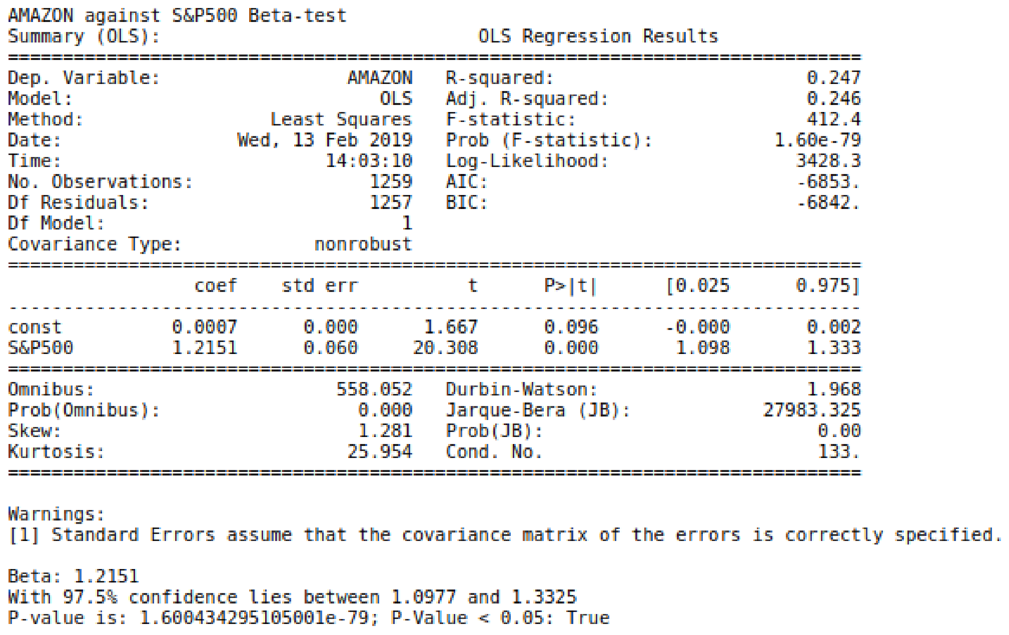
Code for this task can be found in 'Question 1 (vii)' in the python notebook. The data was downloaded from Yahoo Finance and saved as .csv format in the following path *'src/data/part_1'*. The files were saved as *'GOOGLE.csv'*, *'AMAZON.csv'* and *'S&P500BETA.csv'*. The data was then reloaded from the files and converted into pandas dataframes. A new dataframe with the percentage changes (daily adjusted closing prices) for the three assets was created, with each column holding the percentage changes for each asset.

A function called 'beta_test_ols' (uses Ordinary Least Squares from 'statsmodels' [4]) was created in the 'fintech' library and was utilised to perform the beta-test. Two beta-test were conducted 'GOOGLE VS S&P500' as shown in Fig. 1.7a and 'AMAZON vs S&P500' as shown in Fig. 1.7b. The plots for the results are shown in Fig. 1.8a and Fig. 1.8b respectively.



```
GOOGLE against S&P500 Beta-test
Summary (OLS):                             OLS Regression Results
==============================================================================
Dep. Variable:                 GOOGLE   R-squared:                       0.337
Model:                            OLS   Adj. R-squared:                  0.337
Method:                 Least Squares   F-statistic:                     639.4
Date:                Wed, 13 Feb 2019   Prob (F-statistic):          2.18e-114
Time:                        14:03:10   Log-Likelihood:                 3863.6
No. Observations:                1259   AIC:                            -7723.
Df Residuals:                    1257   BIC:                            -7713.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0004      0.000      1.244      0.214      -0.000       0.001
S&P500         1.0707      0.042     25.287      0.000       0.988       1.154
==============================================================================
Omnibus:                     1153.518   Durbin-Watson:                   1.980
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           130533.677
Skew:                           3.771   Prob(JB):                         0.00
Kurtosis:                      52.310   Cond. No.                         133.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Beta: 1.0707
With 97.5% confidence lies between 0.9876 and 1.1538
P-value is: 2.184727151530658e-114; P-Value < 0.05: True
```

(a) GOOGLE VS S&P500 Beta-test.



```
AMAZON against S&P500 Beta-test
Summary (OLS):                             OLS Regression Results
==============================================================================
Dep. Variable:                 AMAZON   R-squared:                       0.247
Model:                            OLS   Adj. R-squared:                  0.246
Method:                 Least Squares   F-statistic:                     412.4
Date:                Wed, 13 Feb 2019   Prob (F-statistic):           1.60e-79
Time:                        14:03:10   Log-Likelihood:                 3428.3
No. Observations:                1259   AIC:                            -6853.
Df Residuals:                    1257   BIC:                            -6842.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0007      0.000      1.667      0.096      -0.000       0.002
S&P500         1.2151      0.060     20.308      0.000       1.098       1.333
==============================================================================
Omnibus:                      558.052   Durbin-Watson:                   1.968
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            27983.325
Skew:                           1.281   Prob(JB):                         0.00
Kurtosis:                      25.954   Cond. No.                         133.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Beta: 1.2151
With 97.5% confidence lies between 1.0977 and 1.3325
P-value is: 1.600434295105001e-79; P-Value < 0.05: True
```

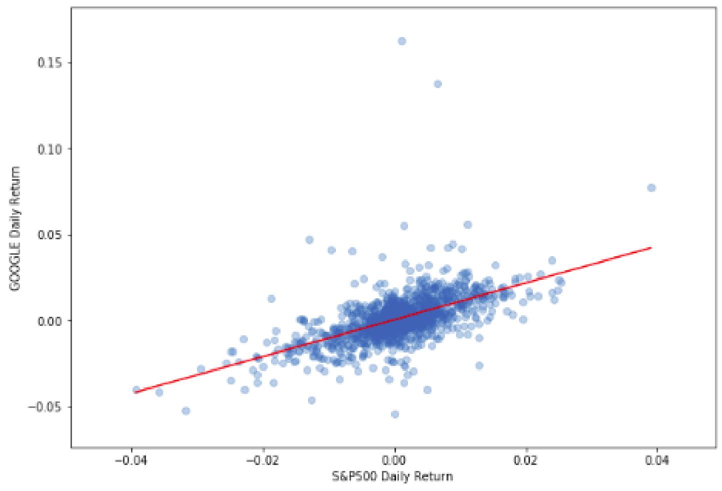
(b) AMAZON vs S&P500 Beta-test.

The beta-test (Capital Asset Pricing Model (CAPM)) is used to provide a measure which describes the risk/return ratio for the two assets. In this task we use the beta-test to test the relation of a stock price relative to a stock market index (systematic risk). The beta is calculated using Eq. 1.7.

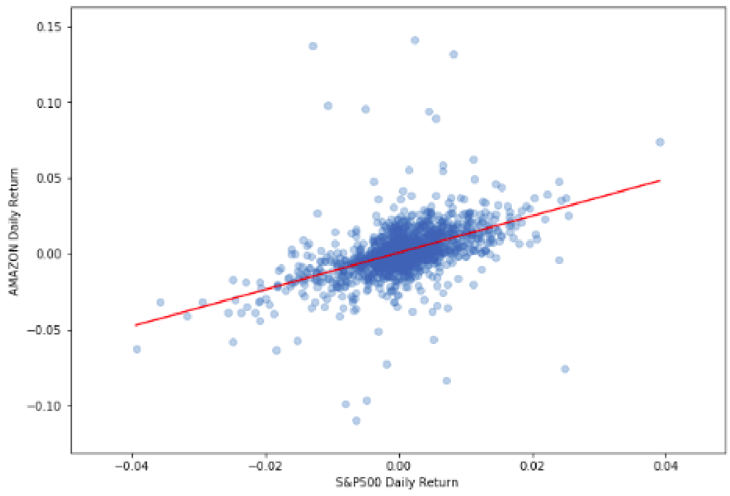
$$Market\ Return = \alpha + (\beta \times stock\ return) \tag{1.7}$$

Let's look at the first result 'GOOGLE vs S&P500' which has a beta of 1.0707 and a p-value which makes this measurement statistically significant. With 95% confidence that the value lies between 0.9876 and 1.1538. Since the $\beta > 1$, this asset is volatile and moves with the rest of the market. This means that if the market moves in an upwards direction this asset is likely to move in the same direction and the same goes if the market moves in a downward direction. So as a benchmark we know that this asset is positively correlated with the market index as shown in the plot in Fig. 1.8a. Looking at the confidence levels, although the beta can be less than 1 it is still a very high value and this means that this stock is likely to move with the market or follows a similar trend.

In the other test 'AMAZON VS S&P500' the beta was 1.2151 and has a p-vale which makes the measurement statistically significant. With 95% confidence that the value lies between 1.0977 and 1.3325. Like the previous test this stock is volatile and moves with the rest of the market, since it's correlated according to the test. One must note that the beta confidence levels and the beta is a bit higher than the previous stock which makes this asset to move a bit higher or lower than the benchmark (market-index). This means if the market is moving in an upward direction this stock will see more returns relative to the market and it will see more loses when moving in a downward direction. In fact the beta for the Google stock is close to 1 meaning it will move very similar to the market and this difference between the movements of the two assets can be seen in both the plots Fig. 1.8a and Fig. 1.8b.



(a) GOOGLE VS S&P500 Beta-test plot.



(b) AMAZON vs S&P500 Beta-test plot.

We would like to add that beta-test does not detect any unsystematic risk. Since we are measuring beta for separate stocks it will give us an indication of how much risk such assets will add or subtract to a portfolio. Such measure does not always predict the stock movements, but it can be a useful indication when building a portfolio as it gives us some indication of how a stock moves with the market. It is also important to note that we used the daily values to measure the beta while it is more common to use the monthly measurements. When using the monthly data, it can faster to compute, easier to identify change in trends and can be good for long term forecasting but if you are looking at daily changes it is better to use daily data. On the other hand, daily is more optimal if you are forecasting for short to medium periods of time but data can be susceptible to noise. Choosing the time period to work with depends on the problem you are trying to solve/predict.

1.2 Question 2

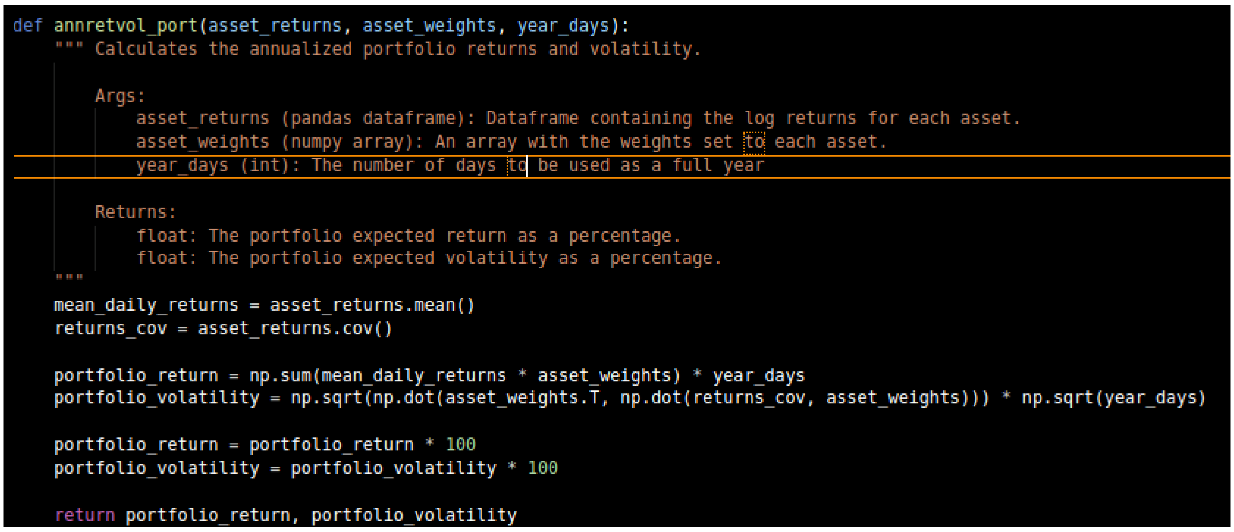
Modern or mean-variance portfolio theory (MPT) is a major cornerstone of financial theory. Based on this theoretical breakthrough the Nobel Prize in Economics was awarded to its inventor, Harry Markowitz, in 1990. Using the data from Question 1, we need to investigate the right allocation across a portfolio made up of 3 investments, S&P500, FTSE 100 and Gold (SPDR).

1.2.1 Q2 (i)

In question 1, you identified the individual expected return and volatility of the 3 investments separately. Calculate the expected return and volatility of the portfolio, considering equal weight for the 3 investments.

Code for this question can be found in the notebook in section 'Question 2 (i)'. Before calculating both measurements for the portfolio, it was noted that there is some discrepancy between the dates of the assets. The dates of S&P500 and Gold were not matching with the dates for the FTSE100 index. To solve this dates were aligned to capture more accurate measurements.

To calculate the annualized return and volatility for the portfolio a new function called 'ann_ret_vol' was created in the 'fintech' library as shown in Fig. 1.9. This was later called from the notebook cell. As shown, this function takes three parameters. For first parameter a new pandas dataframe with the log returns for each asset was created (each column has the log returns for each asset). After doing so an equal weight was set as a numpy array and passed as a parameter. In this task it was assumed that a year has 250 days.



```python
def annretvol_port(asset_returns, asset_weights, year_days):
    """ Calculates the annualized portfolio returns and volatility.

        Args:
            asset_returns (pandas dataframe): Dataframe containing the log returns for each asset.
            asset_weights (numpy array): An array with the weights set to each asset.
            year_days (int): The number of days to be used as a full year

        Returns:
            float: The portfolio expected return as a percentage.
            float: The portfolio expected volatility as a percentage.
    """
    mean_daily_returns = asset_returns.mean()
    returns_cov = asset_returns.cov()

    portfolio_return = np.sum(mean_daily_returns * asset_weights) * year_days
    portfolio_volatility = np.sqrt(np.dot(asset_weights.T, np.dot(returns_cov, asset_weights))) * np.sqrt(year_days)

    portfolio_return = portfolio_return * 100
    portfolio_volatility = portfolio_volatility * 100

    return portfolio_return, portfolio_volatility
```

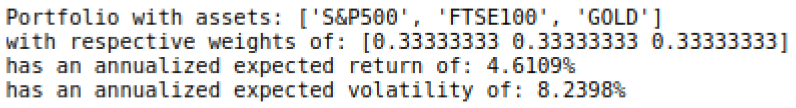
Figure 1.9: Function to measure expected returns and volatility for portfolio.

To calculate the expected return for the portfolio the weighted sum of the mean return for each asset was taken and multiplied by the number of days (to annualize) as shown in Eq. 1.8.

$$E(r) = \Sigma w_i k_i \tag{1.8}$$

On the other hand, calculating portfolio risk is a bit more complicated than taking the sum of the weighted volatility of each asset. To calculate the volatility of the portfolio, first we need to find the covariance between the asset's returns, since the risk of an asset may be correlated with another asset. Eq. 1.9 shows how to calculate volatility for a portfolio with two assets but this can be easily extended to three assets when writing it in code as shown in Fig. 1.9. The final measurements for the portfolio are shown in Fig. 1.10.

$$\sigma = \sqrt{(w_1^2\sigma_1^2) + (w_2^2\sigma_2^2) + 2(w_1)(w_2)(Corr(R_1, R_2)\sigma_1\sigma_2)} \tag{1.9}$$



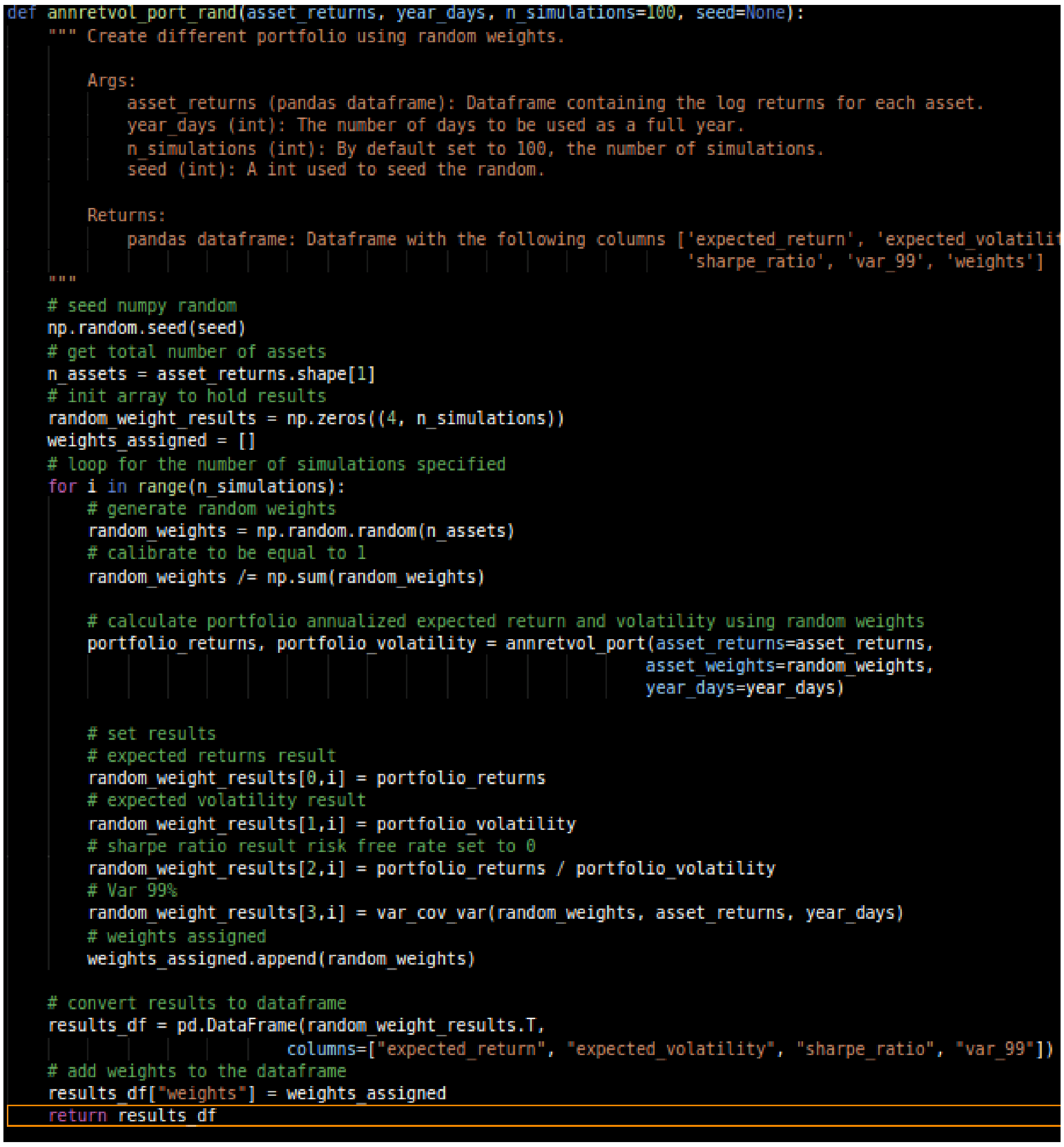
```
Portfolio with assets: ['S&P500', 'FTSE100', 'GOLD']
with respective weights of: [0.33333333 0.33333333 0.33333333]
has an annualized expected return of: 4.6109%
has an annualized expected volatility of: 8.2398%
```

Figure 1.10: Results for portfolio volatility and returns.

1.2.2 Q2 (ii)

Investigate different portfolio expected return and volatility by simulating different random weights of your investments (2000 simulations). Assume that all weights have to be > 0 and that the sum of all weights should be equal to 1. Create a plot showing the expected return (y-axis) and volatility (x-axis) for different/random portfolio weights.

The code for this simulation can be found in 'Question 2 (ii)' in the notebook and a function called 'annretvol_port_rand' was created in the 'fintech' library to simulate different weights for a portfolio as shown in Fig. 1.11. The plot shown in Fig. 1.12 shows the expected return (y-axis) and the volatility (x-axis) for these simulations.



```
def annretvol_port_rand(asset_returns, year_days, n_simulations=100, seed=None):
    """ Create different portfolio using random weights.

        Args:
            asset_returns (pandas dataframe): Dataframe containing the log returns for each asset.
            year_days (int): The number of days to be used as a full year.
            n_simulations (int): By default set to 100, the number of simulations.
            seed (int): A int used to seed the random.

        Returns:
            pandas dataframe: Dataframe with the following columns ['expected_return', 'expected_volatili
                                                                    'sharpe_ratio', 'var_99', 'weights']
    """
    # seed numpy random
    np.random.seed(seed)
    # get total number of assets
    n_assets = asset_returns.shape[1]
    # init array to hold results
    random_weight_results = np.zeros((4, n_simulations))
    weights_assigned = []
    # loop for the number of simulations specified
    for i in range(n_simulations):
        # generate random weights
        random_weights = np.random.random(n_assets)
        # calibrate to be equal to 1
        random_weights /= np.sum(random_weights)

        # calculate portfolio annualized expected return and volatility using random weights
        portfolio_returns, portfolio_volatility = annretvol_port(asset_returns=asset_returns,
                                                                 asset_weights=random_weights,
                                                                 year_days=year_days)

        # set results
        # expected returns result
        random_weight_results[0,i] = portfolio_returns
        # expected volatility result
        random_weight_results[1,i] = portfolio_volatility
        # sharpe ratio result risk free rate set to 0
        random_weight_results[2,i] = portfolio_returns / portfolio_volatility
        # Var 99%
        random_weight_results[3,i] = var_cov_var(random_weights, asset_returns, year_days)
        # weights assigned
        weights_assigned.append(random_weights)

    # convert results to dataframe
    results_df = pd.DataFrame(random_weight_results.T,
                              columns=["expected_return", "expected_volatility", "sharpe_ratio", "var_99"])
    # add weights to the dataframe
    results_df["weights"] = weights_assigned
    return results_df
```

Figure 1.11: Function to generate portfolio returns and volatility using random weights.

]

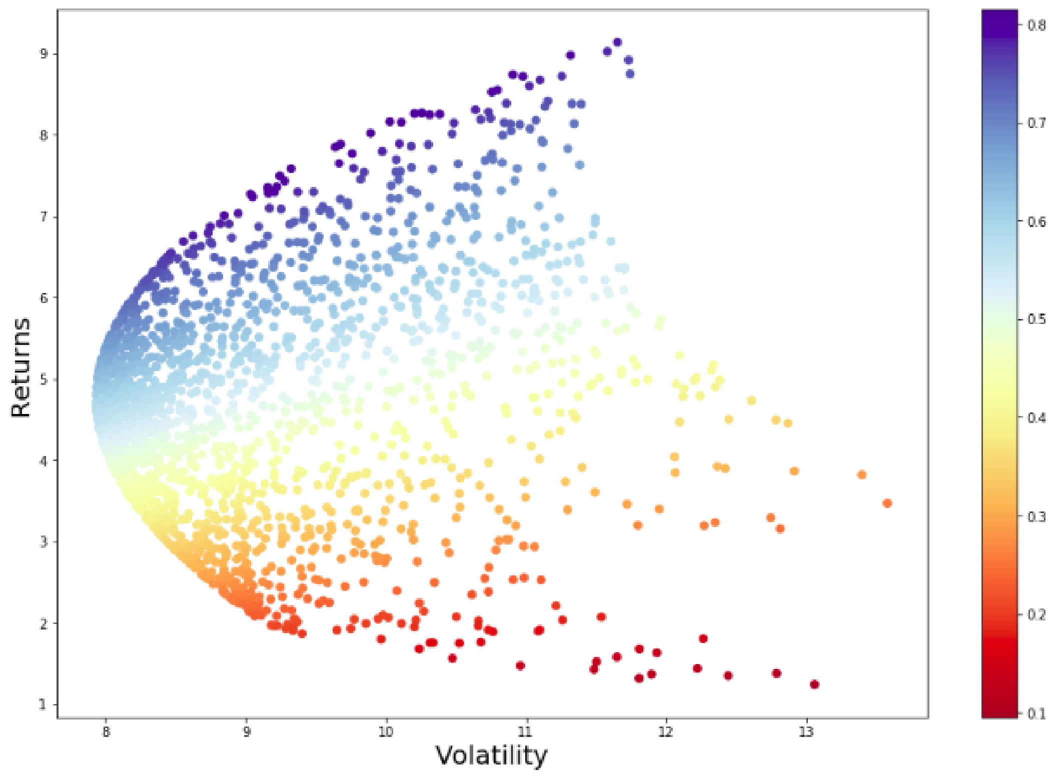


Figure 1.12: Portfolio returns and volatility using random weights (2000 Simulations).

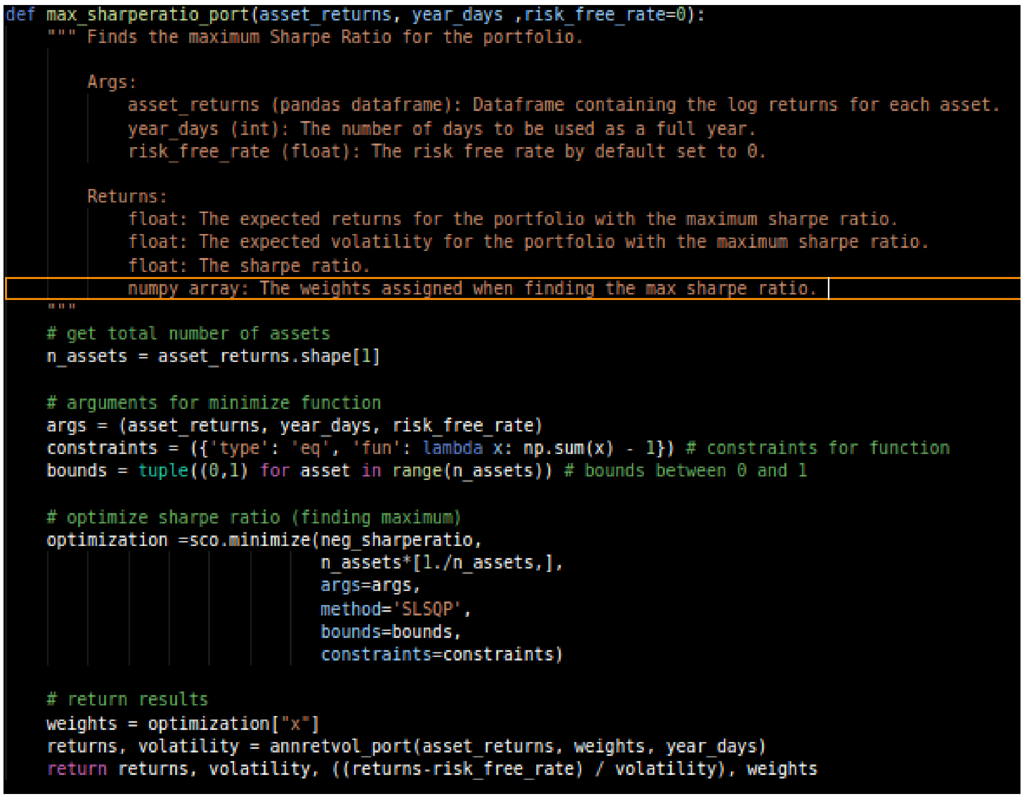
1.2.3 Q2 (iii)

Using an optimisation library (e.g. using solver in excel or an optimisation library in python), identify the two portfolios that will return (a) the highest Sharpe ratio and (b) the lowest Value at Risk. For the Sharpe ratio, assume that the risk free rate is zero. Using the plot in question (ii), indicate the position of these two portfolios.

