Monte Carlo Simulation for Estimating Financial Risk

- Import the necessary libraries.
- Create a Spark session, which is the entry point for all Spark applications. This allows users to perform operations over distributed datasets.

In [ ]:
```
import pyspark
import os
import sys
from pyspark import SparkContext
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable
from pyspark.sql import SparkSession

spark = SparkSession.builder.config("spark.driver.memory", "16g").appName('chapter_
```

Data Preparation

- The data is read from the csv files specified by the paths in the list.
- The data files contain a header and the schema is automatically inferred from the data.

In [ ]:
```
stocks = spark.read.csv(["data/stocksA/ABAX.csv","data/stocksA/AAME.csv","data/stoc
                        header='true', inferSchema='true')

stocks.show(2)
```

The table is transformed to include the column 'symbol' which is the type of stock extracted from the name of the input file of each record

In [ ]:
```
from pyspark.sql import functions as fun
stocks = stocks.withColumn("Symbol", fun.input_file_name()).\
    withColumn("Symbol",fun.element_at(fun.split("Symbol", "/"), -1)).\
    withColumn("Symbol",fun.element_at(fun.split("Symbol", "\."), 1))
stocks.show(2)
```

The steps are repeated to read the csv data into a dataframe called 'factors' and add the 'symbol' column extracted from the name of the input file

In [ ]:
```
factors = spark.read.csv(["data/stocksA/ABAX.csv","data/stocksA/AAME.csv","data/sto
                         header='true', inferSchema='true')
factors = factors.withColumn("Symbol", fun.input_file_name()).\
        withColumn("Symbol",fun.element_at(fun.split("Symbol", "/"), -1)).\
        withColumn("Symbol",fun.element_at(fun.split("Symbol", "\."), 1))
```

A window is defined to group the records in the dataframe by distinct values of the 'symbol' column, and for each window, an aggregation is performed to calculate the count of the records falling under each symbol. Only the the symbols with a count greater than 260*5+10 are retained, and the rest are filtered

A window function is used to perform statistical operations on groups of data in the dataframe, in this case, the separate symbols

```
In [ ]: from pyspark.sql import Window
        stocks = stocks.withColumn('count', fun.count('Symbol').\
            over(Window.partitionBy('Symbol'))).\
            filter(fun.col('count') > 260*5 + 10)
```

An SQL command is run which sets the spark.sql.legacy.timeParserPolicy to the mode LEGACY. This is a configuration parameter that controls parsing and formatting of dates and timestamps. In the LEGACY setting, extraneous string values that exceed the specified pattern are accepted and the hybrid Julian and Gregorian calendars are used

```
In [ ]: spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")
```

The string values in the 'Date' column of the stocks dataframe are replaced by their dates in the correct type.

- the to_timestamp() function converts a string of a pattern specified in the arguments to a timestamp
- to to_date() function converts a timestamp to a date

```
In [ ]: stocks = stocks.withColumn('Date',fun.to_date(fun.to_timestamp(fun.col('Date'),'dd-
            stocks.printSchema()
```

All records in the table falling before 