In your answers explain the method/steps used.

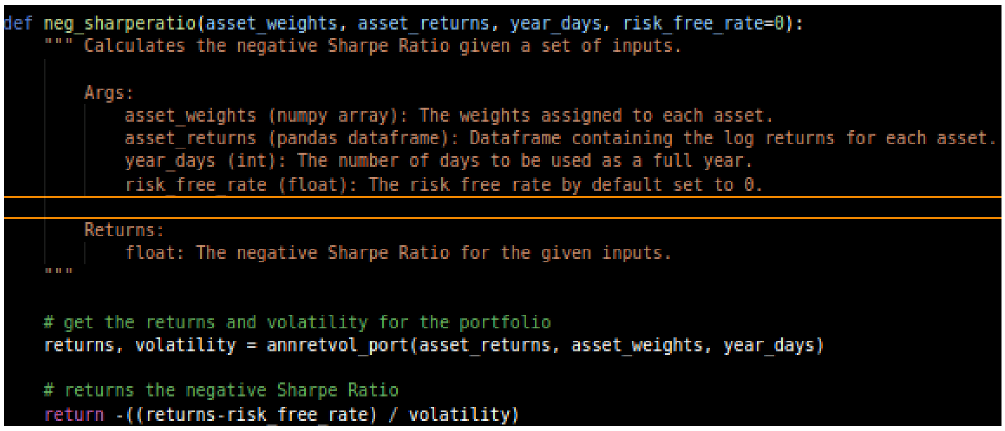
The code for this question is shown in 'Question 2 (iii)' and as an optimization library a third party library called 'scipy.optimize' [3] was utilised for this process. For the highest Sharpe ratio a function called 'max_sharperatio_port' was created in the 'fintech' library as shown in Fig. 1.13.

In this function the third party library described, is used to minimize the negative Sharpe ratio which gives us the portfolio with the highest Sharpe ratio. The function which is being minimized is shown in Fig. 1.14. After the function finds the minimum, the values described in the code comments in Fig. 1.13 are returned. The final output for the highest Sharpe Ratio is shown in Fig. 1.15.



```python
def max_sharperatio_port(asset_returns, year_days ,risk_free_rate=0):
    """ Finds the maximum Sharpe Ratio for the portfolio.

        Args:
            asset_returns (pandas dataframe): Dataframe containing the log returns for each asset.
            year_days (int): The number of days to be used as a full year.
            risk_free_rate (float): The risk free rate by default set to 0.

        Returns:
            float: The expected returns for the portfolio with the maximum sharpe ratio.
            float: The expected volatility for the portfolio with the maximum sharpe ratio.
            float: The sharpe ratio.
            numpy array: The weights assigned when finding the max sharpe ratio.
    """
    # get total number of assets
    n_assets = asset_returns.shape[1]

    # arguments for minimize function
    args = (asset_returns, year_days, risk_free_rate)
    constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) # constraints for function
    bounds = tuple((0,1) for asset in range(n_assets)) # bounds between 0 and 1

    # optimize sharpe ratio (finding maximum)
    optimization =sco.minimize(neg_sharperatio,
                                n_assets*[1./n_assets,],
                                args=args,
                                method='SLSQP',
                                bounds=bounds,
                                constraints=constraints)

    # return results
    weights = optimization["x"]
    returns, volatility = annretvol_port(asset_returns, weights, year_days)
    return returns, volatility, ((returns-risk_free_rate) / volatility), weights
```

Figure 1.13: Function used to find the maximum Sharpe Ratio for a portfolio.



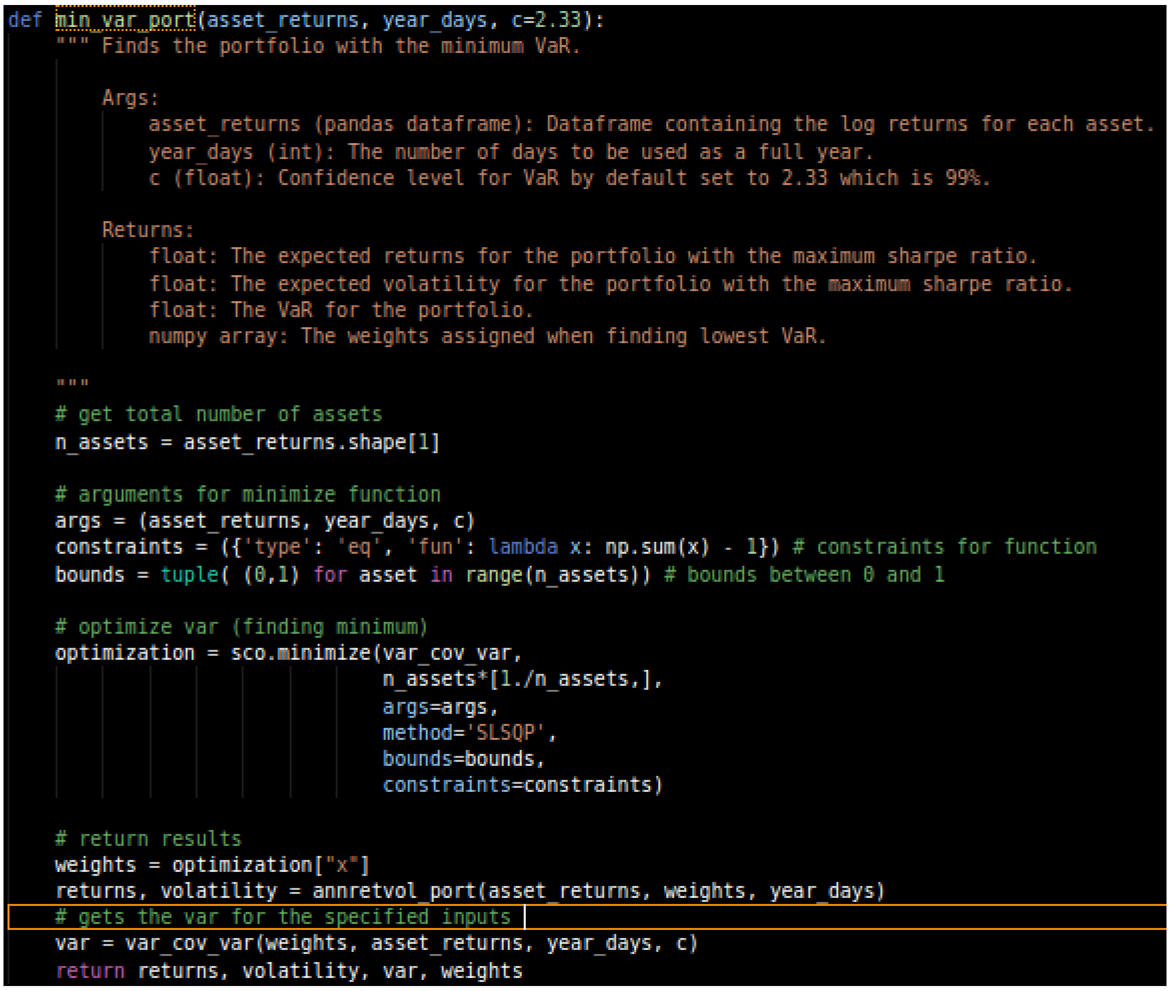
```python
def neg_sharperatio(asset_weights, asset_returns, year_days, risk_free_rate=0):
    """ Calculates the negative Sharpe Ratio given a set of inputs.

        Args:
            asset_weights (numpy array): The weights assigned to each asset.
            asset_returns (pandas dataframe): Dataframe containing the log returns for each asset.
            year_days (int): The number of days to be used as a full year.
            risk_free_rate (float): The risk free rate by default set to 0.

        Returns:
            float: The negative Sharpe Ratio for the given inputs.
    """

    # get the returns and volatility for the portfolio
    returns, volatility = annretvol_port(asset_returns, asset_weights, year_days)

    # returns the negative Sharpe Ratio
    return -((returns-risk_free_rate) / volatility)
```

Figure 1.14: The function being minimized to find the maximum Sharpe Ratio.

```
Portfolio 1 (Red Star) ['S&P500', 'FTSE100', 'GOLD']:
Highest Sharpe Ratio of 0.8159

From Monte Carlo simulation with n: 2000 simulation, the highest Sharpe Ratio was: 0.8144

        Expected returns for this porfolio: 7.914%
        Expected volatility for this porfolio: 9.6992%
        Respective weights of: [8.09364712e-01 4.47775497e-17 1.90635288e-01]
        Sum of all weights: 1.0

#############################################################################
```

Figure 1.15: The results for the portfolio with the max Sharpe Ratio.

For the other measurement, which is the lowest VaR(99%) for the portfolio, another function was created called 'min_var_port' in the 'fintech' library as shown in Fig. 1.16. The same optimization library used to get the previous measurement was utilised.

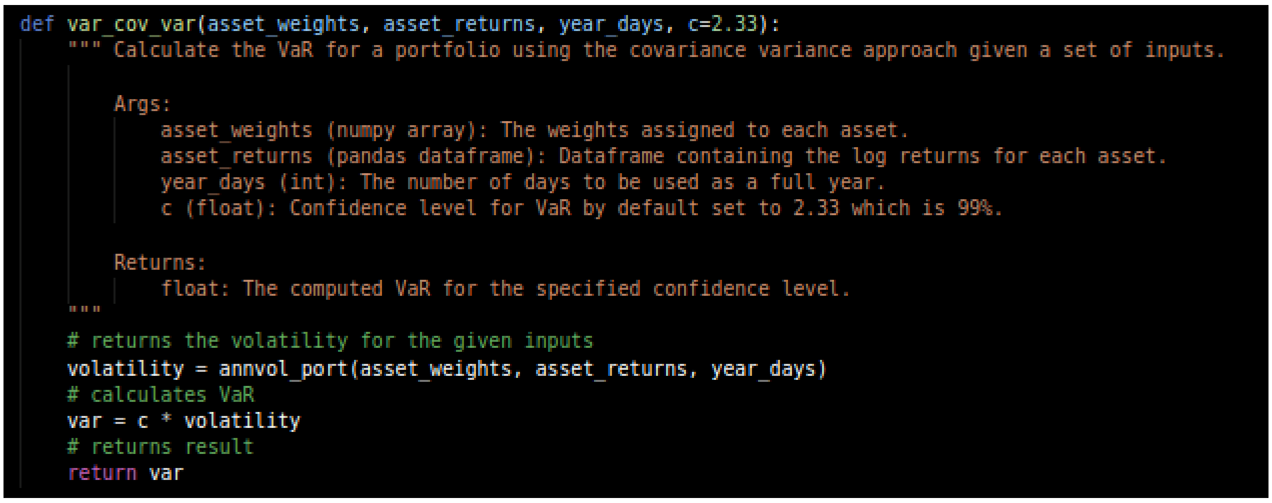


```python
def min_var_port(asset_returns, year_days, c=2.33):
    """ Finds the portfolio with the minimum VaR.

        Args:
            asset_returns (pandas dataframe): Dataframe containing the log returns for each asset.
            year_days (int): The number of days to be used as a full year.
            c (float): Confidence level for VaR by default set to 2.33 which is 99%.

        Returns:
            float: The expected returns for the portfolio with the maximum sharpe ratio.
            float: The expected volatility for the portfolio with the maximum sharpe ratio.
            float: The VaR for the portfolio.
            numpy array: The weights assigned when finding lowest VaR.

    """
    # get total number of assets
    n_assets = asset_returns.shape[1]

    # arguments for minimize function
    args = (asset_returns, year_days, c)
    constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) # constraints for function
    bounds = tuple( (0,1) for asset in range(n_assets)) # bounds between 0 and 1

    # optimize var (finding minimum)
    optimization = sco.minimize(var_cov_var,
                                n_assets*[1./n_assets,],
                                args=args,
                                method='SLSQP',
                                bounds=bounds,
                                constraints=constraints)

    # return results
    weights = optimization["x"]
    returns, volatility = annretvol_port(asset_returns, weights, year_days)
    # gets the var for the specified inputs
    var = var_cov_var(weights, asset_returns, year_days, c)
    return returns, volatility, var, weights
```

Figure 1.16: Function used to get the minimum VaR for a portfolio.

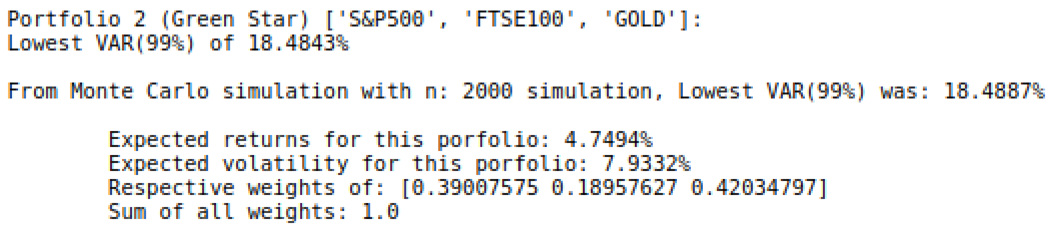
Similarly, to the previous function a value/function is being minimized to get the desired output. The function being minimized is the computed VaR using the covariance variance approach. This function is shown in Fig. 1.17. After the function finds the minimum, the values described in the code comments in Fig. 1.16 are returned. The final output for the lowest VaR(99%) is shown in Fig. 1.18.

Finally these measurements were plotted on the same plot used in (ii) where the red star indicates the portfolio with the maximum Sharpe Ratio and the green star indicates the portfolio with the lowest VaR(99%). This plot is shown in Fig. 1.19.



```python
def var_cov_var(asset_weights, asset_returns, year_days, c=2.33):
    """ Calculate the VaR for a portfolio using the covariance variance approach given a set of inputs.

        Args:
            asset_weights (numpy array): The weights assigned to each asset.
            asset_returns (pandas dataframe): Dataframe containing the log returns for each asset.
            year_days (int): The number of days to be used as a full year.
            c (float): Confidence level for VaR by default set to 2.33 which is 99%.

        Returns:
            float: The computed VaR for the specified confidence level.
    """
    # returns the volatility for the given inputs
    volatility = annvol_port(asset_weights, asset_returns, year_days)
    # calculates VaR
    var = c * volatility
    # returns result
    return var
```

Figure 1.17: The function being minimized to find the lowest VaR.



```
Portfolio 2 (Green Star) ['S&P500', 'FTSE100', 'GOLD']:
Lowest VAR(99%) of 18.4843%

From Monte Carlo simulation with n: 2000 simulation, Lowest VAR(99%) was: 18.4887%

        Expected returns for this porfolio: 4.7494%
        Expected volatility for this porfolio: 7.9332%
        Respective weights of: [0.39007575 0.18957627 0.42034797]
        Sum of all weights: 1.0
```

Figure 1.18: The results for the portfolio with the lowest VaR.

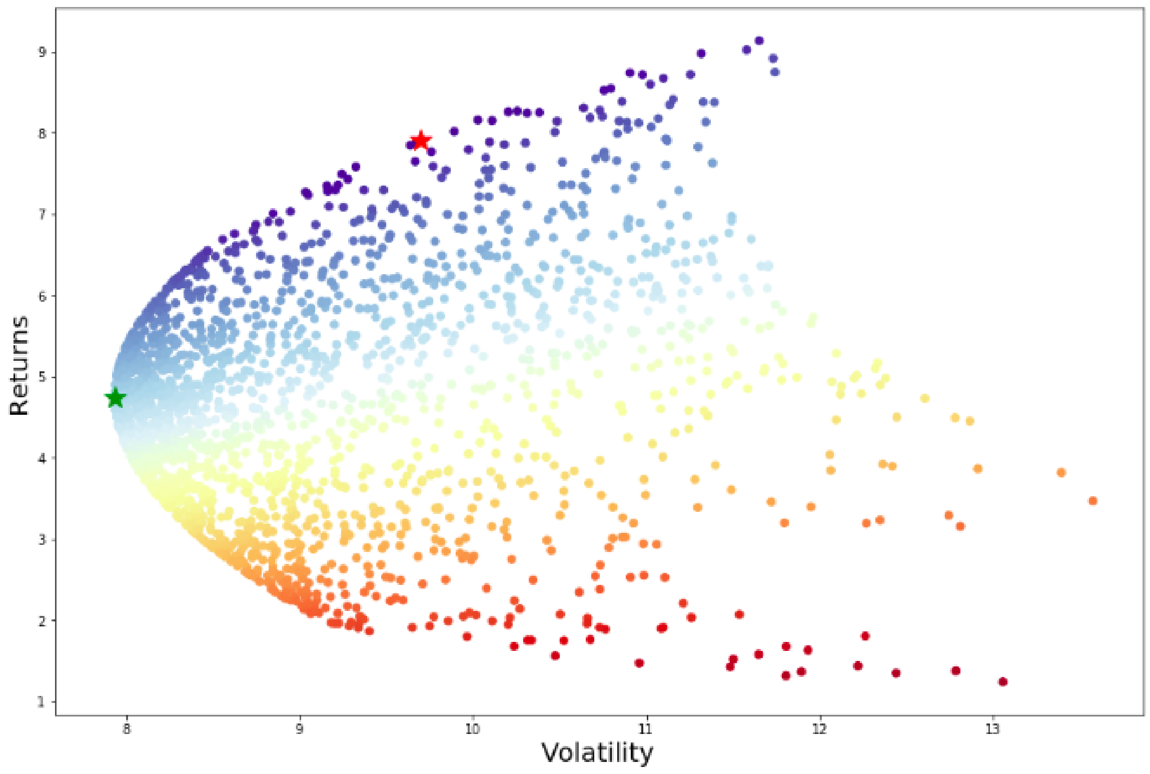


Figure 1.19: The portfolio with the Highest Sharpe Ratio (Red Star) and with the lowest VaR(99%) (Green Start) on the plot used in (ii).

1.3 Question 3

Under the assumption of Markov property, the Binomial tree provides a method of how the future price of an instrument, can be modelled. By investigating the statistics of Google daily returns for the last 5 years (till 31/12/2017), construct a binomial tree model that projects Google stock price for the first 6 months of 2018, on a monthly basis.

In your answers explain the method/steps used.

1.3.1 Q3 (i)

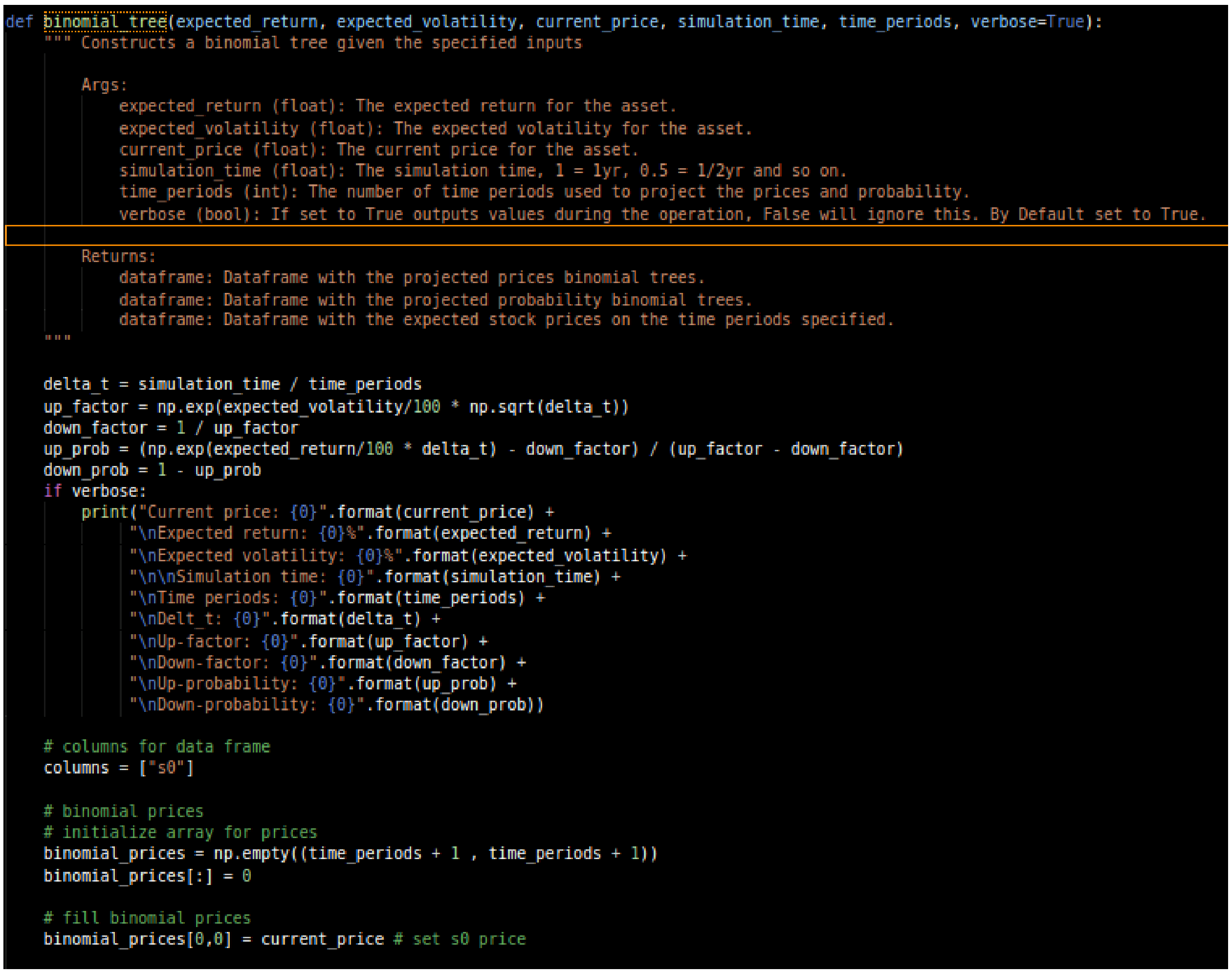
Present the projected probability and stock price binomial trees.

Code used in this question can be found in 'Question 3 (i)' in the notebook. The Google prices downloaded in Q1 (vii) were reloaded and used for this task. After reloading the prices, the log return was computed for each row. The annualized returns and volatility were also computed assuming that a year has 250 days. Furthermore, the last closing price was referenced and saved in a variable, as this will be used as a parameter to compute the binomial tree. A function called 'binomial_tree' was created in the 'fintech' library to construct such tree. This function is shown in Fig. 1.20 and Fig. 1.21.

First off to construct the binomial tree we need to compute the following variables; time step ($\frac{0.5}{6}$) since we have half a year and projecting for 6 months, Up Factor ($\exp(volatility \times \sqrt{time\ step})$), Down Factor ($\frac{1}{Up\ Factor}$), Up Probability ($\frac{(\exp(return \times time\ step))}{(Up\ Factor - Down\ Factor)}$) and Down Probability ($1 - Up\ Probability$).

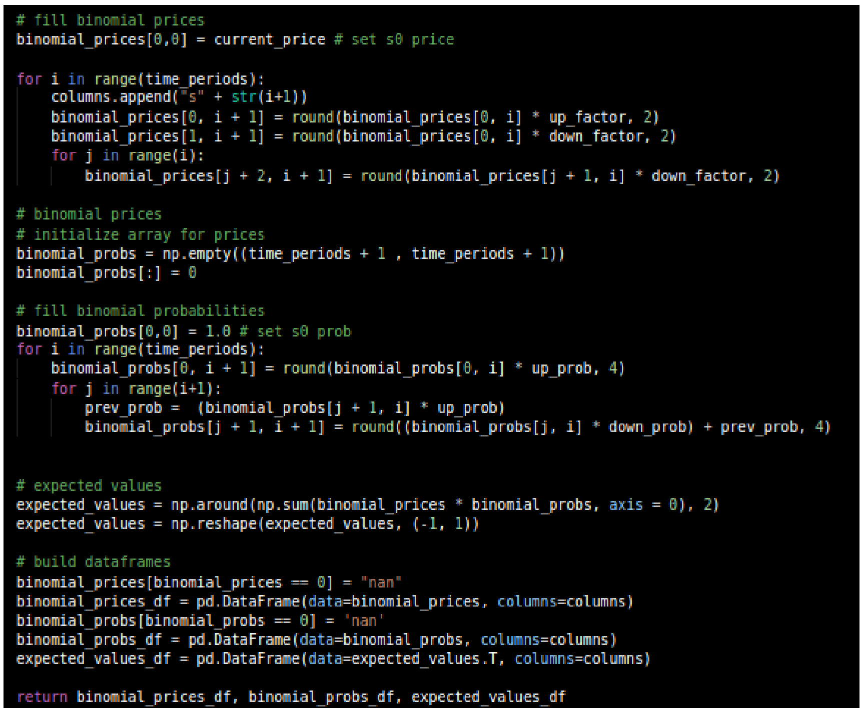
After these variables are computed an empty 2D numpy array was initialized to hold the projected prices for the binomial tree. The first element (0,0) was set to the last closing price. The function then loops for the number of time periods specified and computes the stock price binomial tree. After this iteration finishes, you will end up with a sparse matrix where half of the elements are empty, and the other half are filled with the prices. The symmetry between the prices and the empty values is diagonal. Fig. 1.22 better explains what is happening inside the for loop to compute the projected prices. Table. 1.1 shows the output for this part of the function, which is the projected stock price binomial trees.

Similarly to the prices, another empty 2D numpy array was initialized to hold the projected probabilities. The first element (0,0) was set to 1 since this value corresponds to the last closing price. Again, this function loops for the number of time periods specified to compute the probabilities. The output is quite like the previous step as it is also outputs a sparse matrix, where half of the elements are empty, and the other half are filled the probabilities. The symmetry between the probabilities and the empty values is diagonal. Fig. 1.23 better explains what is happening inside the for loop to compute the projected probabilities. Table. 1.2 shows the output for this part of the function, which is the projected probability binomial trees.



```python
def binomial_tree(expected_return, expected_volatility, current_price, simulation_time, time_periods, verbose=True):
    """ Constructs a binomial tree given the specified inputs

        Args:
            expected_return (float): The expected return for the asset.
            expected_volatility (float): The expected volatility for the asset.
            current_price (float): The current price for the asset.
            simulation_time (float): The simulation time, 1 = 1yr, 0.5 = 1/2yr and so on.
            time_periods (int): The number of time periods used to project the prices and probability.
            verbose (bool): If set to True outputs values during the operation, False will ignore this. By Default set to True.

        Returns:
            dataframe: Dataframe with the projected prices binomial trees.
            dataframe: Dataframe with the projected probability binomial trees.
            dataframe: Dataframe with the expected stock prices on the time periods specified.
    """

    delta_t = simulation_time / time_periods
    up_factor = np.exp(expected_volatility/100 * np.sqrt(delta_t))
    down_factor = 1 / up_factor
    up_prob = (np.exp(expected_return/100 * delta_t) - down_factor) / (up_factor - down_factor)
    down_prob = 1 - up_prob
    if verbose:
        print("Current price: {0}".format(current_price) +
              "\nExpected return: {0}%".format(expected_return) +
              "\nExpected volatility: {0}%".format(expected_volatility) +
              "\n\nSimulation time: {0}".format(simulation_time) +
              "\nTime periods: {0}".format(time_periods) +
              "\nDelt_t: {0}".format(delta_t) +
              "\nUp-factor: {0}".format(up_factor) +
              "\nDown-factor: {0}".format(down_factor) +
              "\nUp-probability: {0}".format(up_prob) +
              "\nDown-probability: {0}".format(down_prob))

    # columns for data frame
    columns = ["s0"]

    # binomial prices
    # initialize array for prices
    binomial_prices = np.empty((time_periods + 1 , time_periods + 1))
    binomial_prices[:] = 0

    # fill binomial prices
    binomial_prices[0,0] = current_price # set s0 price
```

Figure 1.20: Function to construct Binomial Tree part 1.



```
# fill binomial prices
binomial_prices[0,0] = current_price # set s0 price

for i in range(time_periods):
    columns.append("s" + str(i+1))
    binomial_prices[0, i + 1] = round(binomial_prices[0, i] * up_factor, 2)
    binomial_prices[1, i + 1] = round(binomial_prices[0, i] * down_factor, 2)
    for j in range(i):
        binomial_prices[j + 2, i + 1] = round(binomial_prices[j + 1, i] * down_factor, 2)

# binomial prices
# initialize array for prices
binomial_probs = np.empty((time_periods + 1 , time_periods + 1))
binomial_probs[:] = 0

# fill binomial probabilities
binomial_probs[0,0] = 1.0 # set s0 prob
for i in range(time_periods):
    binomial_probs[0, i + 1] = round(binomial_probs[0, i] * up_prob, 4)
    for j in range(i+1):
        prev_prob =  (binomial_probs[j + 1, i] * up_prob)
        binomial_probs[j + 1, i + 1] = round((binomial_probs[j, i] * down_prob) + prev_prob, 4)


# expected values
expected_values = np.around(np.sum(binomial_prices * binomial_probs, axis = 0), 2)
expected_values = np.reshape(expected_values, (-1, 1))

# build dataframes
binomial_prices[binomial_prices == 0] = "nan"
binomial_prices_df = pd.DataFrame(data=binomial_prices, columns=columns)
binomial_probs[binomial_probs == 0] = 'nan'
binomial_probs_df = pd.DataFrame(data=binomial_probs, columns=columns)
expected_values_df = pd.DataFrame(data=expected_values.T, columns=columns)

return binomial_prices_df, binomial_probs_df, expected_values_df
```

Figure 1.21: Function to construct Binomial Tree part 2.

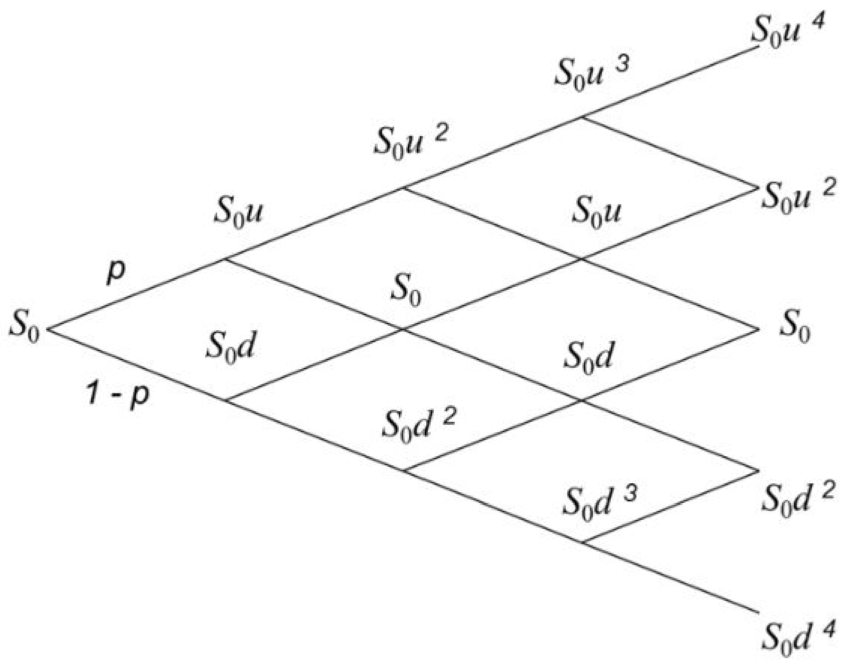
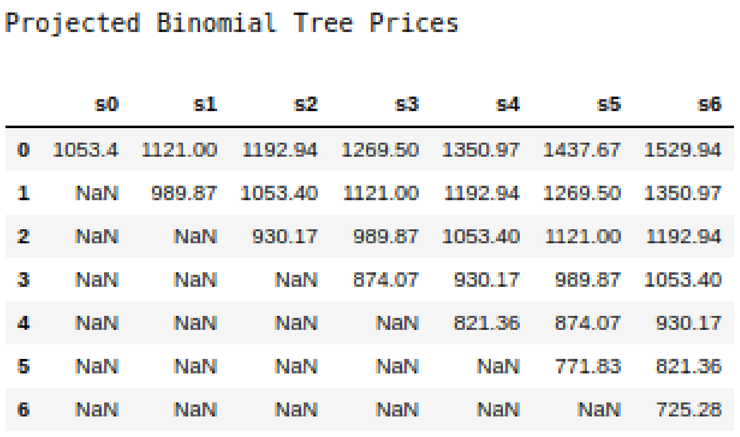


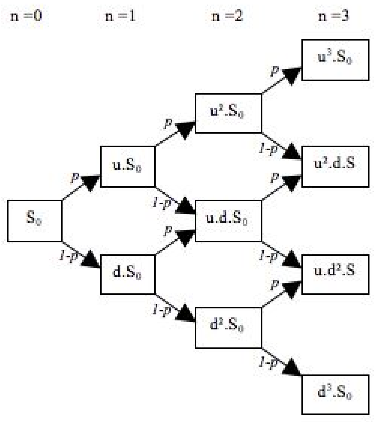
Figure 1.22: Visual Explanation of how the projected prices are computed. u = Up Factor and d = Down Factor



```
Projected Binomial Tree Prices
```

	s0	s1	s2	s3	s4	s5	s6
0	1053.4	1121.00	1192.94	1269.50	1350.97	1437.67	1529.94
1	NaN	989.87	1053.40	1121.00	1192.94	1269.50	1350.97
2	NaN	NaN	930.17	989.87	1053.40	1121.00	1192.94
3	NaN	NaN	NaN	874.07	930.17	989.87	1053.40
4	NaN	NaN	NaN	NaN	821.36	874.07	930.17
5	NaN	NaN	NaN	NaN	NaN	771.83	821.36
6	NaN	NaN	NaN	NaN	NaN	NaN	725.28

Table 1.1: The projected Stock Price Binomial Trees.

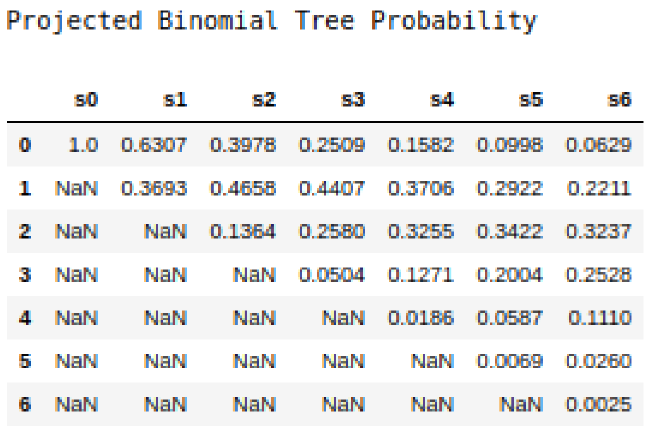


$$p = \frac{e^{rt/n} - d}{u - d}$$

$$u = e^{\sigma\sqrt{t/n}}$$

$$d = e^{-\sigma\sqrt{t/n}}$$

Figure 1.23: Visual Explanation of how the projected probabilities are computed.



```
Projected Binomial Tree Probability
```

	s0	s1	s2	s3	s4	s5	s6
0	1.0	0.6307	0.3978	0.2509	0.1582	0.0998	0.0629
1	NaN	0.3693	0.4658	0.4407	0.3706	0.2922	0.2211
2	NaN	NaN	0.1364	0.2580	0.3255	0.3422	0.3237
3	NaN	NaN	NaN	0.0504	0.1271	0.2004	0.2528
4	NaN	NaN	NaN	NaN	0.0186	0.0587	0.1110
5	NaN	NaN	NaN	NaN	NaN	0.0069	0.0260
6	NaN	NaN	NaN	NaN	NaN	NaN	0.0025

Table 1.2: The projected Probability Binomial Trees.

The final step was then to compute the expected values which will be shown in (iii). The expected values are computed as the sum product of each column of the two outputs described above. All the outputted numpy arrays described above were converted and returned as pandas dataframes.

1.3.2 Q3 (ii)

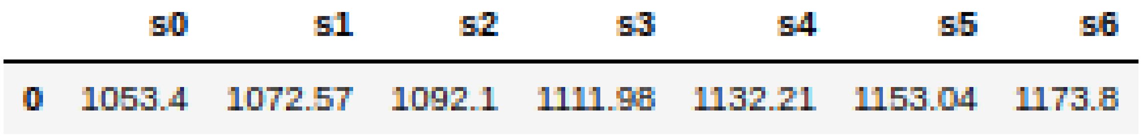
Utilizing the binomial tree from (i), what is the probability that at the end of the 6 month period the price will be greater than the starting price.

The code for this task can be found in 'Question 3 (ii)' in the python notebook. To calculate the probability for this question we took the sum of the last column for Table. 1.2 where the corresponding price is greater than the last closing price. This means $p = (0.0629 + 0.2211 + 0.3237) \times 100$, which is equal to 60.77%.

1.3.3 Q3 (iii)

Calculate the expected stock price, on a monthly basis.

The computation to get the stock prices on a monthly basis was described in the function which was used in (i), and these prices are shown in Table. 1.3. Code for this question can be found in 'Question 3 (iii)' in the python notebook.



	s0	s1	s2	s3	s4	s5	s6
0	1053.4	1072.57	1092.1	1111.98	1132.21	1153.04	1173.8

Table 1.3: Table for the expected stock prices on a monthly basis using Binomial Trees.

Chapter 2

Assignment – Part 2

The code used to answer the following questions can be found in the *'fintech_part_2.ipynb'* which is an IPython notebook. The notebook is in the following path *'src/fintech_part_2.ipynb'*. Also, the notebook references the 'fintech' library which was created for the purpose of this assignment. The library can be found in *'/src'*. All the code is well commented and easy to follow with.

2.1 Question 1

Consider an investment portfolio consisting of \$100K in Aluminium and \$400K in Zinc. The daily volatility of Aluminium is 0.70% and for Zinc is 0.20%. The correlation coefficient between them is 0.70.

2.1.1 Q1 (i)

Calculate the 15-day, 99% Value at Risk (VaR) of the portfolio.

Code to calculate the 15-day, 99% Value at Risk (VaR) of the portfolio for this question can be found in 'Question 1 (i)' in the python notebook.

The 15-day VaR(99%) for the portfolio is \$12,484.54 with a portfolio volatility of \$5,358.17.

2.1.2 Q1 (ii)

Comment on the impact of diversification on the portfolio VaR (what would be the VaR if the total \$500K is invested in only Aluminium and Zinc?).

After answering question (i) the VaR(99%) was calculated as if the full \$500K was to be invested in Aluminium only. The 15-day VaR(99%) for this asset would be ($2.33 \times 0.007 \times$

$\sqrt{15} \times 500K$) which is equal to \$31,584.18. On the other hand, if the \$500K was invested in Zinc only the 15-day VaR(99%) would be ($2.33 \times 0.002 \times \sqrt{15} \times 500K$) which is equal to \$9024.05.

With a a correlation of 0.70 between Aluminium and Zinc, both assets are likely to move in the same direction which means if one increases or decreases in price the other is likely to do so, since they are positively correlated. This correlation alone already indicates that the diversification in this portfolio is not ideal, since the reason of diversification is to invest in non-correlated assets. In fact although the VaR(99%) of the portfolio (\$12,484.54) is less than the VaR(99%) when investing in Aluminium only (\$31,584.18), it is still higher than the VaR(99%) when investing in Zinc only (\$9024.05).

So this indicates that using diversification in this case in not beneficial since the reason for diversification is to decrease risk. If both assets were less correlated, say with a 0.1 correlation the diversification for this portfolio would be beneficial as the VaR(99%) for the portfolio would be \$7,877.33 but in this specific case both assets have a high positive correlation of 0.70 which makes diversification not beneficial.

2.2 Question 2

In this question we will analyse Apple Stock.

2.2.1 Q2 (i)

Download closing prices for Apple for the period 01/01/2013 and 30/11/2017. Apply daily log returns and measure the average daily return.

Code to download the closing prices for Apple can be found in 'Question 2 (i)' in the python notebook. The data was downloaded from Yahoo Finance and saved as .csv format in the following path *'src/data/part_2'*. The file was saved as *'AAPL.csv'*. The data was then reloaded from the file and converted into pandas dataframe.

After applying the log returns the measure of the average daily log returns was 0.091%.

2.2.2 Q2 (ii)

We wish to measure volatility. Compare and discuss pros and cons of utilising the Historical method, Exponentially Moving Average method or Garch.

Historical Method

To measure volatility using the historical method, first the historic data must be obtained. Once the data is obtained the log returns must be calculated and then the standard deviation is calculated on the log returns. In this model when higher frequency data is used the sum of the squared returns is used instead. In this method all past data is given an equal weight (no weight at all).

Pros:

- Faster to compute and easy to implement.
- Does not require any parameters.
- No model risk since no parameters are used.

Cons:

- Does not assign any weights and all values are given the same importance.
- Less accurate since it does not give importance to recent values.
- Data is not utilised properly (most of the data is not used).

EWMA

Unlike the historical method, the exponentially weighted moving average method does not give equal weights to the data points. This is done as the recent square returns should give a better indication for volatility. In this method an exponentially decreasing weight is set to the squared returns (starts from the most recent data point).

Pros:

- Assigns exponentially decreasing weight to the squared returns. Recent returns better describe volatility.
- Less parameters than the GARCH model, thus less Model Risk.

Cons:

- This model do not handle mean reversion (no term used for mean reversion).
- Does not capture trend information.
- Does not forecast volatility.
- A volatility shock is suddenly dropped from the computation as the trailing window of observations pass.

Garch

The Generalized ARCH model is an extension to the ARCH model. This model also uses weights, but the weights are assigned to the following three components; lagged variance, lagged squared return and the long-run variance. The sum of the weights assigned to the first two components is known as persistence. Commonly this model is solved using the maximum likelihood estimation technique.

Pros:

- More weight on recent information.
- Can output more accurate estimations.
- A term is used to handle mean reversion.
- Can be used to forecast volatility.
- Overcomes Ghosting as a shock in volatility will be immediately impact the estimates (Fades as time passes).

Cons:

- This models requires more parameters and has a greater Model Risk.

- More computationally expensive.

Comparing the three models, the first model does not assign weights to the log returns, while the EWMA uses exponentially decreasing weights to the squared returns. On the other hand, the GARCH models' assigns weights to the following components lagged variance, lagged squared returns and long run variance. Both the GARCH and EWMA work with conditional variance but the other model works with realized variance. Also both GARCH and EWMA apply exponential smoothing to their weights. Unlike GARCH the EWMA does not have a term for mean reversion. Both GARCH and EWMA are recursive and parametric.

2.2.3 Q2 (iii)

Use Garch(1,1) to measure volatility based on the data from (i).

The measurement for the annualized volatility for Apple when using Garch(1,1) is 22.46%.

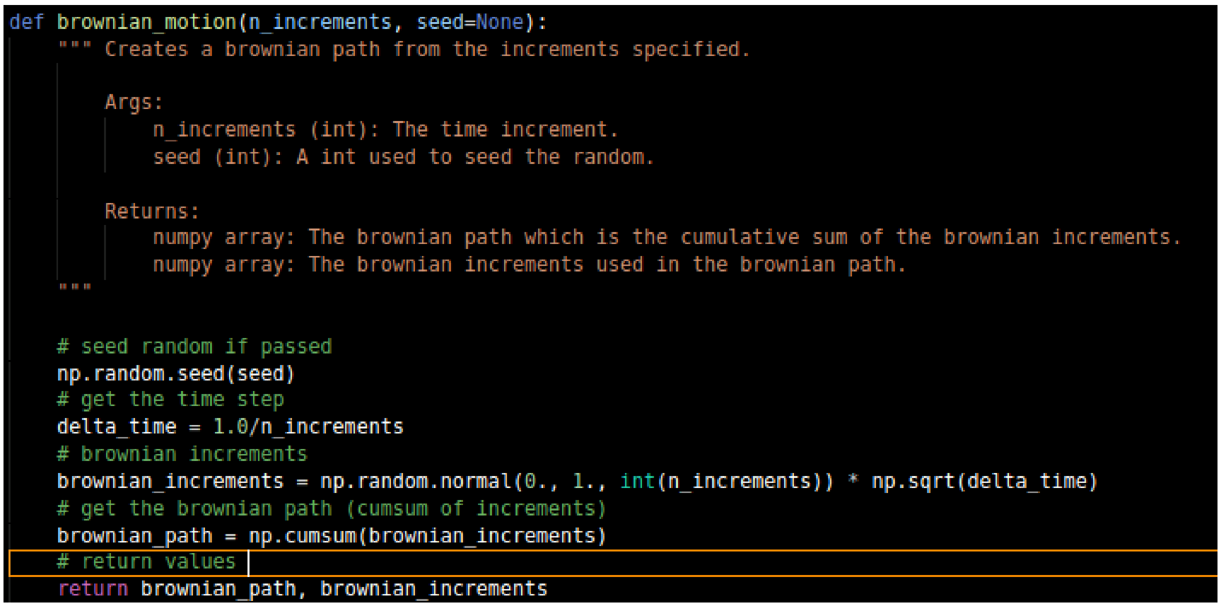
Volatility was computed using the code found in 'Question 2 (iii)' in the python notebook and this notebook cell utilises a function called 'arch_vol' found in the 'fintech' library. Such function uses a third party library called 'arch' [5] which provides different volatility models written in python.

2.2.4 Q2 (iv)

Assume that Apple stock prices follow a Geometric Brownian stochastic process. Utilising the average return and volatility statistics identified in part (i) and (iii), simulate, on a day by day basis, the movement of Apple stock for the next 20 days (i.e. starting from 1st December, as per date range in part (i)). Repeat this process for 100 times, hence creating 100 possible price realisations over the next 20 days. Describe the steps/approach used and present a plot showing the projected 100 realisations.

First the current price was referenced, which is the adjusted close price for the 1st of December. The expected volatility was computed by using the annualized volatility found in (iii) and scaling it to get the expected volatility for the next day. This was done using the following computation (assuming a year has 250 days) ($Annualized\ Volatility \times \sqrt{\frac{1}{250}}$).

Now that we have the expected return and expected volatility, an iteration to create different projections was written. The iteration runs for 100 times, since we need to create 100 possible price realisations. In each iteration a new Brownian path is generated using a function which was created in the 'fintech' library called 'brownian_motion' as shown in Fig. 2.1. When calling this function, the number of increments is passed as a parameter which in our case is 20 since we are projecting for the next 20 days.



```python
def brownian_motion(n_increments, seed=None):
    """ Creates a brownian path from the increments specified.

        Args:
            n_increments (int): The time increment.
            seed (int): A int used to seed the random.

        Returns:
            numpy array: The brownian path which is the cumulative sum of the brownian increments.
            numpy array: The brownian increments used in the brownian path.
    """

    # seed random if passed
    np.random.seed(seed)
    # get the time step
    delta_time = 1.0/n_increments
    # brownian increments
    brownian_increments = np.random.normal(0., 1., int(n_increments)) * np.sqrt(delta_time)
    # get the brownian path (cumsum of increments)
    brownian_path = np.cumsum(brownian_increments)
    # return values
    return brownian_path, brownian_increments
```

Figure 2.1: Function to create Brownian path.

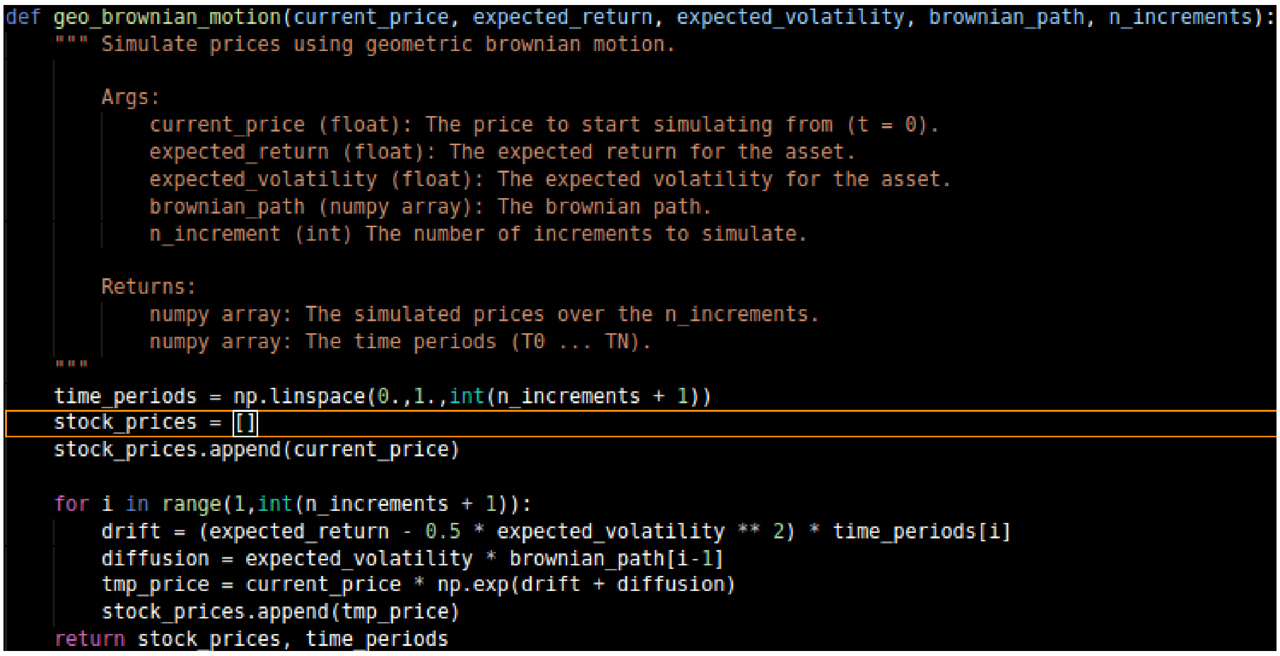
In this function the Brownian increments W_i are computed by getting z_i which is a random variable selected from a normal distribution with $\mu = 0$ and $\sigma = 1$ and multiplying it by $\sqrt{t_i}$ where t_i is the time increment, as shown in Eq. 2.1.

$$W_i = z_i\sqrt{\Delta t_i} \tag{2.1}$$

After computing the Brownian increments, the Brownian path is computed by getting the cumulative sum of the Brownian increments. This is shown in Eq. 2.2, and the 'brownian_motion' function returns both the path and the increments.

$$W_n(t) = \sum_{i=1}^{n} W_i(t) \tag{2.2}$$

In each iteration when a new Brownian path is generated, the path is used as a parameter in a function called 'geo_brownian_motion' which is also found in the 'fintech' library and this function is shown in Fig. 2.2. The returned stock prices and time periods were used to plot the possible price realisations. Each price at t is computed using Eq. 2.3.

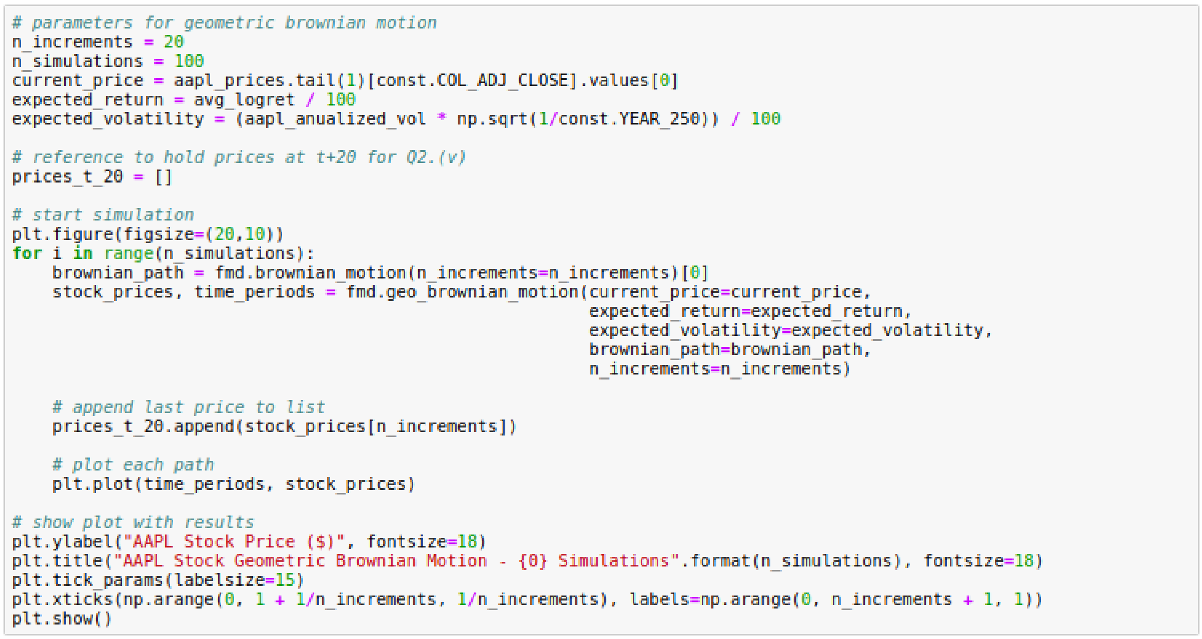


```python
def geo_brownian_motion(current_price, expected_return, expected_volatility, brownian_path, n_increments):
    """ Simulate prices using geometric brownian motion.

        Args:
            current_price (float): The price to start simulating from (t = 0).
            expected_return (float): The expected return for the asset.
            expected_volatility (float): The expected volatility for the asset.
            brownian_path (numpy array): The brownian path.
            n_increment (int) The number of increments to simulate.

        Returns:
            numpy array: The simulated prices over the n_increments.
            numpy array: The time periods (T0 ... TN).
    """
    time_periods = np.linspace(0.,1.,int(n_increments + 1))
    stock_prices = []
    stock_prices.append(current_price)

    for i in range(1,int(n_increments + 1)):
        drift = (expected_return - 0.5 * expected_volatility ** 2) * time_periods[i]
        diffusion = expected_volatility * brownian_path[i-1]
        tmp_price = current_price * np.exp(drift + diffusion)
        stock_prices.append(tmp_price)
    return stock_prices, time_periods
```

Figure 2.2: Function to create Geometric Brownian Motion.

$$S_{t+\Delta t} = S_t \exp[(\mu - \frac{\sigma^2}{2}) + \Delta t + \sigma\epsilon\sqrt{\Delta t}] \tag{2.3}$$

These functions were then called in the notebook in 'Question 2 (iv)' as shown in Fig. 2.3 and the final output was plotted as shown in Fig.2.4.



```python
# parameters for geometric brownian motion
n_increments = 20
n_simulations = 100
current_price = aapl_prices.tail(1)[const.COL_ADJ_CLOSE].values[0]
expected_return = avg_logret / 100
expected_volatility = (aapl_anualized_vol * np.sqrt(1/const.YEAR_250)) / 100

# reference to hold prices at t+20 for Q2.(v)
prices_t_20 = []

# start simulation
plt.figure(figsize=(20,10))
for i in range(n_simulations):
    brownian_path = fmd.brownian_motion(n_increments=n_increments)[0]
    stock_prices, time_periods = fmd.geo_brownian_motion(current_price=current_price,
                                                         expected_return=expected_return,
                                                         expected_volatility=expected_volatility,
                                                         brownian_path=brownian_path,
                                                         n_increments=n_increments)

    # append last price to list
    prices_t_20.append(stock_prices[n_increments])

    # plot each path
    plt.plot(time_periods, stock_prices)

# show plot with results
plt.ylabel("AAPL Stock Price ($)", fontsize=18)
plt.title("AAPL Stock Geometric Brownian Motion - {0} Simulations".format(n_simulations), fontsize=18)
plt.tick_params(labelsize=15)
plt.xticks(np.arange(0, 1 + 1/n_increments, 1/n_increments), labels=np.arange(0, n_increments + 1, 1))
plt.show()
```

Figure 2.3: Code for simulation using Geometric Brownian path.

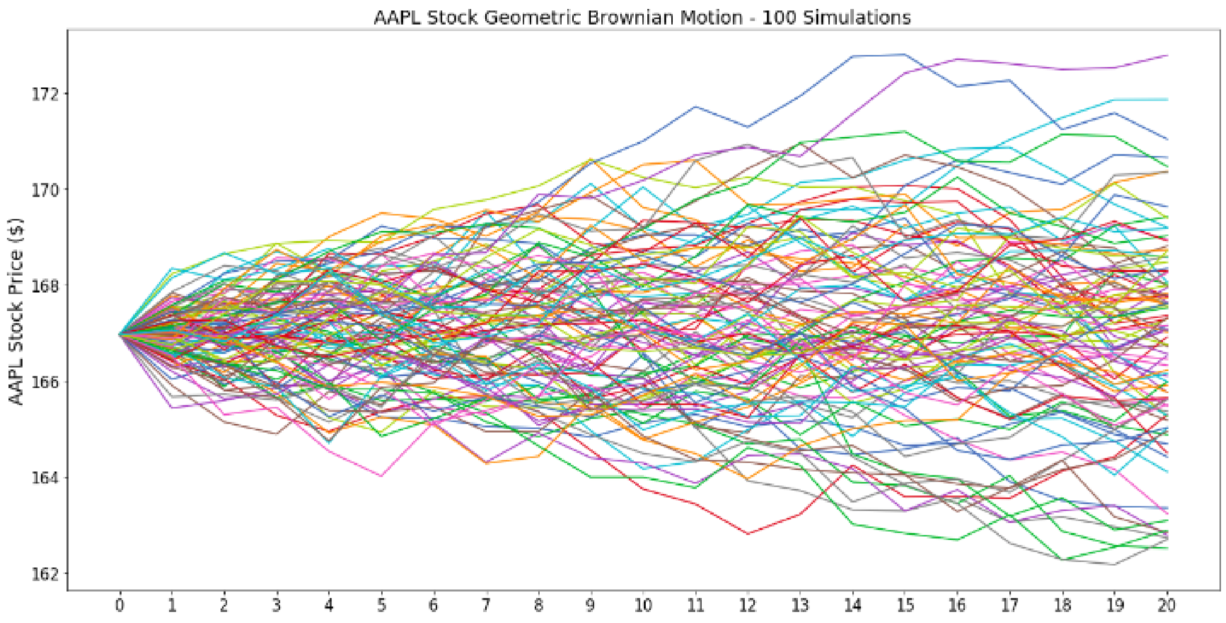


Figure 2.4: 100 price realisations over the next 20 days using Geometric Brownian Motion.

2.2.5 Q2 (v)

Using the prices obtained at the end of each simulated realisation, i.e. simulated prices at t+20 days, calculate the 20 day Value at Risk (VaR) at 95% confidence level. Hint: You can use the percentile approach on the 20 day returns.

The 20 day VaR(95%) based on the simulation for question (iv) is \$3.40. Code for this measurement is shown in 'Question 2 (v)' in the notebook.

2.3 Question 3

Hidden Markov Models (HMM) are an important unsupervised learning method used to analyse sequence data and identifying underlying regimes that govern the data.

This question relates to detecting regime changes in volatility (risk).

2.3.1 Q2 (i)

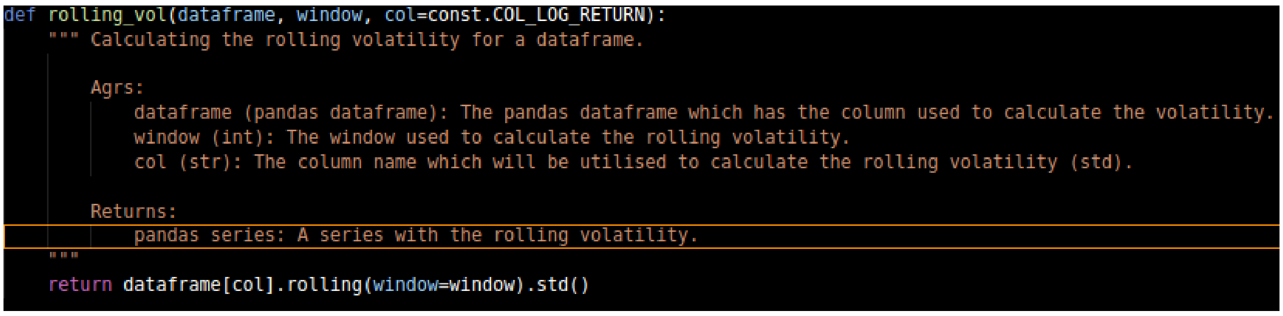
Download and calculate the last 10 year daily returns of the S&P500 index (till 31/12/2017).

Code to download the closing prices for S&P500 in this question can be found in ‘Question 3 (i)’ in the python notebook. The data was downloaded from Yahoo Finance and saved as .csv format in the following path *‘src/data/part_2’*. The file was saved as *‘GSPC.csv’*. The data was then reloaded from the file and converted into pandas dataframe and the daily log returns were computed.

2.3.2 Q2 (ii)

Create a new series of volatility based on 10 day rolling time window (for each day apply standard deviation on the previous 10 day returns).

To calculate the 10 day rolling volatility a function called ’rolling_vol’ was created in the ’fintech’ library as shown in Fig. 2.5, and this was called from the notebook in ’Question 3 (ii)’.



```
def rolling_vol(dataframe, window, col=const.COL_LOG_RETURN):
    """ Calculating the rolling volatility for a dataframe.

        Agrs:
            dataframe (pandas dataframe): The pandas dataframe which has the column used to calculate the volatility.
            window (int): The window used to calculate the rolling volatility.
            col (str): The column name which will be utilised to calculate the rolling volatility (std).

        Returns:
            pandas series: A series with the rolling volatility.
    """
    return dataframe[col].rolling(window=window).std()
```

Figure 2.5: Function to calculate rolling volatility.

2.3.3 Q2 (iii)

Utilize this new series to train two Gaussian HMM: one using 2 latent states and the other with 3 latent states (components). These latent states indicate the different volatility regimes.

The code to train the two models can be found in 'Question 3 (iii)'. Both models were trained using 1000 iteration using a diagonal covariance matrix. Also both models were trained using one feature, which is the series created in (ii). A third party library called 'hmmlearn' [6] was utilised for this task.

2.3.4 Q2 (iv)

For each of the two models above, present a visualization (one or more plots) which clearly shows the identified volatility regimes over the 10-year period being investigated.

Fig. 2.6 shows the volatility regime over the 10-period when using 2 latent states while Fig. 2.7 shows the volatility regimes when using 3 latent states. The code for these plots can be found in the notebook in 'Question 3 (iv)'.

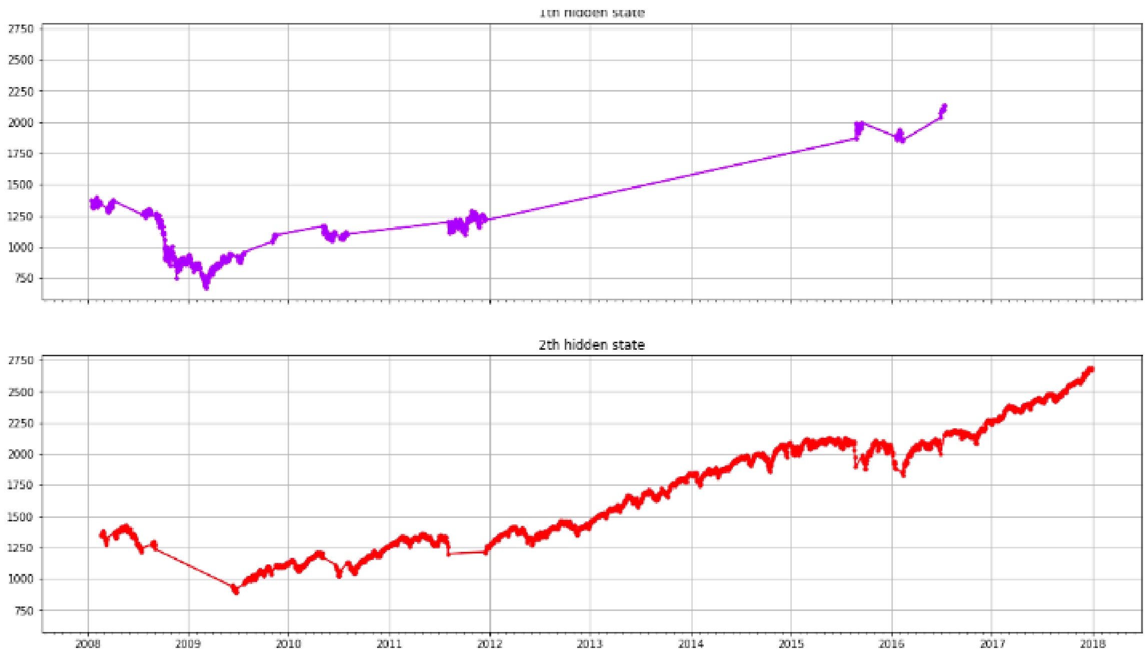


Figure 2.6: Volatility regime over the 10-period when using 2 latent states.

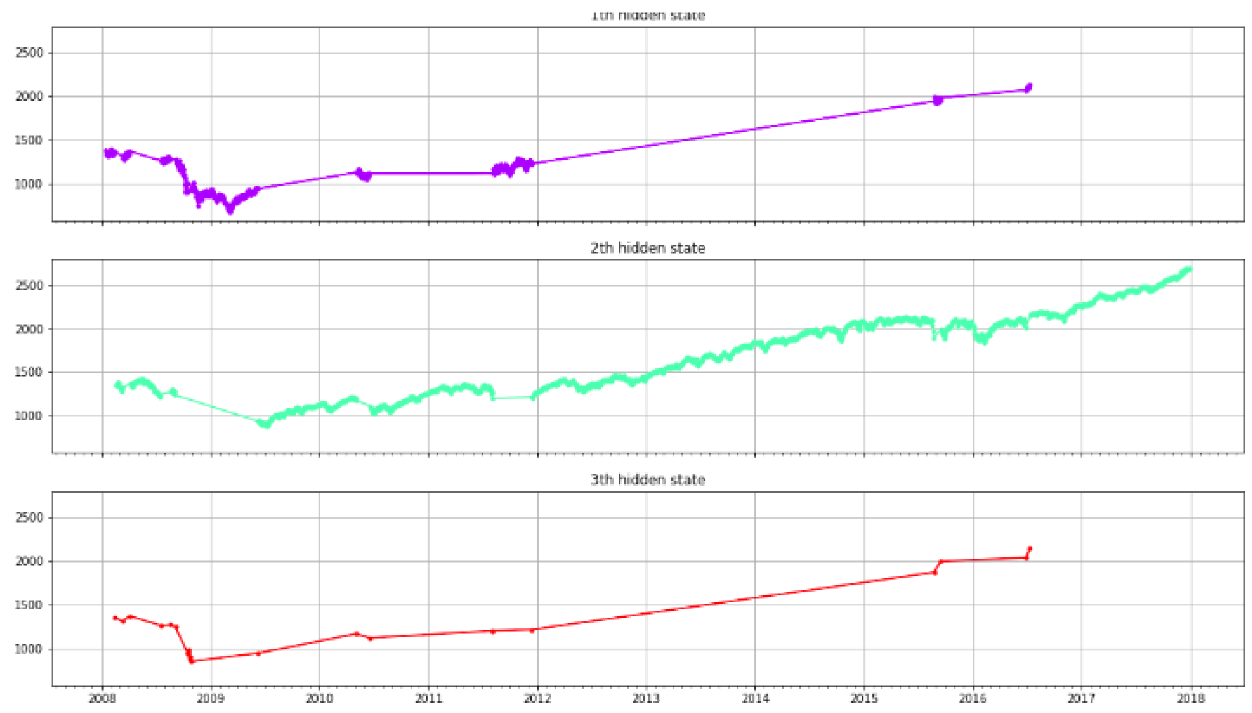


Figure 2.7: Volatility regime over the 10-period when using 3 latent states.

2.3.5 Q2 (v)

Comment on the findings and discuss the difference in outcomes between the two models.

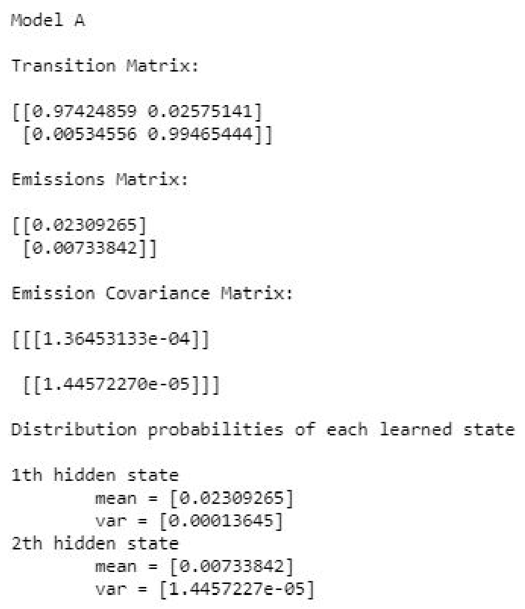
Using the models in the previous question (iii) the Hidden Markov model enables us to investigate underlying latent states (must specify the number of components in this unsupervised learning model) and the probability transitions between them even though they are not directly observable.

Finding volatility regimes can help an investor to manage risk more effectively by identifying states where there is high volatility. This model has the potential of identifying incorrect identification of a “trend”.

As seen from both the plots the regime detection captures highly volatile and “trending” periods. In the first model (two latent states) the majority is captured in the 2nd state but most of 2008-2009 and late 2011 is captured in the first state. In the second model the majority was captured in the 2nd state, while most of 2008-2009 was captured in the first state together with the late 2011 period. Utilizing these regimes one can apply different investing/risk management methods according to the state the asset is in, in terms of volatility. The states which were captured show a low volatility state and a high volatility state/s.

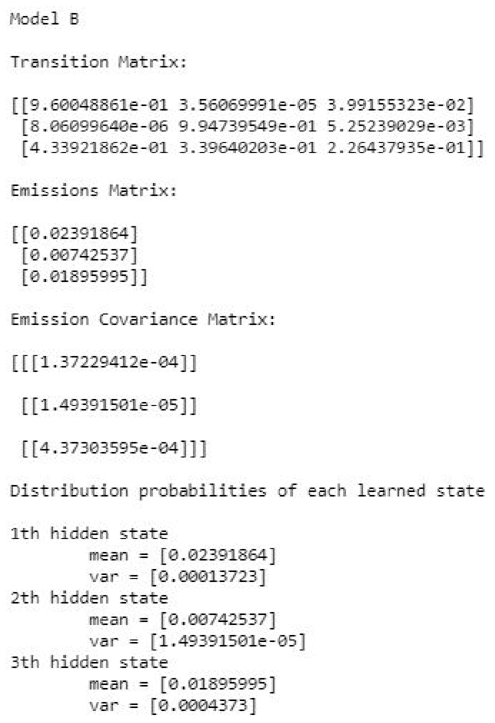
All the states in the two models have a positive mean. The variance of the 2nd state in both models indicate low variance (low volatility periods), but the first state has a variance which is significantly higher in both models (high volatility periods). This indicates a state with high volatility. The third state in the second model also indicate high volatility but it must be noted that this state captures relatively low information.

It is also important to note the states during the 2008 period (market crash), as you can see both models captured high volatility during that period (both models capture this in the first state). The measurements discussed above (mean and var) are shown in Fig. 2.8a and Fig. 2.8b.



```
Model A

Transition Matrix:

[[0.97424859 0.02575141]
 [0.00534556 0.99465444]]

Emissions Matrix:

[[0.02309265]
 [0.00733842]]

Emission Covariance Matrix:

[[[1.36453133e-04]]

 [[1.44572270e-05]]]

Distribution probabilities of each learned state

1th hidden state
        mean = [0.02309265]
        var = [0.00013645]
2th hidden state
        mean = [0.00733842]
        var = [1.4457227e-05]
```

(a) HMM measurements for Model A (two latent states).



```
Model B

Transition Matrix:

[[9.60048861e-01 3.56069991e-05 3.99155323e-02]
 [8.06099640e-06 9.94739549e-01 5.25239029e-03]
 [4.33921862e-01 3.39640203e-01 2.26437935e-01]]

Emissions Matrix:

[[0.02391864]
 [0.00742537]
 [0.01895995]]

Emission Covariance Matrix:

[[[1.37229412e-04]]

 [[1.49391501e-05]]

 [[4.37303595e-04]]]

Distribution probabilities of each learned state

1th hidden state
        mean = [0.02391864]
        var = [0.00013723]
2th hidden state
        mean = [0.00742537]
        var = [1.49391501e-05]
3th hidden state
        mean = [0.01895995]
        var = [0.0004373]
```

(b) HMM measurements for Model B (three latent states).

2.4 Question 4

Cryptocurrencies and blockchain are amongst the most currently discussed topics in the Fintech world.

2.4.1 Q4 (i)

Provide, technically feasible situations, where you as an expert would suggest the adoption of blockchain technology. In your answer explain the cons and pros of using blockchain viz-a-viz other commonly used architectures.

Although Blockchain is an innovative technology, it is not always feasible to implement every solution on to the chain. One of the main advantages of Blockchain is that it allows the use of decentralization meaning that you don't need a middleman/server (removing the dependency of a central server or database) to communicate/transact with and it allows a user to directly cut off the middleman. Another advantage is that some chains allow for code to hold value similarly to Ethereum's smart contracts. For the purposes of this question we will discuss feasible solutions which can be implemented on to the chain and we assume that the Ethereum's network will be used, since there are many variations which have different functionalities, but most of the features described here are implemented in other chains which allow programmability (Generalised Blockchains). Another great advantage when using the chain is that transactions are immutable meaning, once they are written, they cannot be changed and are permanently stored on the chain.

The following are some situations where we believe blockchain could be useful/feasible:

Registry Systems

An example where we think that Blockchain can be useful is in situations where someone needs to register something. For example, let's look at land and car registry. A system built using the Blockchain can allow for land registries or car registries to be saved on the chain. This will allow the ones using the system to view ownership, previous owners and property/car rights from data saved on the chain. Due to the immutability property of the Blockchain, data cannot be tempered with, which is very important in this situation.

Clients do not have to trust the ones who take care of the system, as in itself data cannot be changed. Compared with the traditional systems of using a database, where a server can be vulnerable to attack, and data can be changed for the benefit of an attacker. Such solution can help mitigate property fraud. Although this solution allows for immutability, which may be a great advantage, all the data saved on the chain is publicly available. Let's say that a property ownership is linked to an owner via an address, once someone can link this address to a person, they can easily know what property or car he/she owns or previously owned. The use of private chains can help with this situation, but again every node in this chain has the capability of doing the same thing described.

An example where ownership of a property is saved on the chain is 'CryptoKitties' [7] where users purchase, breed and collect virtual cats. So, similar applications where there is a need for ownership of a virtual property, such as game, can be implemented in the same manner. In the case of games there might be a problem with scalability as the Ethereum network can only process about 25 tx/s and you don't want to stop a player from purchasing a virtual property just because the system is under a heavy load.

Some other feasible situations which were not discussed: Money (cryptocurrencies), Exchanges, Crowd Sourcing, Markets and Automating regulatory compliance.

Overall the pros and cons described in the such situations above, can be generalized as follows: *Blockchain*

Pros:

- Trustless: No party needs to trust the other party.
- Immutability: Once the data is written it cannot be changed.
- Security against DoS attacks as it would be expensive to conduct such an attack.
- Decentralization: All of the nodes contain a copy of the chain, so if one node fails the system can still continue to run.

Cons:

- Cost: Each transaction comes with a fee (Miner Fees).
- Secrets: In most of the chains it is very difficult to store secrets without a third party interference.
- Scalability: The state of public Blockchains right now do not scale well (delays under massive amounts of transactions.)
- 51% Attacks: Public Blockchains using PoW can be targeted with colluding malicious miners, which can end up in 'Double Spending'.

Commonly used Architectures

Pros:

- Secrets: Can store secrets on the system without public access by anyone.
- Cost: A transaction does not require a cost (No Miner Fee).
- Time: Transactions do not need miners to be processed.
- Scalability: Most commonly used architectures are easily scalable (if written well).

Cons:

- Mutable: Data is mutable and can be tempered with.
- Attacks: Prone to DoS attacks.
- Trust: The client has to trust the service.
- One point of failure, if the system fails a service is unusable (although this is mitigated by having backup servers).

Before implementing a system using Blockchain we believe that the following questions must be answered before even considering it:

- Is it feasible to use a public database where multiple parties can access it ? (Yes)
- Is there a need for an intermediary or this could be cut entirely ? (Yes)
- Are the transactions done on the system dependent on each other ? (Yes)
- Is it a requirement for the system to have immutability ? (Yes)
- Can the system wait for the transactions to be processed ? (Yes)

2.4.2 Q4 (ii)

Explain proof-of-work and how the algorithm permits decentralised concensus.

PoW (proof-of-work) is the consensus algorithm used in the Bitcoin [8] network, although there are other networks which also use such algorithm, i.e the Ethereum network [9]. The PoW concept was introduced in HashCash were it was used to limit email spam [10] and DoS attacks [11] by making the processor use some of the processing power to compute a hashcash stamp to be added to the header of an email, so that the sender can prove that it did some work before sending an email.

In the case of cryptocurrencies, the algorithm is used to combat Byzantine Failures in a decentralized network and to reach consensus between the nodes connected to the network. This is a probabilistic approach to solve the Byzantine Generals Problem, a problem in decentralized networks described by Lamport [12]. Such algorithm is basically a cryptographic puzzle which each miner on the network is trying to solve. Now we will be describing how PoW consensus works in the Bitcoin network.

To correctly explain this mechanism, first we need to define some data structures used in the network, to be able to know what a block header is. Each block in the blockchain contains a data structure called a Merkle tree. This structure is used to hold the transactions in a specific block. For simplicity reasons let assume that the leaf node in the Merkle tree shown in Fig. 2.9 is a transaction. The parent node is than composed of two pairs of hashes generated from the leaf nodes. The parent of the parent of the leaf nodes is composed of two other pairs of hashes and this process keeps on going until you reach the root node which has the hash generated from this process.

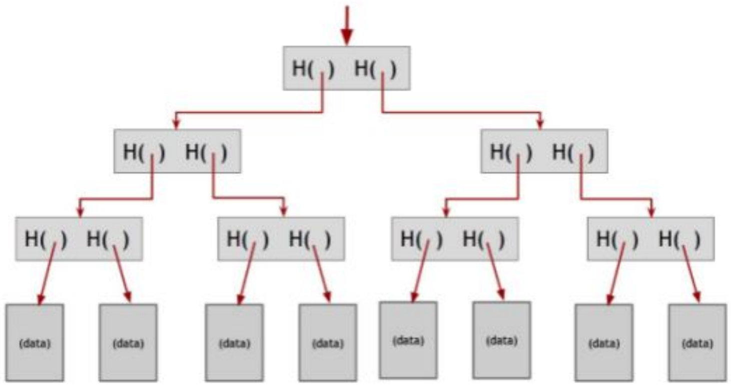


Figure 2.9: A visual representation of a Merkle tree.

In the block header you will have the following components; Bitcoin Version Number, Previous Block Header Hash, Hash of the root node for the Merkle Tree, The block timestamp (unix), Difficulty for the target block and the nonce. What happens in PoW is that miners use their computing power to find a nonce (a numeric value) which when hashed (SHA256) with the block header would create a value with leading zeros. This hash is valid if the value is below the target. The difficulty can be adjusted, in fact in the Bitcoin network it is adjusted every 2016 blocks.

Once a miner solves this puzzle the block is propagated through the network so other miners can start working on a new block. Since the blocks have a hash pointer to the previous block all blocks are dependent on the previous one and if you had to visualize this you will see a chain of blocks as shown in Fig. 2.10 hence the name blockchain. The more computation power a miner has the more nonce values this node can try. The mining process is also incentivised, meaning that when a miner solves the puzzle it will be rewarded with bitcoin (in fact this is how new bitcoin is created although the total supply is capped). The longest chain in the network is the chain which the nodes agree on, meaning the blocks (transactions) which are valid to the network. This mechanism also is way to try and handle the 'Double Spending Problem' described in Satoshi Nakamoto paper [8].

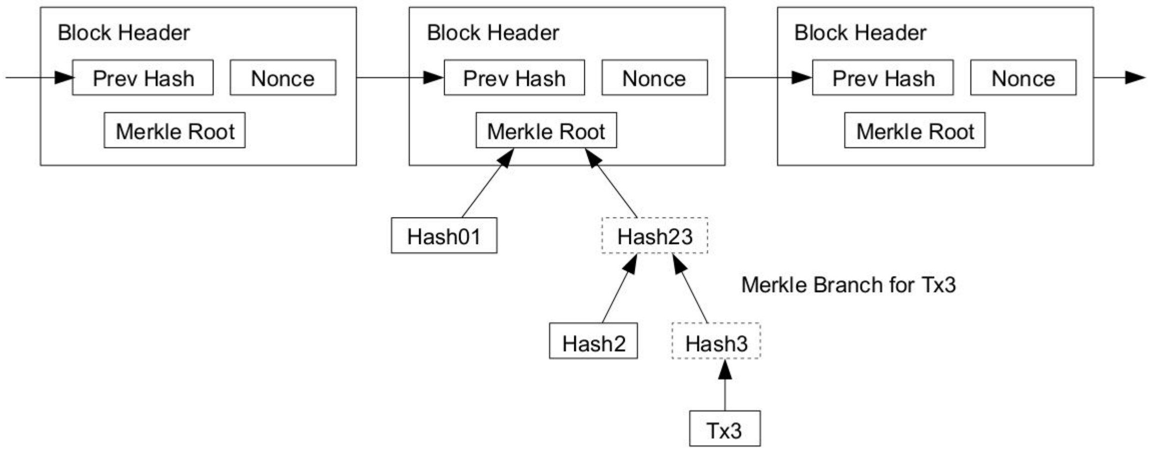


Figure 2.10: A visual representation of the Blockchain.

Such mechanism only works if most of the hashing power belongs to none malicious miners. When you have 51% of the miners who are colluding and acting maliciously it is guaranteed that they can control the network and can double spend their coins. In fact, this attack was done on the network of the cryptocurrency called 'Bitcoin Gold' were they doubled spent their value to defraud exchanges [13].

The PoW mechanism permits decentralise consensus for the following traits; Every full node can verify each transaction independently, each block which is propagated by the miner can be verified independently, transactions can be aggregated independently into new blocks (mining nodes) and a solution is provided if two miners propagate a block at the same time (longest chain) and every miner node can participate to take part of the mining process (although the use of mining pools goes against Satoshi's vision, since hashing power is becoming more centralized).

2.4.3 Q4 (iii)

Your group of friends happen to be football fanatics. You decide to utilise a smart contract to manage bets with your friends for an upcoming tournament. Each participant will need to guess the winner of the competition and is allowed to log his preferred team only once (1 bet allowed). The participants will have to place their bet before the first tournament game – Feb 1st, 2019 - beyond which the smart contract will not allow any further bets.

Each participant would need to put 10 Euro in the pot. To keep things simple, you decide that the pot will be managed externally using normal fiat money, however you believe that for transparency purposes a balance showing the total pot collected should be available on the contract.

Once the official winner is known, you as the organiser will update the contract. The contract should not permit setting the winner before end of competition - 28th of Feb, 2019. Each participant can query the contract and check the amount won by him (if any), depending on (i) whether he guessed or not and (ii) whether he was the only one guessing the winning team (if not, then the total money won will be Total pot divided by total number of winning participants).

Assume that teams are numbered 1-10 (assuming 10 teams).

Create a smart contract that can manage this use case. Provide source code of the contract with explanations of how the problem is approached.

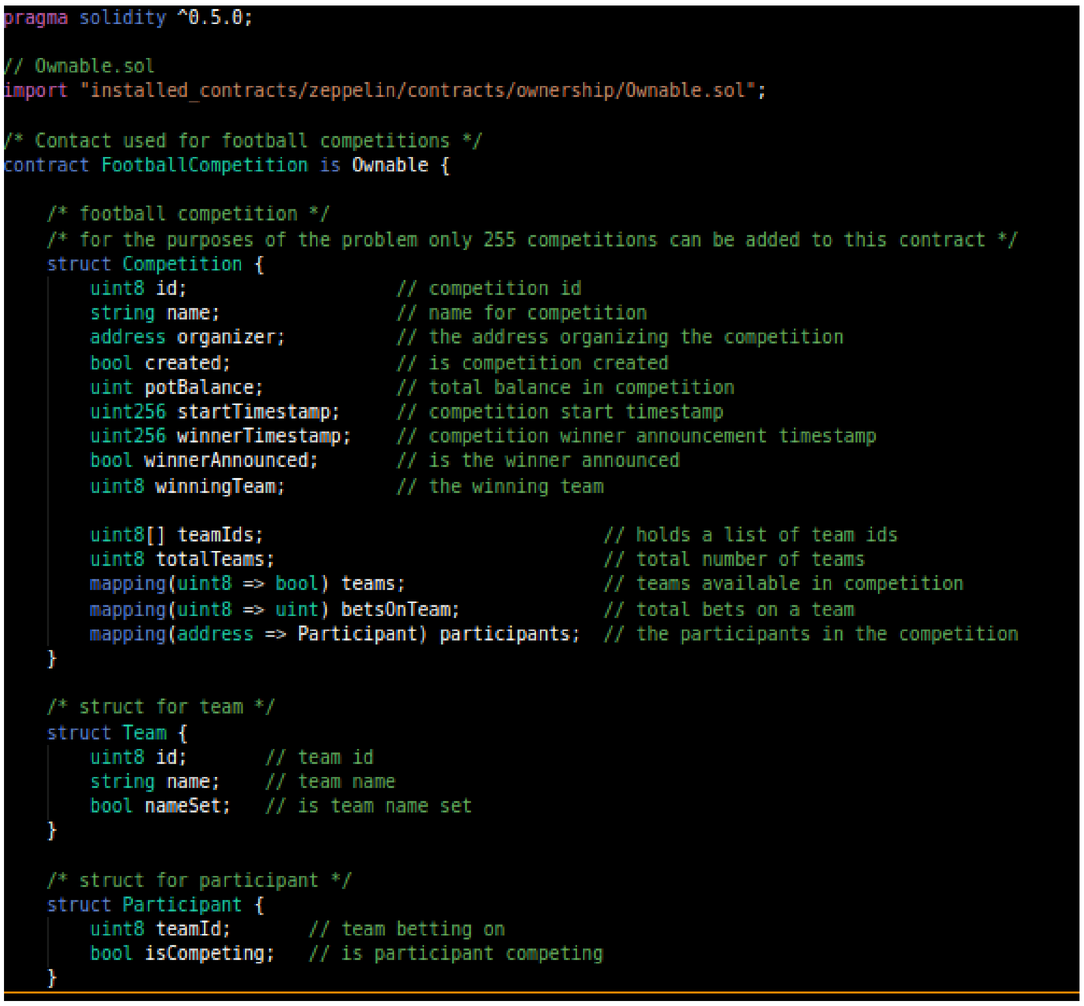
Code for this task is in *'src/solidity/football_competition_truffle'*. Inside the location provided there is a folder called *'contracts'* were the smart contract called 'FootballCompetition.sol' is used to manage this use case. Also, there is a folder called *'python'* which has a script called 'chain_comp.py' which is used to test the contract which was written.

Let's explain the code inside the 'FootballCompetition.sol'. First off, the contract inherits from an open source contract called 'Ownable.sol' from OpenZepplin [14] which basically is a contract which gives some functionality to manage the owner modifier for the contract.

This contract gives us functionality such as checking that the owner of the contract can only access a specific function and some other functions such as transferring ownership to another address. The 'Ownable.sol' is located in:

'src/solidity/football_competition_truffle/installed_contracts/zepplin/contracts/ownership'.

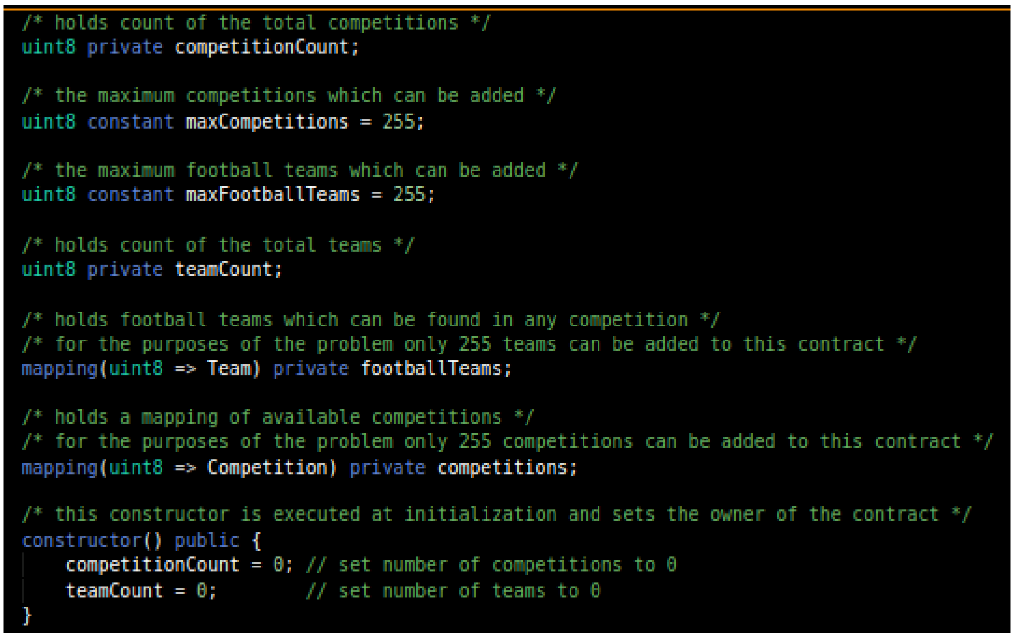
After doing so three different structs were written which will be utilised later in the contract. These can be shown in Fig. 2.11. All variables in the structs are commented so it is self-explanatory for what the variables will be used for. Also, the contract was written in a way that there could be multiple competitions created by different organizers/addresses, this was done to make the use-case more dynamic and can accept multiple competitions.



```
pragma solidity ^0.5.0;

// Ownable.sol
import "installed_contracts/zeppelin/contracts/ownership/Ownable.sol";

/* Contact used for football competitions */
contract FootballCompetition is Ownable {

    /* football competition */
    /* for the purposes of the problem only 255 competitions can be added to this contract */
    struct Competition {
        uint8 id;                   // competition id
        string name;                // name for competition
        address organizer;          // the address organizing the competition
        bool created;               // is competition created
        uint potBalance;            // total balance in competition
        uint256 startTimestamp;     // competition start timestamp
        uint256 winnerTimestamp;    // competition winner announcement timestamp
        bool winnerAnnounced;       // is the winner announced
        uint8 winningTeam;          // the winning team

        uint8[] teamIds;                               // holds a list of team ids
        uint8 totalTeams;                              // total number of teams
        mapping(uint8 => bool) teams;                  // teams available in competition
        mapping(uint8 => uint) betsOnTeam;             // total bets on a team
        mapping(address => Participant) participants;  // the participants in the competition
    }

    /* struct for team */
    struct Team {
        uint8 id;       // team id
        string name;    // team name
        bool nameSet;   // is team name set
    }

    /* struct for participant */
    struct Participant {
        uint8 teamId;       // team betting on
        bool isCompeting;   // is participant competing
    }
```

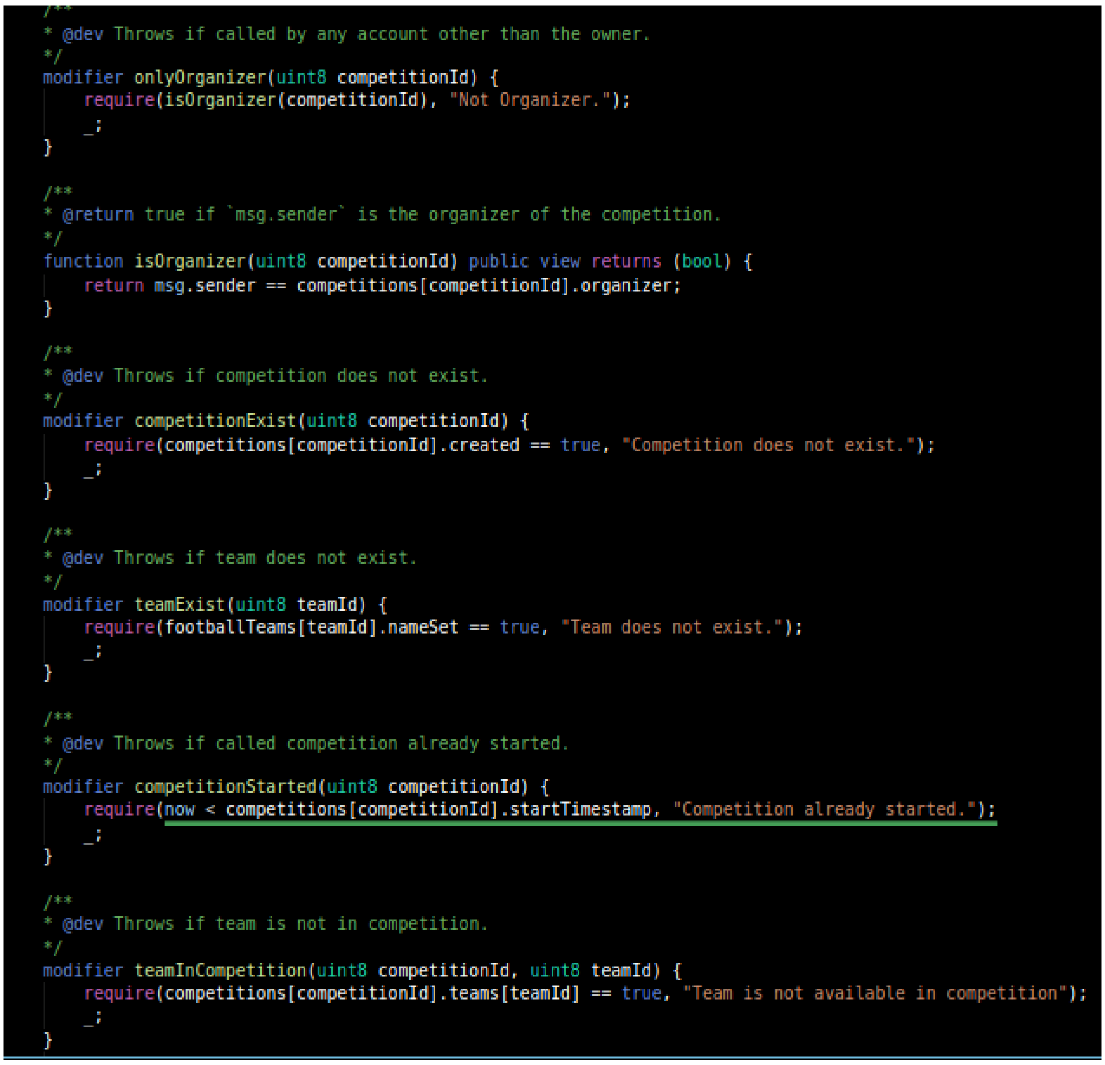
Figure 2.11: Three different structs in the 'FootballCompetition.sol' contract.

Then variables for the contract were defined and again from the comments one can know what the use for these variables is. In the constructor the count for the competition and teams is set to 0, as we want them to be 0 once the contract is deployed. Both the variables and constructor can be shown in Fig. 2.12. Some modifiers which will later be used were also define and these can be shown in Fig. 2.13 (the comments explain the use for each modifier).



```
/* holds count of the total competitions */
uint8 private competitionCount;

/* the maximum competitions which can be added */
uint8 constant maxCompetitions = 255;

/* the maximum football teams which can be added */
uint8 constant maxFootballTeams = 255;

/* holds count of the total teams */
uint8 private teamCount;

/* holds football teams which can be found in any competition */
/* for the purposes of the problem only 255 teams can be added to this contract */
mapping(uint8 => Team) private footballTeams;

/* holds a mapping of available competitions */
/* for the purposes of the problem only 255 competitions can be added to this contract */
mapping(uint8 => Competition) private competitions;

/* this constructor is executed at initialization and sets the owner of the contract */
constructor() public {
    competitionCount = 0; // set number of competitions to 0
    teamCount = 0;        // set number of teams to 0
}
```

Figure 2.12: Variables and constructor for 'FootballCompetition.sol' contract.



```
/**
* @dev Throws if called by any account other than the owner.
*/
modifier onlyOrganizer(uint8 competitionId) {
    require(isOrganizer(competitionId), "Not Organizer.");
    _;
}

/**
* @return true if `msg.sender` is the organizer of the competition.
*/
function isOrganizer(uint8 competitionId) public view returns (bool) {
    return msg.sender == competitions[competitionId].organizer;
}

/**
* @dev Throws if competition does not exist.
*/
modifier competitionExist(uint8 competitionId) {
    require(competitions[competitionId].created == true, "Competition does not exist.");
    _;
}

/**
* @dev Throws if team does not exist.
*/
modifier teamExist(uint8 teamId) {
    require(footballTeams[teamId].nameSet == true, "Team does not exist.");
    _;
}

/**
* @dev Throws if called competition already started.
*/
modifier competitionStarted(uint8 competitionId) {
    require(now < competitions[competitionId].startTimestamp, "Competition already started.");
    _;
}

/**
* @dev Throws if team is not in competition.
*/
modifier teamInCompetition(uint8 competitionId, uint8 teamId) {
    require(competitions[competitionId].teams[teamId] == true, "Team is not available in competition");
    _;
}
```

Figure 2.13: Modifiers for 'FootballCompetition.sol' contract.

The first two functions written in the contract as shown in Fig. 2.14, just outputs the number of teams and competitions available in the contract. This can later be used to query the total number of teams available to select from, when organizing a new competition, while the other gives us the number of competitions available in the contract. This is a common design pattern used in smart contracts, since when you want to show the available teams or competition from a client, first you get the count and then call the get team or get competition to view the details for every team or competition available (ids are set incrementally). This is done because using unbound arrays in the EVM can have a high gas cost.

The other function shown in Fig. 2.14 ('addTeam') lets the owner (the one who deployed the contract) to add a new team. A name for the team must be specified and this was written in this way, because we assume that the contract can have different teams to select from to when creating a new competition. Another consideration which was taken, is that a team name cannot be edited for transparency/trust reasons (we don't want the owner of the contract to change the name of the team with a specific id as this can cause issues when the winner or a bet is set in previous or active competitions). Once a team is added it is saved in the mapping 'footballTeams' and the variable/bool 'nameSet' is set to true. This is a common design pattern in solidity, whenever you need to check if a key exists in the mapping ('teamExists' modifier).

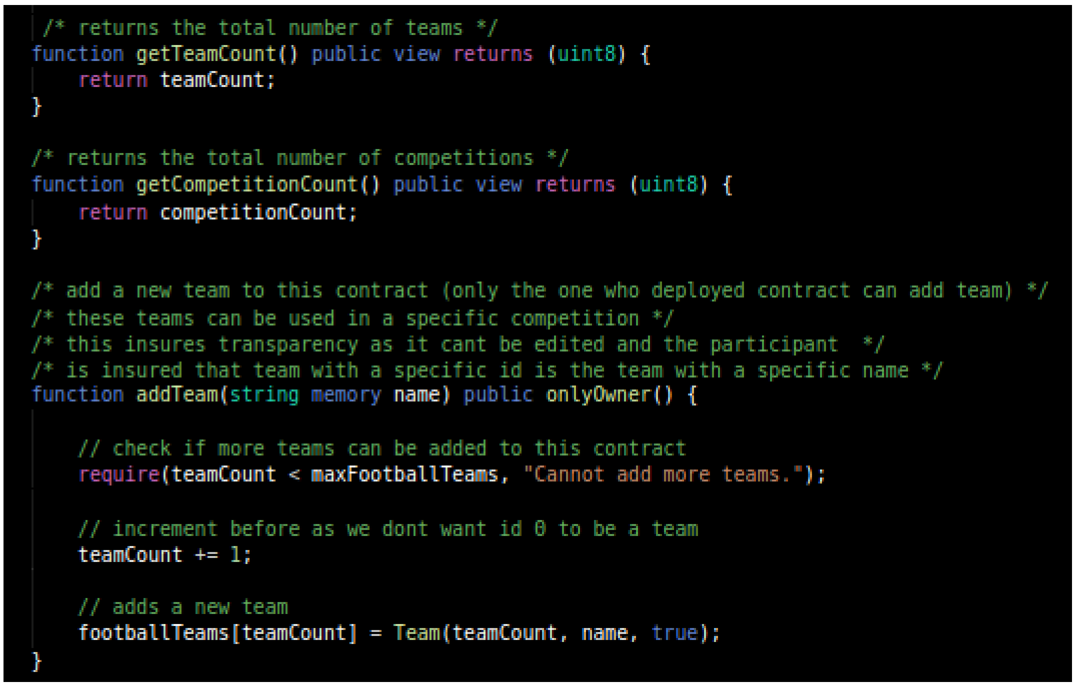
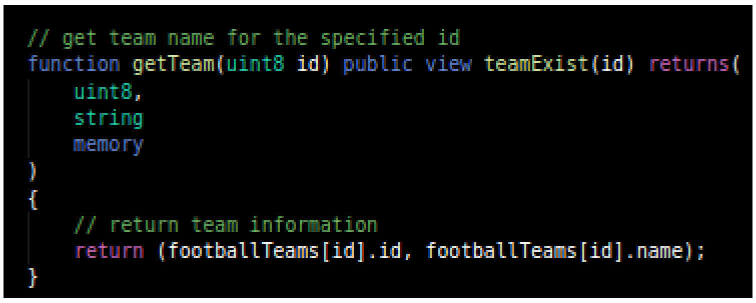

```
/* returns the total number of teams */
function getTeamCount() public view returns (uint8) {
    return teamCount;
}

/* returns the total number of competitions */
function getCompetitionCount() public view returns (uint8) {
    return competitionCount;
}

/* add a new team to this contract (only the one who deployed contract can add team) */
/* these teams can be used in a specific competition */
/* this insures transparency as it cant be edited and the participant  */
/* is insured that team with a specific id is the team with a specific name */
function addTeam(string memory name) public onlyOwner() {

    // check if more teams can be added to this contract
    require(teamCount < maxFootballTeams, "Cannot add more teams.");

    // increment before as we dont want id 0 to be a team
    teamCount += 1;

    // adds a new team
    footballTeams[teamCount] = Team(teamCount, name, true);
}
```

Figure 2.14: First three functions in 'FootballCompetition.sol' contract.

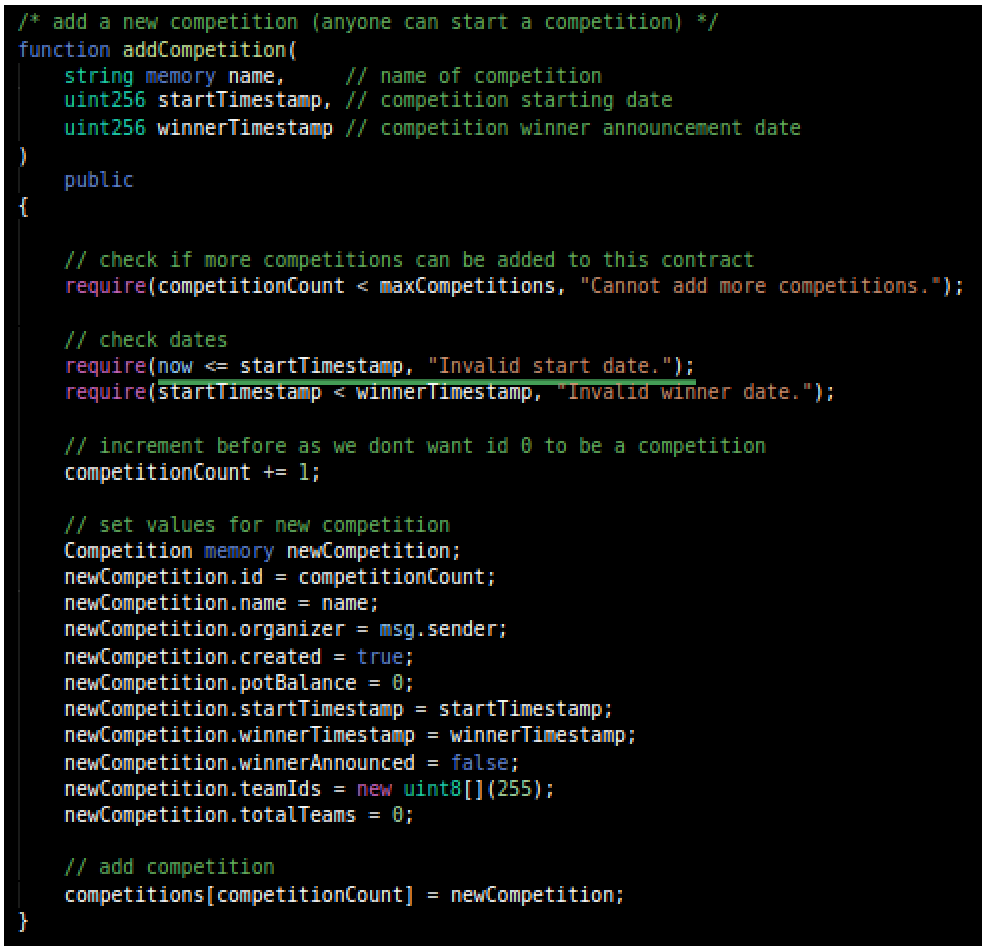
The next function is the 'getTeam' as shown in Fig. 2.15, and this returns details about a team with the specified id. Such function can be called in conjunction with the get count for the teams from a client to get all the available teams, which can be selected when creating a competition (as described due to unbounded arrays).



```
// get team name for the specified id
function getTeam(uint8 id) public view teamExist(id) returns(
    uint8,
    string
    memory
)
{
    // return team information
    return (footballTeams[id].id, footballTeams[id].name);
}
```

Figure 2.15: Get team details function in 'FootballCompetition.sol' contract.

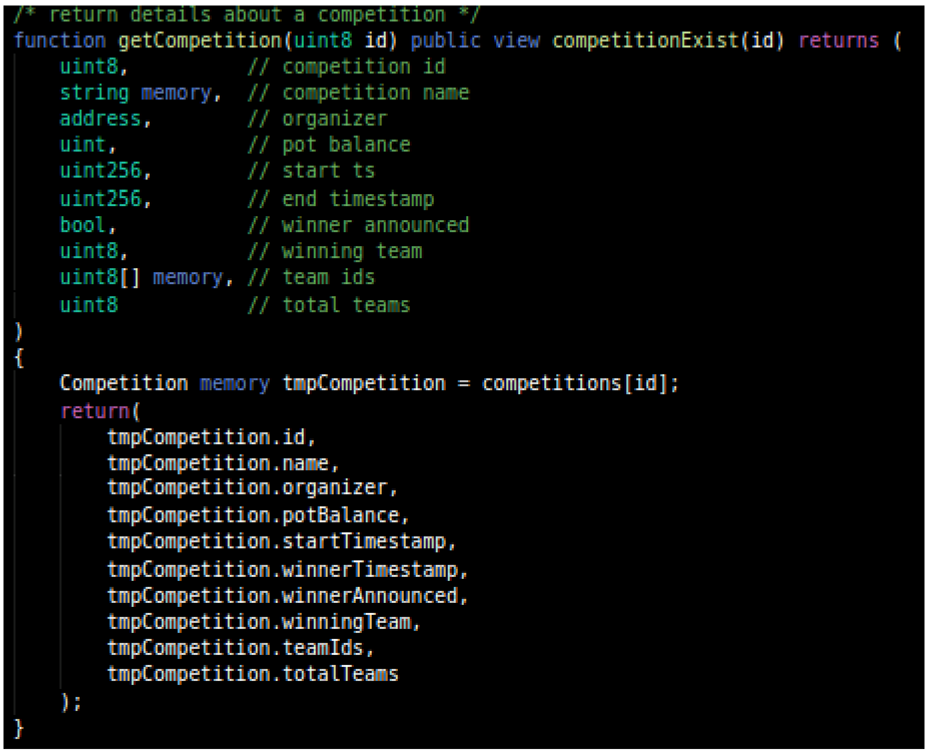
One of the most important function in the contract is the 'addCompetition' function as shown in Fig. 2.16 and this lets anyone to create a new competition which other addresses can join and bet in. The dates when the competition starts, and end must be specified together with the name of the competition. The address who created the competition is also noted and once this function is called the newly created competition is saved in the mapping. The created variable/bool is used in similar ways as the Team struct, which is to check if a competition exists ('competitionExists' modifier).



```
/* add a new competition (anyone can start a competition) */
function addCompetition(
    string memory name,     // name of competition
    uint256 startTimestamp, // competition starting date
    uint256 winnerTimestamp // competition winner announcement date
)
    public
{

    // check if more competitions can be added to this contract
    require(competitionCount < maxCompetitions, "Cannot add more competitions.");

    // check dates
    require(now <= startTimestamp, "Invalid start date.");
    require(startTimestamp < winnerTimestamp, "Invalid winner date.");

    // increment before as we dont want id 0 to be a competition
    competitionCount += 1;

    // set values for new competition
    Competition memory newCompetition;
    newCompetition.id = competitionCount;
    newCompetition.name = name;
    newCompetition.organizer = msg.sender;
    newCompetition.created = true;
    newCompetition.potBalance = 0;
    newCompetition.startTimestamp = startTimestamp;
    newCompetition.winnerTimestamp = winnerTimestamp;
    newCompetition.winnerAnnounced = false;
    newCompetition.teamIds = new uint8[](255);
    newCompetition.totalTeams = 0;

    // add competition
    competitions[competitionCount] = newCompetition;
}
```

Figure 2.16: Add competition function in 'FootballCompetition.sol' contract.

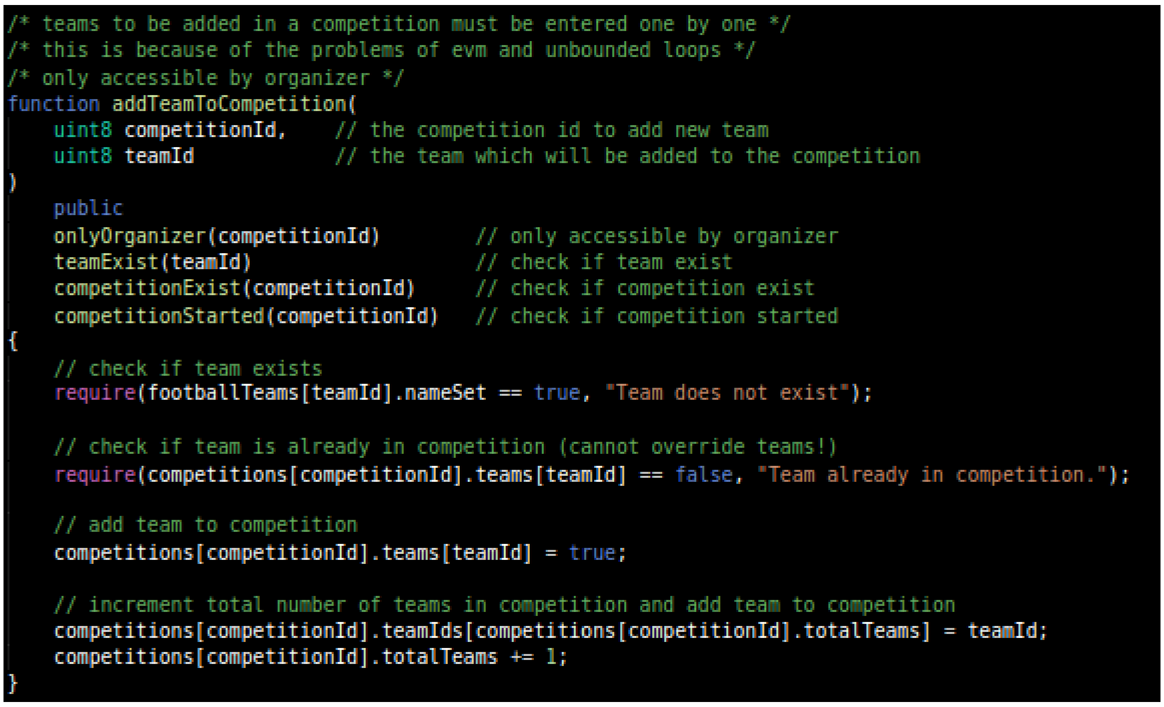
The 'getCompetition' function works similarly as the described function 'getTeam'. This can be used to check the details for available competitions stored on the chain and this is shown in Fig. 2.17. Such details include the available teams an address can bet on for this particular competition.



```
/* return details about a competition */
function getCompetition(uint8 id) public view competitionExist(id) returns (
    uint8,          // competition id
    string memory,  // competition name
    address,        // organizer
    uint,           // pot balance
    uint256,        // start ts
    uint256,        // end timestamp
    bool,           // winner announced
    uint8,          // winning team
    uint8[] memory, // team ids
    uint8           // total teams
)
{
    Competition memory tmpCompetition = competitions[id];
    return(
        tmpCompetition.id,
        tmpCompetition.name,
        tmpCompetition.organizer,
        tmpCompetition.potBalance,
        tmpCompetition.startTimestamp,
        tmpCompetition.winnerTimestamp,
        tmpCompetition.winnerAnnounced,
        tmpCompetition.winningTeam,
        tmpCompetition.teamIds,
        tmpCompetition.totalTeams
    );
}
```

Figure 2.17: Get competition details function in 'FootballCompetition.sol' contract.

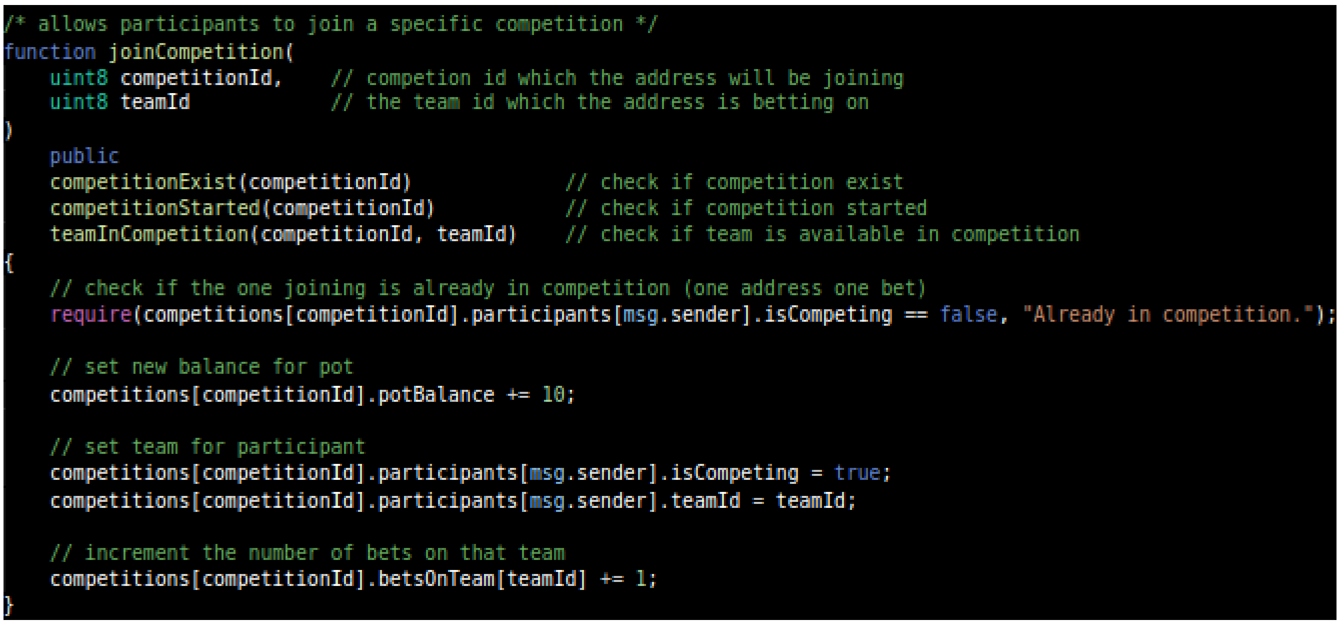
Another important function which can only be accessed by the one who created the competition ('onlyOrganizer' modifier) is the 'addTeamToCompetition' as shown in Fig. 2.18. This lets the organizer to add teams which will be competing in the created competition and this team must be one of the available teams which are saved on the chain ('teamExists' modifier). Note that an organizer cannot add new teams to the competition if the event already started and another restriction added, is that the organizer cannot add already existing teams in the competition.



```
/* teams to be added in a competition must be entered one by one */
/* this is because of the problems of evm and unbounded loops */
/* only accessible by organizer */
function addTeamToCompetition(
    uint8 competitionId,    // the competition id to add new team
    uint8 teamId            // the team which will be added to the competition
)
    public
    onlyOrganizer(competitionId)        // only accessible by organizer
    teamExist(teamId)                   // check if team exist
    competitionExist(competitionId)     // check if competition exist
    competitionStarted(competitionId)   // check if competition started
{
    // check if team exists
    require(footballTeams[teamId].nameSet == true, "Team does not exist");

    // check if team is already in competition (cannot override teams!)
    require(competitions[competitionId].teams[teamId] == false, "Team already in competition.");

    // add team to competition
    competitions[competitionId].teams[teamId] = true;

    // increment total number of teams in competition and add team to competition
    competitions[competitionId].teamIds[competitions[competitionId].totalTeams] = teamId;
    competitions[competitionId].totalTeams += 1;
}
```

Figure 2.18: Add team to competition function in 'FootballCompetition.sol' contract.

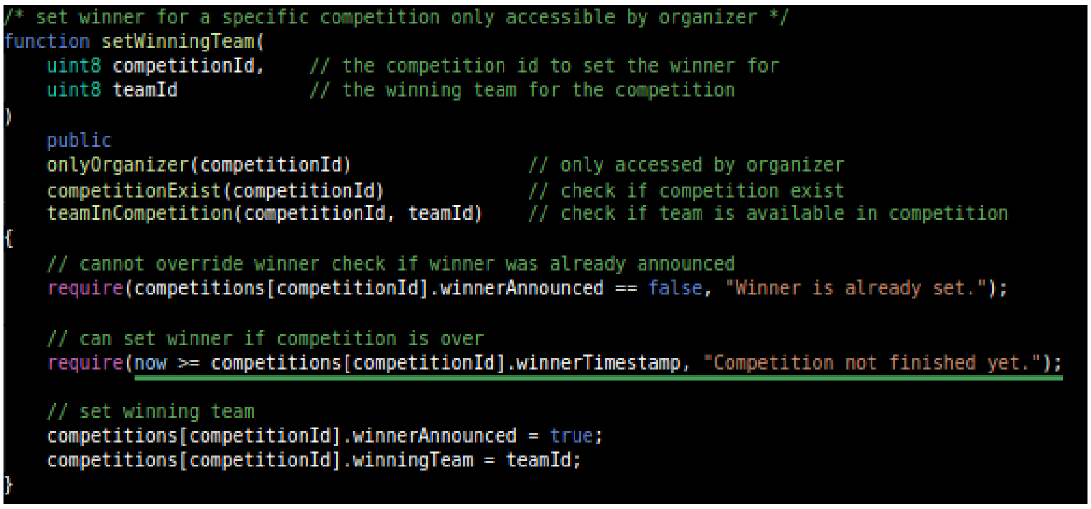
To join an existing competition the function 'joinCompetition' was created as shown in Fig. 2.19. This allows an address to put a bet from the teams in the competition and this restricts the address to one bet only. Also no one can join the competition if the event already started ('competitionStarted' modifier). It is also important to note that the count for each bet done on a specific team is noted in the variable 'betsOnTeam' inside the Competition struct. This was done to easily and efficiently split the winnings in a function which will be discussed soon.



```
/* allows participants to join a specific competition */
function joinCompetition(
    uint8 competitionId,    // competion id which the address will be joining
    uint8 teamId            // the team id which the address is betting on
)
    public
    competitionExist(competitionId)             // check if competition exist
    competitionStarted(competitionId)           // check if competition started
    teamInCompetition(competitionId, teamId)    // check if team is available in competition
{
    // check if the one joining is already in competition (one address one bet)
    require(competitions[competitionId].participants[msg.sender].isCompeting == false, "Already in competition.");

    // set new balance for pot
    competitions[competitionId].potBalance += 10;

    // set team for participant
    competitions[competitionId].participants[msg.sender].isCompeting = true;
    competitions[competitionId].participants[msg.sender].teamId = teamId;

    // increment the number of bets on that team
    competitions[competitionId].betsOnTeam[teamId] += 1;
}
```

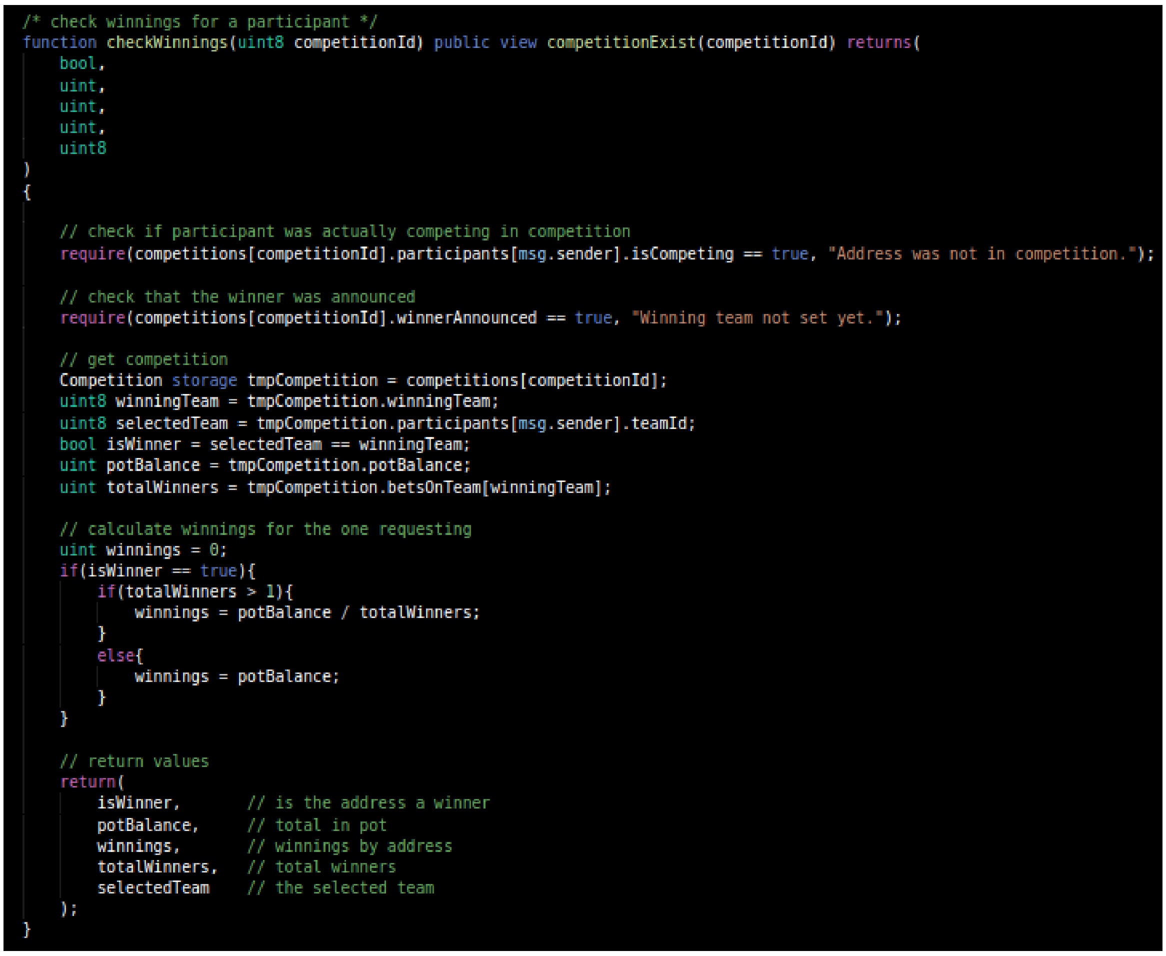
Figure 2.19: Join competition function in 'FootballCompetition.sol' contract.

An organizer then can set the winner for the competition by calling the 'setWinningTeam' function as shown in Fig. 2.20. The winner can only be set if the end date has passed as shown in the second require in the function. The last function which was written lets any address which was competing to check if they won anything. This could be called once the winning team is set. Such function is shown in Fig. 2.21.



```
/* set winner for a specific competition only accessible by organizer */
function setWinningTeam(
    uint8 competitionId,    // the competition id to set the winner for
    uint8 teamId            // the winning team for the competition
)
    public
    onlyOrganizer(competitionId)                // only accessed by organizer
    competitionExist(competitionId)             // check if competition exist
    teamInCompetition(competitionId, teamId)    // check if team is available in competition
{
    // cannot override winner check if winner was already announced
    require(competitions[competitionId].winnerAnnounced == false, "Winner is already set.");

    // can set winner if competition is over
    require(now >= competitions[competitionId].winnerTimestamp, "Competition not finished yet.");

    // set winning team
    competitions[competitionId].winnerAnnounced = true;
    competitions[competitionId].winningTeam = teamId;
}
```

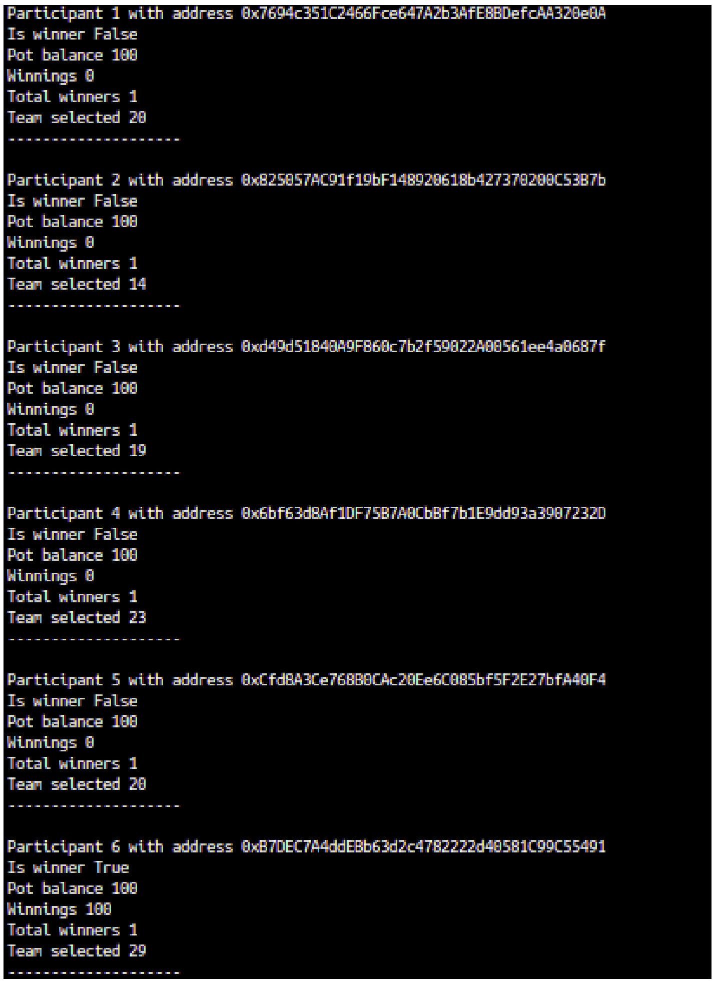
Figure 2.20: Set winner for competition function in 'FootballCompetition.sol' contract.



```
/* check winnings for a participant */
function checkWinnings(uint8 competitionId) public view competitionExist(competitionId) returns(
    bool,
    uint,
    uint,
    uint,
    uint8
)
{

    // check if participant was actually competing in competition
    require(competitions[competitionId].participants[msg.sender].isCompeting == true, "Address was not in competition.");

    // check that the winner was announced
    require(competitions[competitionId].winnerAnnounced == true, "Winning team not set yet.");

    // get competition
    Competition storage tmpCompetition = competitions[competitionId];
    uint8 winningTeam = tmpCompetition.winningTeam;
    uint8 selectedTeam = tmpCompetition.participants[msg.sender].teamId;
    bool isWinner = selectedTeam == winningTeam;
    uint potBalance = tmpCompetition.potBalance;
    uint totalWinners = tmpCompetition.betsOnTeam[winningTeam];

    // calculate winnings for the one requesting
    uint winnings = 0;
    if(isWinner == true){
        if(totalWinners > 1){
            winnings = potBalance / totalWinners;
        }
        else{
            winnings = potBalance;
        }
    }

    // return values
    return(
        isWinner,       // is the address a winner
        potBalance,     // total in pot
        winnings,       // winnings by address
        totalWinners,   // total winners
        selectedTeam    // the selected team
    );
}
```

Figure 2.21: Check winnings for competition function in 'FootballCompetition.sol' contract.

Once the contract was written it was compiled and deployed from python. Some tests were done to make sure everything works well. One thing to keep note of, is the time on the chain is set to UTC so the timestamps send to create a competition must be converted to UTC for this contract to work properly. The test was conducted using the current timestamp and add 5 seconds to the start timestamp and 10 seconds to the end timestamp. Then a delay is set in python, and this was done to make sure that the timestamp checks are working correctly. Finally, we would like to add that this contract can be used for the use case specified in this question. Fig. 2.22 shows the final output for the test, which is a call to the function which checks the winnings for a participant (first 6 participants are shown using a loop).



```
Participant 1 with address 0x7694c351C2466Fce647A2b3AfE8BDefcAA320e0A
Is winner False
Pot balance 100
Winnings 0
Total winners 1
Team selected 20
--------------------

Participant 2 with address 0x825057AC91f19bF148920618b427370200C53B7b
Is winner False
Pot balance 100
Winnings 0
Total winners 1
Team selected 14
--------------------

Participant 3 with address 0xd49d51840A9F860c7b2f59022A00561ee4a0687f
Is winner False
Pot balance 100
Winnings 0
Total winners 1
Team selected 19
--------------------

Participant 4 with address 0x6bf63d8Af1DF75B7A0CbBf7b1E9dd93a3907232D
Is winner False
Pot balance 100
Winnings 0
Total winners 1
Team selected 23
--------------------

Participant 5 with address 0xCfd8A3Ce768B0CAc20Ee6C085bf5F2E27bfA40F4
Is winner False
Pot balance 100
Winnings 0
Total winners 1
Team selected 20
--------------------

Participant 6 with address 0xB7DEC7A4ddEBb63d2c4782222d40581C99C55491
Is winner True
Pot balance 100
Winnings 100
Total winners 1
Team selected 29
--------------------
```

Figure 2.22: Final output of winnings when testing the contract.

2.5 Question 5

You are working in an organisation providing Machine Learning consultancy. As an organisation you have access to internal private data, however you wish to start selling Machine Learning models, as a service, trained on, but without sharing, this private data. With the availability of blockchain you decide to make this business model work via smart contracts. As a proof of concept you are asked to:

2.5.1 Q5 (i)

Utilize the standard Iris data set (assuming it's internal/private data) and create a simple logistic regression model to create a classification model (utilize only 80% of the data for training).

2.5.2 Q5 (ii)

Define and deploy a smart contract. As the owner of the contract, you are allowed to upload (or update) the trained model parameters from the off-chain ML training software on the smart contract.

2.5.3 Q5 (iii)

Simulate a typical organisation client, assuming that the client has the remaining 20% unseen data (hence assuming that this 20% of the data is public). The client has his own off-chain client program that is allowed to request the trained model parameters from the smart contract. For each request

- (a) The client receives the logistic regression trained parameters.
- (b) The client is automatically charged 0.01 ether from his account and transferred to the organisation account.

2.5.4 Q5 (iv)

Once the client receives the trained model parameters, the client applies the logistic regression parameters received from the smart contract to his local untrained model and performs out-of-sample testing on the remaining 20% of the data set.

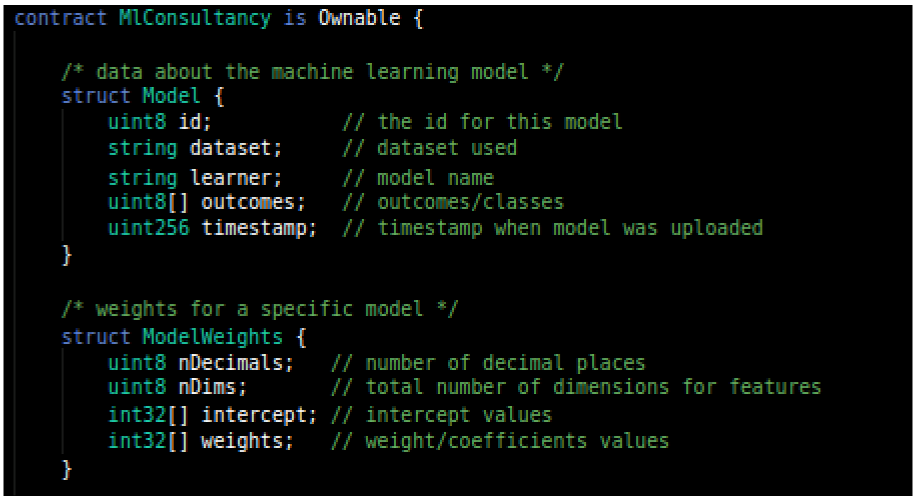
2.5.5 Present (a)

A description of the approach and technologies used.

Code for this task is in *'src/solidity/ml_consultancy_truffle'*. Inside the location provided there is a folder called *'contracts'* were the smart contract called 'MlConsultancy.sol' is used to manage this use case. Also, there is a folder called *'python'* which has two scripts called 'ml_consultancy.py' (holds classes for Client and Operator) and another script called 'prototype.py' which is used to showcase the prototype.

The following approach was taken to create a prototype for this task. First off, a new contract was created called 'MlConsultancy.sol' and this contract inherits from 'Ownable.sol' provided by OpenZepplin. The 'Ownable.sol' gives us basic functionality to check that the owner of the contract can access a specific function (with the use of a modifier in the contract) and other functionalities such as transferring ownership. Such contract was slightly modified, so the owner address is set to be payable since we need the payable keyword to send funds to the owner as described in the question.

After creating the contract two structs were written for specific purposes as shown in Fig. 2.23. The first struct 'Model' holds details about the model. This contract was written with the assumption that the operator/organization can upload multiple models which clients can pay for (makes the proof of concept more similar to a real-world application). From the comments it is easy to understand what the variables inside the struct are used for. The other struct 'ModelWeights' is used to hold information about the weights for a specific available model. Comments in the code explains what the variables inside the struct are used for. It is important to note that the variable called 'nDecimals' is used to keep track of where to put the decimal point, since the weights and intercept are saved as integer values on the network (EVM does not support floating point numbers). This variable will be utilised by the client to convert the integer values to floating points.



```
contract MlConsultancy is Ownable {

    /* data about the machine learning model */
    struct Model {
        uint8 id;           // the id for this model
        string dataset;     // dataset used
        string learner;     // model name
        uint8[] outcomes;   // outcomes/classes
        uint256 timestamp;  // timestamp when model was uploaded
    }

    /* weights for a specific model */
    struct ModelWeights {
        uint8 nDecimals;   // number of decimal places
        uint8 nDims;       // total number of dimensions for features
        int32[] intercept; // intercept values
        int32[] weights;   // weight/coefficients values
    }
```

Figure 2.23: Two structs used in the 'MlConsultancy.sol' contract.

After creating the structs described the variables and constructor shown in Fig. 2.24 were written to the contract. The 'SERVICE_COST' is the cost (0.01 ether) for which the client needs to pay to access the weights as specified in the question. For the purposes of this proof of concept the number of different models which can be uploaded to the contract/chain is capped to 255, which is why the constant ‘maxModels’ was specified. The ‘modelWeights’ variable holds a reference to the ‘ModelWeight’ struct given the model id (the same id in the ‘Model’ struct) is used as a key. ‘clientAccess’ mapping is a used to hold a reference for which access and to what model a specific address/client has. Finally, the ‘modelCount’ variable is set, which holds a reference to how many models are available in the contract. When the contract is deploying and the constructor is called we want that the ‘modelCount’ is set to 0.

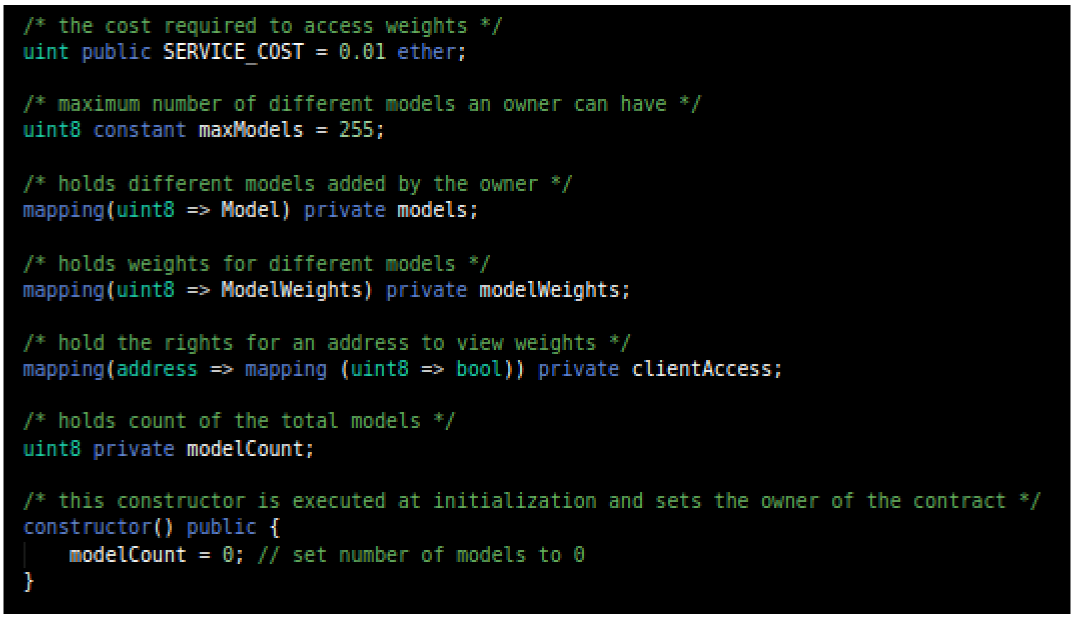

```
/* the cost required to access weights */
uint public SERVICE_COST = 0.01 ether;

/* maximum number of different models an owner can have */
uint8 constant maxModels = 255;

/* holds different models added by the owner */
mapping(uint8 => Model) private models;

/* holds weights for different models */
mapping(uint8 => ModelWeights) private modelWeights;

/* hold the rights for an address to view weights */
mapping(address => mapping (uint8 => bool)) private clientAccess;

/* holds count of the total models */
uint8 private modelCount;

/* this constructor is executed at initialization and sets the owner of the contract */
constructor() public {
    modelCount = 0; // set number of models to 0
}
```

Figure 2.24: Variables and constructor in the ’MlConsultancy.sol’ contract.

Two simple functions then were written, which basically return the service cost and the number of models available as shown in Fig. 2.25.

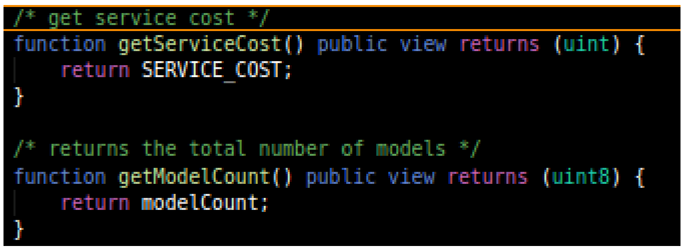
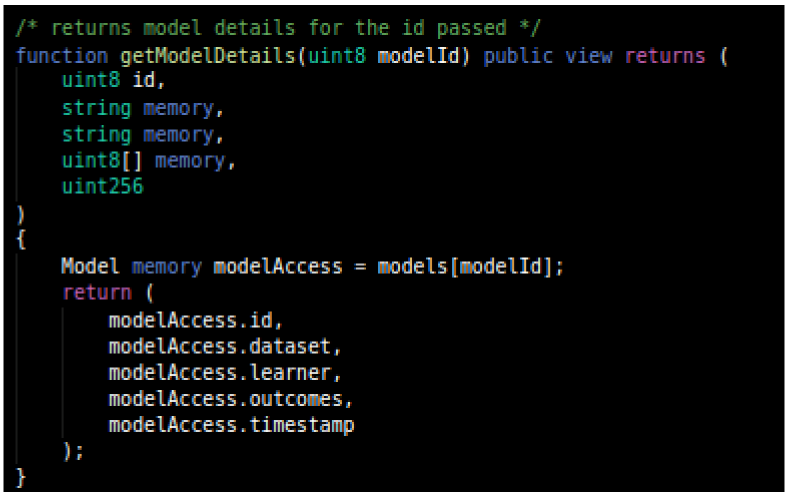

```
/* get service cost */
function getServiceCost() public view returns (uint) {
    return SERVICE_COST;
}

/* returns the total number of models */
function getModelCount() public view returns (uint8) {
    return modelCount;
}
```

Figure 2.25: Get cost and get model count functions in the ’MlConsultancy.sol’ contract.

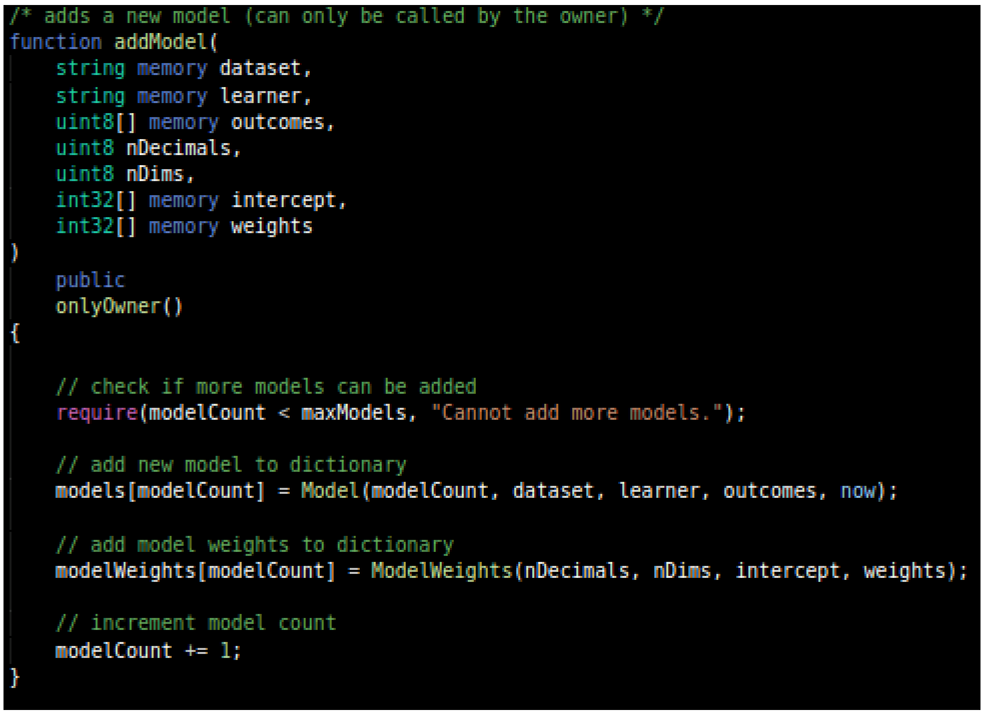
Another function was created called ‘getModelDetails’ and as the name implies it gets the details for a specific model. The model id must be passed as a parameter and this function like the other two functions above can be accessed by any address. This will return the model id, the name of the dataset used to train this model, the name of the machine learning model used, the different labels which can be outputted for the problem it was trained for and the timestamp when the model details were uploaded. Such function gives potential client/customers or even existing customers to view available models which can be sold by the organization. This function is shown in Fig. 2.26.



```
/* returns model details for the id passed */
function getModelDetails(uint8 modelId) public view returns (
    uint8 id,
    string memory,
    string memory,
    uint8[] memory,
    uint256
)
{
    Model memory modelAccess = models[modelId];
    return (
        modelAccess.id,
        modelAccess.dataset,
        modelAccess.learner,
        modelAccess.outcomes,
        modelAccess.timestamp
    );
}
```

Figure 2.26: Get model details function in the 'MlConsultancy.sol' contract.

The next function which was written is the 'addModel' function which allows the owner (the one who deployed the contract) to add new models along with the weights to the contract. In this case the owner of the contract (organization providing the service) can access this function and this is why there is a 'onlyOwner()' modifier to restrict others from adding models to this contract. This function is shown in Fig. 2.27.

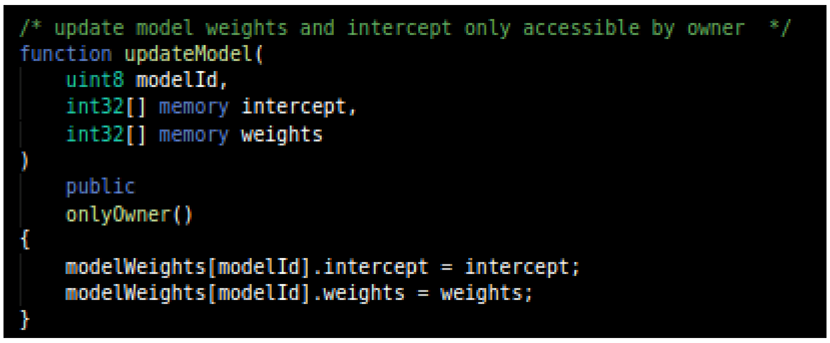


```
/* adds a new model (can only be called by the owner) */
function addModel(
    string memory dataset,
    string memory learner,
    uint8[] memory outcomes,
    uint8 nDecimals,
    uint8 nDims,
    int32[] memory intercept,
    int32[] memory weights
)
    public
    onlyOwner()
{

    // check if more models can be added
    require(modelCount < maxModels, "Cannot add more models.");

    // add new model to dictionary
    models[modelCount] = Model(modelCount, dataset, learner, outcomes, now);

    // add model weights to dictionary
    modelWeights[modelCount] = ModelWeights(nDecimals, nDims, intercept, weights);

    // increment model count
    modelCount += 1;
}
```

Figure 2.27: Add model function in the 'MlConsultancy.sol' contract.

The following parameters must be specified to add the model to the contract; the name of the dataset used to train the model, the name of the machine learning model, the outcomes which can be outputted by the model, the place where to put the decimal point to (when converting back integers to floats), the number of dimensions/features in the dataset, the intercept learned by the model and finally the weights learned by the model. Once these parameters are passed a struct ‘Model’ is created and added to the ‘models’ mapping and a ‘ModelWeights’ struct is created and added to the ‘modelWeights’ mapping. The id for both mapping is the ‘modelCount’ and at the end of this function this value is incremented by 1.

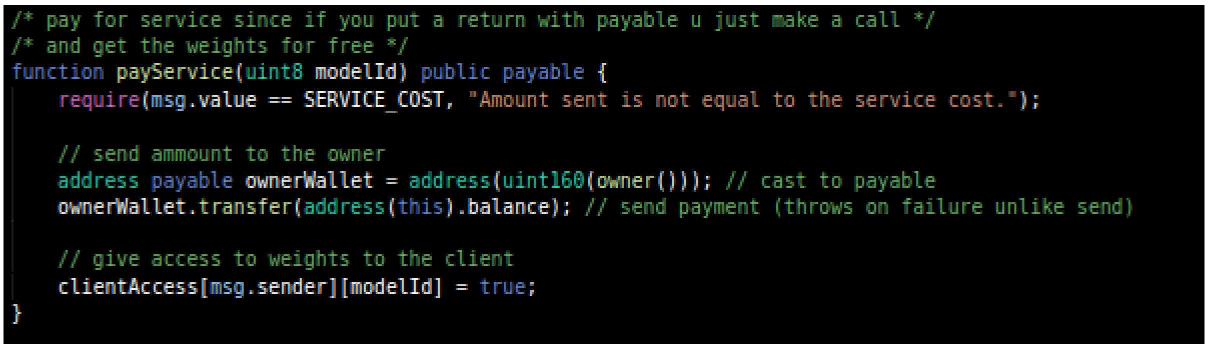
Following this function, the ‘updateModel’ was created which enables the organizer to update the weights and intercept of an existing model as described in the question. This function is shown in Fig. 2.28.



```
/* update model weights and intercept only accessible by owner  */
function updateModel(
    uint8 modelId,
    int32[] memory intercept,
    int32[] memory weights
)
    public
    onlyOwner()
{
    modelWeights[modelId].intercept = intercept;
    modelWeights[modelId].weights = weights;
}
```

Figure 2.28: Update model function in the ’MlConsultancy.sol’ contract.

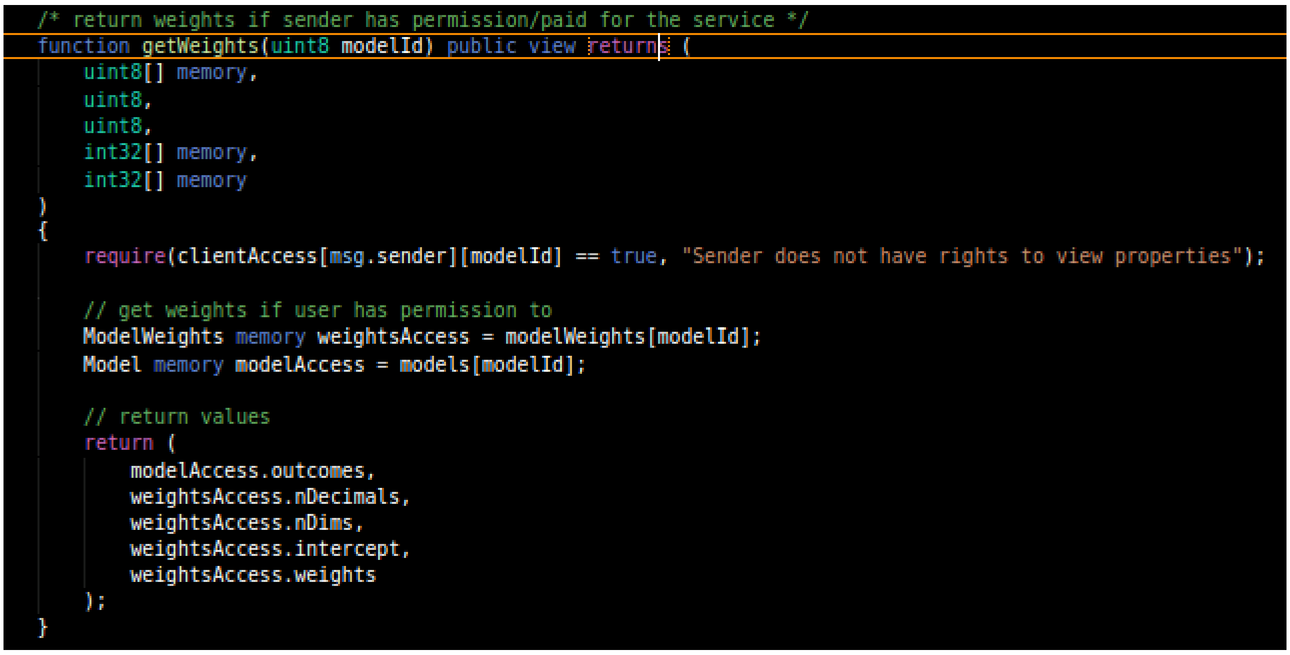
The next function which was written is quite important for this task, the ‘payService’ allows clients to pay for the service to be able to access the model weights for the model which they paid for. This function is shown in Fig. 2.29. The client who transacts this function must send the specified value of 0.01 ether and the model id must be specified. In this function the funds are automatically transferred to the organizer as specified in the question. A reference in the ‘clientAccess’ mapping is taken where it is noted (setting bool to true), that an address has access to the weights for the model id which was specified (passed as a parameter).



```
/* pay for service since if you put a return with payable u just make a call */
/* and get the weights for free */
function payService(uint8 modelId) public payable {
    require(msg.value == SERVICE_COST, "Amount sent is not equal to the service cost.");

    // send ammount to the owner
    address payable ownerWallet = address(uint160(owner())); // cast to payable
    ownerWallet.transfer(address(this).balance); // send payment (throws on failure unlike send)

    // give access to weights to the client
    clientAccess[msg.sender][modelId] = true;
}
```

Figure 2.29: Pay for service function in the ’MlConsultancy.sol’ contract.

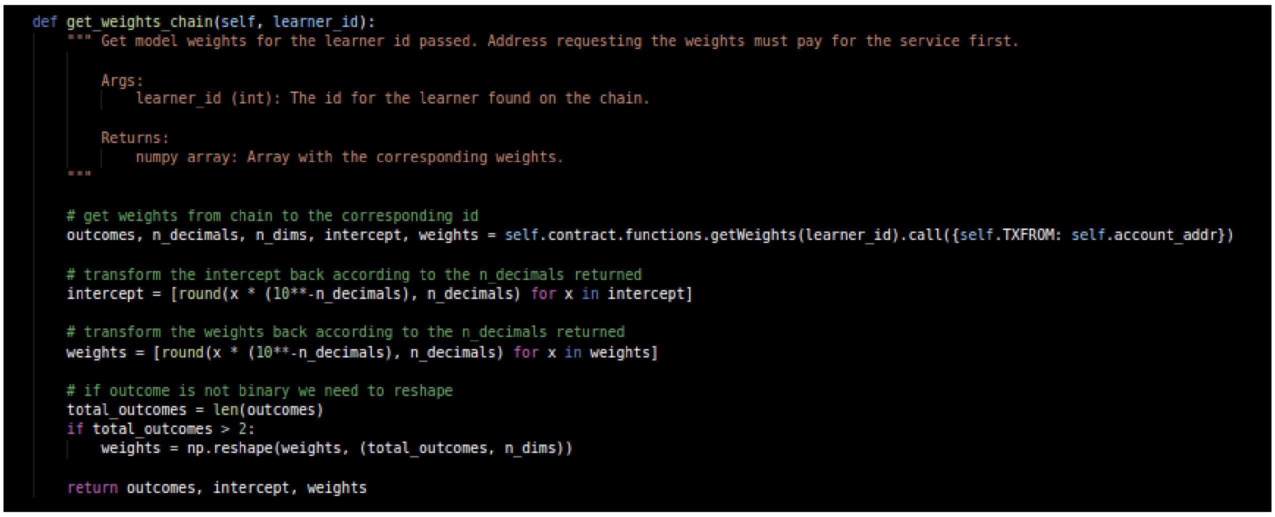
The final function is the ‘getWeights’ which allows clients who paid for a specific model to access its weights. This function is shown in Fig. 2.30. The client requesting the weights must specify the model id and the weights are returned only if the client previously paid by calling the ’payService’ function (the require part does this validation by checking that the value in the mapping 'clientAccess' is set to true). The following information is then sent to the client if everything is valid; the outcomes for the model, the point where to put the decimal point, the number of features, the intercept which was learnt and the weights which was learnt. That concludes the approach taken from the contract side.



```
/* return weights if sender has permission/paid for the service */
function getWeights(uint8 modelId) public view returns (
    uint8[] memory,
    uint8,
    uint8,
    int32[] memory,
    int32[] memory
)
{
    require(clientAccess[msg.sender][modelId] == true, "Sender does not have rights to view properties");

    // get weights if user has permission to
    ModelWeights memory weightsAccess = modelWeights[modelId];
    Model memory modelAccess = models[modelId];

    // return values
    return (
        modelAccess.outcomes,
        weightsAccess.nDecimals,
        weightsAccess.nDims,
        weightsAccess.intercept,
        weightsAccess.weights
    );
}
```

Figure 2.30: Get weights function in the 'MlConsultancy.sol' contract.

Two new python classes were created in the 'ml_consutlancy.py', one for the client which is called 'Client' and the other is called 'Operator' which is used by the organization. Both classes inherit from another created class called 'Service' which basically provides common functions between the two to interact with the contract. In this section the most important functions in the classes will be described, as for the other functions everything is well documented and can be viewed from the script.

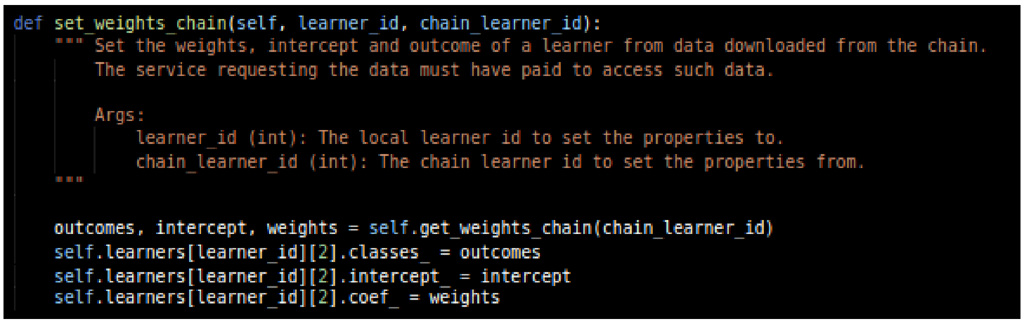


```
def get_weights_chain(self, learner_id):
    """ Get model weights for the learner id passed. Address requesting the weights must pay for the service first.

        Args:
            learner_id (int): The id for the learner found on the chain.

        Returns:
            numpy array: Array with the corresponding weights.
    """

    # get weights from chain to the corresponding id
    outcomes, n_decimals, n_dims, intercept, weights = self.contract.functions.getWeights(learner_id).call({self.TXFROM: self.account_addr})

    # transform the intercept back according to the n_decimals returned
    intercept = [round(x * (10**-n_decimals), n_decimals) for x in intercept]

    # transform the weights back according to the n_decimals returned
    weights = [round(x * (10**-n_decimals), n_decimals) for x in weights]

    # if outcome is not binary we need to reshape
    total_outcomes = len(outcomes)
    if total_outcomes > 2:
        weights = np.reshape(weights, (total_outcomes, n_dims))

    return outcomes, intercept, weights
```

Figure 2.31: Get weights from the chain in the 'Service' class in the 'mlconsultancy.py'.

The function ‘get_weights_chain’ as shown in Fig. 2.31 is found the ‘Service’ class which is a base class for the other two classes mentioned above. This allows for the instance to fetch the weight for a specific model with a specific id if this instance have paid for the service. In our use case this will be utilised by the ‘Client’ class, but it was decided to put this function in the base class. As shown in the function both the ‘weights’ and ‘intercept’ are transformed back to floating point numbers using the ‘n_decimals’ sent back from the chain. Also, if the outcome for the specified model is not binary the ‘weights’ array is transformed accordingly.

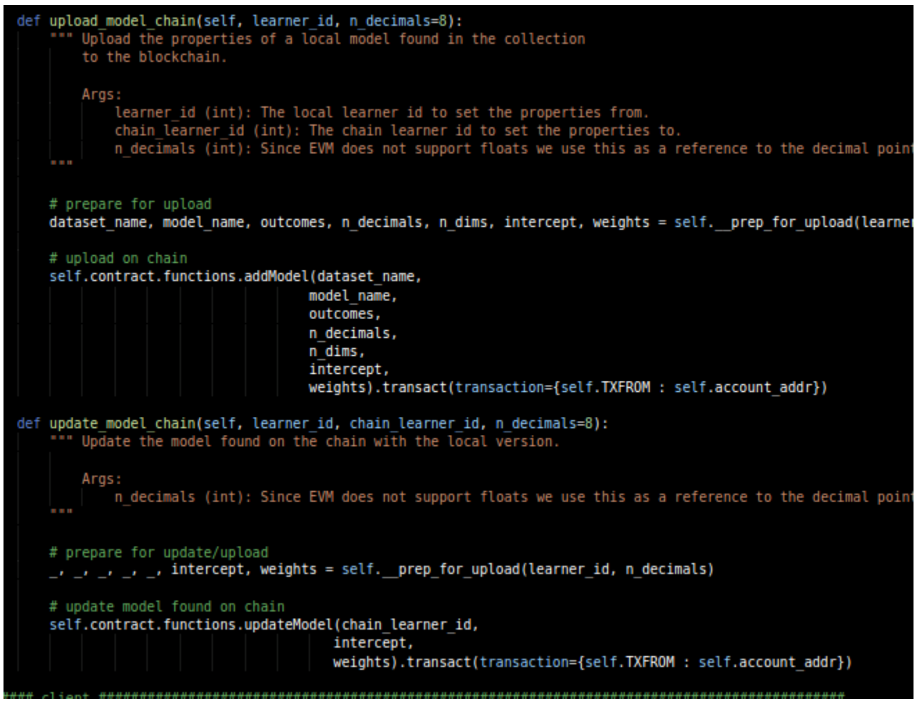
The second function in the python script which will be described is the ‘set_weights_chain’ as shown in Fig. 2.32 and this too reside in the ‘Service’ base class. This will be utilised by the client to set the weights for a local model to the weights received from the chain. The main reasons why these two functions were written in the base class although these are mostly utilised by the ‘Client’, is that it may be the case that the service might want to reload a model with the weights found on the chain. This function takes two parameters, the first parameter ‘learner_id’ is used to retrieve the local model from the collection of local models and the second parameter ‘chain_learner_id’ is used to retrieve the weights from a model which is found on the chain. First the function calls the ‘get_weights_chain’ function which was described previously and apply the returned values to the local model.



```
def set_weights_chain(self, learner_id, chain_learner_id):
    """ Set the weights, intercept and outcome of a learner from data downloaded from the chain.
        The service requesting the data must have paid to access such data.

        Args:
            learner_id (int): The local learner id to set the properties to.
            chain_learner_id (int): The chain learner id to set the properties from.
    """

    outcomes, intercept, weights = self.get_weights_chain(chain_learner_id)
    self.learners[learner_id][2].classes_ = outcomes
    self.learners[learner_id][2].intercept_ = intercept
    self.learners[learner_id][2].coef_ = weights
```

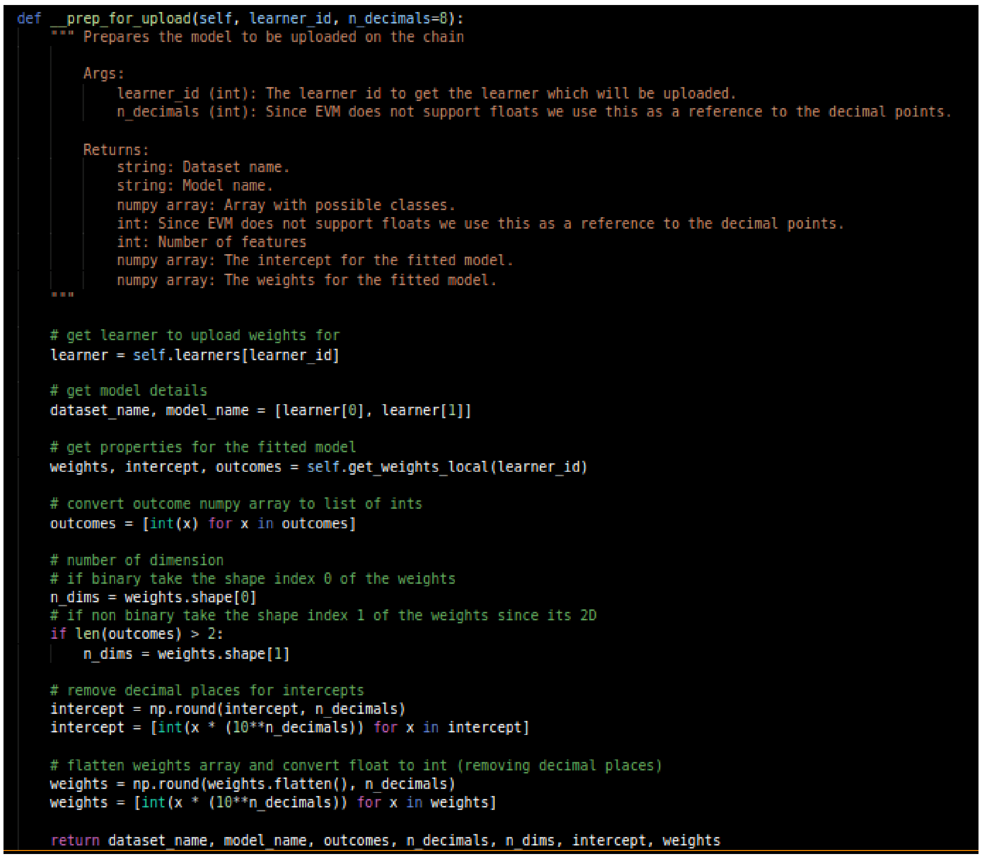
Figure 2.32: Set weights from the chain in the ’Service’ class in the ’mlconsultancy.py’.

Now we will look at the important functions found in the ‘Operator’ class. The first function ‘upload_model_chain’ as shown in Fig. 2.33 allows the operator/organization (the ones who deployed the contract) to upload a model’s weight from a local model in the collection to the chain. Before data could be uploaded on the chain it must be processed using the function ‘__prep_for_upload’ as shown in Fig. 2.34. In this function the necessary values are gathered and the floating-point values in both the ‘weights’ and ‘intercept’ values are transformed to integers for reasons described above. Once the data is processed the function adds this information on the chain for clients to pay for. The final function is called ‘update_model_chain’ which lets the operator update the weights on the chain for a specific model. Only the weights can be updated as shown in Fig. 2.33.



```python
def upload_model_chain(self, learner_id, n_decimals=8):
    """ Upload the properties of a local model found in the collection
        to the blockchain.

        Args:
            learner_id (int): The local learner id to set the properties from.
            chain_learner_id (int): The chain learner id to set the properties to.
            n_decimals (int): Since EVM does not support floats we use this as a reference to the decimal poin
    """

    # prepare for upload
    dataset_name, model_name, outcomes, n_decimals, n_dims, intercept, weights = self.__prep_for_upload(learne

    # upload on chain
    self.contract.functions.addModel(dataset_name,
                                     model_name,
                                     outcomes,
                                     n_decimals,
                                     n_dims,
                                     intercept,
                                     weights).transact(transaction={self.TXFROM : self.account_addr})

def update_model_chain(self, learner_id, chain_learner_id, n_decimals=8):
    """ Update the model found on the chain with the local version.

        Args:
            n_decimals (int): Since EVM does not support floats we use this as a reference to the decimal poin
    """

    # prepare for update/upload
    _, _, _, _, _, intercept, weights = self.__prep_for_upload(learner_id, n_decimals)

    # update model found on chain
    self.contract.functions.updateModel(chain_learner_id,
                                        intercept,
                                        weights).transact(transaction={self.TXFROM : self.account_addr})

#### client ##################################################################################
```

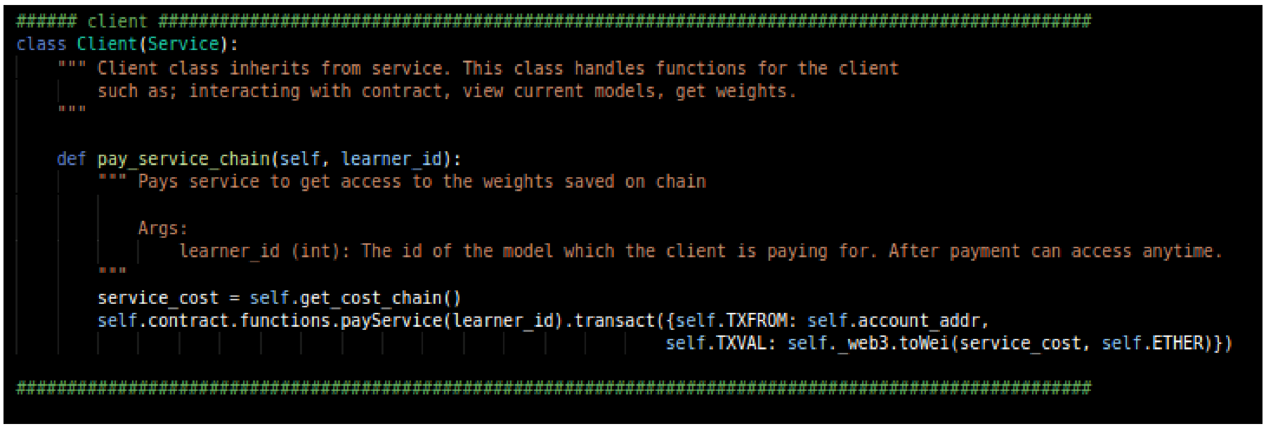
Figure 2.33: Add and Update model functions in the 'Operator' class in the 'mlconsultancy.py'.



```python
def __prep_for_upload(self, learner_id, n_decimals=8):
    """ Prepares the model to be uploaded on the chain

        Args:
            learner_id (int): The learner id to get the learner which will be uploaded.
            n_decimals (int): Since EVM does not support floats we use this as a reference to the decimal points.

        Returns:
            string: Dataset name.
            string: Model name.
            numpy array: Array with possible classes.
            int: Since EVM does not support floats we use this as a reference to the decimal points.
            int: Number of features
            numpy array: The intercept for the fitted model.
            numpy array: The weights for the fitted model.
    """

    # get learner to upload weights for
    learner = self.learners[learner_id]

    # get model details
    dataset_name, model_name = [learner[0], learner[1]]

    # get properties for the fitted model
    weights, intercept, outcomes = self.get_weights_local(learner_id)

    # convert outcome numpy array to list of ints
    outcomes = [int(x) for x in outcomes]

    # number of dimension
    # if binary take the shape index 0 of the weights
    n_dims = weights.shape[0]
    # if non binary take the shape index 1 of the weights since its 2D
    if len(outcomes) > 2:
        n_dims = weights.shape[1]

    # remove decimal places for intercepts
    intercept = np.round(intercept, n_decimals)
    intercept = [int(x * (10**n_decimals)) for x in intercept]

    # flatten weights array and convert float to int (removing decimal places)
    weights = np.round(weights.flatten(), n_decimals)
    weights = [int(x * (10**n_decimals)) for x in weights]

    return dataset_name, model_name, outcomes, n_decimals, n_dims, intercept, weights
```

Figure 2.34: Prepare data for upload function in the 'Operator' class in the 'mlconsultancy.py'.

The final function described ‘pay_service_chain’ as shown in Fig. 2.35 is found in the ‘Client’ class. Such function allows for a client instance to pay for a specific model found on the chain. The model id must be passed, and this refers to the id for the model which is on the chain.



```
###### client ##############################################################################
class Client(Service):
    """ Client class inherits from service. This class handles functions for the client
        such as; interacting with contract, view current models, get weights.
    """

    def pay_service_chain(self, learner_id):
        """ Pays service to get access to the weights saved on chain

            Args:
                learner_id (int): The id of the model which the client is paying for. After payment can access anytime.
        """
        service_cost = self.get_cost_chain()
        self.contract.functions.payService(learner_id).transact({self.TXFROM: self.account_addr,
                                                                  self.TXVAL: self._web3.toWei(service_cost, self.ETHER)})

############################################################################################
```

Figure 2.35: Pay for service function in the ’Client’ class in the ’mlconsultancy.py’.

Now we will look at the step by step process used to showcase the proof of concept. Code for this task can be found in the python script called ‘prototype.py’. The iris dataset was loaded from ‘sklearn.datasets’ and it was split into 80% which will be utilized by the ‘Operator’ and 20% which will be utilized by the ‘Client’. A new instance of the ‘Operator’ was instantiated, and the contract was deployed from this address using the function ‘deploy_contract’. The path for the compiled contract was passed as a parameter to this function. After doing so a ‘LogisticRegression’ model was added to the local collection of models in the ‘Operator’ instance using the ‘add_learner’ function. This local model was then fitted using the ‘fit_learner’ function. Weights for the fitted model were then uploaded to the chain using the ‘upload_model_chain’ function.

An instance for ‘Client’ was instantiated. The ‘get_details_chain’ function was called to view the available models which are offered by the consultancy (simulating the real-world environment). The model which was previously uploaded by the ‘Operator’ is shown in the list. A local model was added to the Client’s collection of local models which reflects the model which the Client intends to pay for (a Logistic Regression Model). The client pays for the service to view the weights for the selected model by calling the ‘pay_service_chain’ function. Before and after this call both the client and operator balance were printed to test that the transfer is being made. The client retrieves the weights it paid for from the chain and set these weights to the untrained local model by calling the ‘set_weights_chain’ function.

Using the 20% of data found on the client side the ‘predict_learner’ function was called, and the accuracy score was printed. Now we tested the model update function in the ‘Operator’ class. To do so the penalty for the local model in the operator’s collection was changed to ‘L1’ by calling the ‘set_params_learner’ function. The local model was then refitted and updated using the respective functions ‘fit_learner’ and ‘update_model_chain’. The client reupdates the weights by re-querying the chain using the ‘set_weights_chain’ function and the accuracy score with the new weights was noted.

Technologies used in the process:

- **Node.js:** To use the package manager 'npm' provided by Node.js to install JavaScript packages used in this task.
- **Truffle:** Installed using 'npm'. This was used to compile the contract which was used in this task. The contract could have been compiled using 'py-solc' from python but there was no support for solidity version greater or equal to 0.5. The solidity compiler could also have been used for this process but preferred to utilise truffle.
- **Ganache-cli:** Installed using 'npm', which is a command line interface for ganache. This allowed us to deploy the contract in a development/test environment.
- **Web3.py:** Installed using 'pip' which is a python library to interact with the network. This is a python wrapper of the Web3.js implementation.
- **OpenZepplin:** Installed using 'npm', a package which offers readily available contracts and interfaces. This was specifically installed to use the 'Ownable.sol' contract.

As we described our approach for the proof of concept, we would like to add some comments about this implementation. Since data in the Ethereum network is found on every full node, someone can find ways to read these weights found on the chain by not paying for the service at all. An approach to mitigate such risk is to encrypt the weights being uploaded by the consultancy agency. Once a client pays for the service the keys are sent via another means and the client can decrypt the information, but this solution reverts to having trust in a third party. Up till now there is now way to store secrets on the Ethereum chain. Another solution could be to use 'Codius', which provide a decentralized application layer which handles the use of secrets, but this project is in its early stages and it is very limited in functionality (also bugs are present).

2.5.6 Present (b)

Source code of the smart contract, code that handles the ML model training and upload of model parameters on smart contract, and client application (for the latter, showing both the code requesting the parameters and also the out-of-sample testing/results).

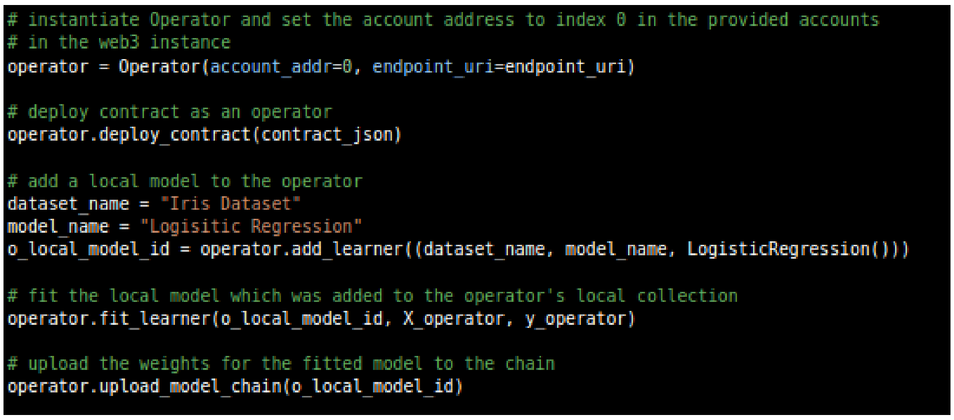
As described previous all the code used in this task can be found in:

'src/solidity/ml_consultancy_truffle'

The source code for the smart contract can be found in the location specified under the folder *'contracts'* and it is named as 'MlConsultancy.sol'.

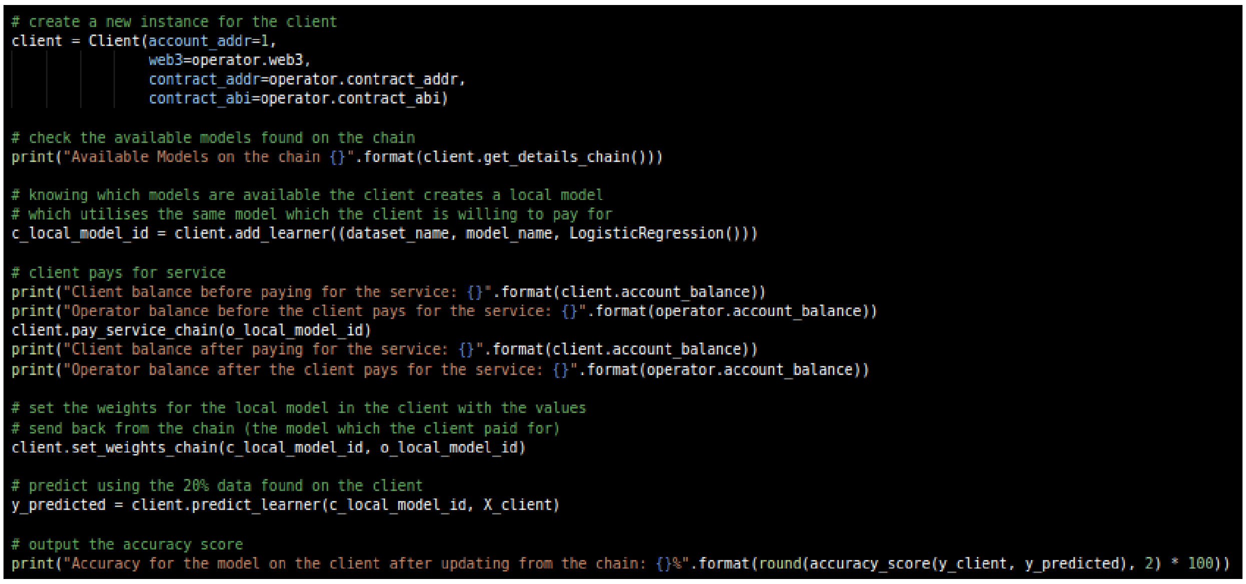
The code that handles the ML model training and upload of model parameters on smart contract can be found under the folder *'python'* in the 'ml_consultancy.py' script (described in the previous question). The code for the interaction between the client application is then showcased in the 'prototype.py' script which is found in the same folder. Below are some figures showing the interaction and the results in the proof of concept (interactions were previously described in detail in the previous answer).

Fig. 2.36 shows the code which handles the machine learning model training and the upload for the parameters. Fig. 2.37 shows the interaction between the client and the smart contract and in Fig. 2.38 the final outputs are shown.



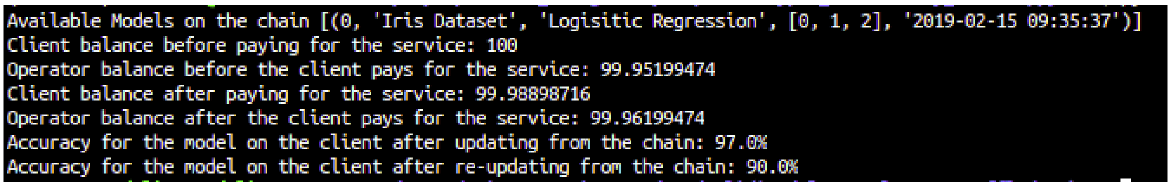
```
# instantiate Operator and set the account address to index 0 in the provided accounts
# in the web3 instance
operator = Operator(account_addr=0, endpoint_uri=endpoint_uri)

# deploy contract as an operator
operator.deploy_contract(contract_json)

# add a local model to the operator
dataset_name = "Iris Dataset"
model_name = "Logisitic Regression"
o_local_model_id = operator.add_learner((dataset_name, model_name, LogisticRegression()))

# fit the local model which was added to the operator's local collection
operator.fit_learner(o_local_model_id, X_operator, y_operator)

# upload the weights for the fitted model to the chain
operator.upload_model_chain(o_local_model_id)
```

Figure 2.36: Code which handles the machine learning model training and the upload for the parameters in ’prototype.py’.



```
# create a new instance for the client
client = Client(account_addr=1,
                web3=operator.web3,
                contract_addr=operator.contract_addr,
                contract_abi=operator.contract_abi)

# check the available models found on the chain
print("Available Models on the chain {}".format(client.get_details_chain()))

# knowing which models are available the client creates a local model
# which utilises the same model which the client is willing to pay for
c_local_model_id = client.add_learner((dataset_name, model_name, LogisticRegression()))

# client pays for service
print("Client balance before paying for the service: {}".format(client.account_balance))
print("Operator balance before the client pays for the service: {}".format(operator.account_balance))
client.pay_service_chain(o_local_model_id)
print("Client balance after paying for the service: {}".format(client.account_balance))
print("Operator balance after the client pays for the service: {}".format(operator.account_balance))

# set the weights for the local model in the client with the values
# send back from the chain (the model which the client paid for)
client.set_weights_chain(c_local_model_id, o_local_model_id)

# predict using the 20% data found on the client
y_predicted = client.predict_learner(c_local_model_id, X_client)

# output the accuracy score
print("Accuracy for the model on the client after updating from the chain: {}%".format(round(accuracy_score(y_client, y_predicted), 2) * 100))
```

Figure 2.37: Interaction between the client and the smart contract (requesting weights) in ’prototype.py’.



```
Available Models on the chain [(0, 'Iris Dataset', 'Logisitic Regression', [0, 1, 2], '2019-02-15 09:35:37')]
Client balance before paying for the service: 100
Operator balance before the client pays for the service: 99.95199474
Client balance after paying for the service: 99.98898716
Operator balance after the client pays for the service: 99.96199474
Accuracy for the model on the client after updating from the chain: 97.0%
Accuracy for the model on the client after re-updating from the chain: 90.0%
```

Figure 2.38: Final output produced by the ’prototype.py’ script.

References

[1] A. Meucci, “Quant nugget 2: Linear vs. compounded returns–common pitfalls in portfolio management,” 2010.

[2] T. Oliphant, “NumPy: A guide to NumPy,” USA: Trelgol Publishing, 2006–, [Online; accessed Feb 2019]. [Online]. Available: http://www.numpy.org/

[3] E. Jones, T. Oliphant, P. Peterson *et al.*, “SciPy: Open source scientific tools for Python,” 2001–, [Online; accessed Feb 2019]. [Online]. Available: http://www.scipy.org/

[4] S. Seabold and J. Perktold, “Statsmodels: Econometric and statistical modeling with python,” in *9th Python in Science Conference*, 2010.

[5] K. Sheppard *et al.*, “arch: Open source volatility models for Python,” 2014–, [Online; accessed Feb 2019]. [Online]. Available: https://arch.readthedocs.io/en/latest/index.html

[6] hmmlearn developers, “hmmlearn: Simple algorithms and models to learn hmms for Python,” 2010–, [Online; accessed Feb 2019]. [Online]. Available: https://hmmlearn.readthedocs.io/en/0.2.0/index.html

[7] C. Team, “A collection about awesome cryptokitties on the blockchain,” 2017–, [Online; accessed Feb 2019]. [Online]. Available: https://www.cryptokitties.co/

[8] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

[9] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[10] C. Dwork and M. Naor, “Pricing via processing or combatting junk mail,” in *Annual International Cryptology Conference.* Springer, 1992, pp. 139–147.

[11] A. Back *et al.*, “Hashcash-a denial of service counter-measure,” 2002.

[12] L. Lamport, R. Shostak, and M. Pease, “The byzantine generals problem,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[13] “Coindesk blockchain’s once-feared 51% attack is now becoming regular,” https://www.coindesk.com/blockchains-feared-51-attack-now-becoming-regular, accessed: 2019-02-13.

[14] O. developers, “OpenZepplin: a battle-tested framework of reusable smart contracts for ethereum and other evm and ewasm blockchains.” 2016–, [Online; accessed Feb 2019]. [Online]. Available: https://openzeppelin.org/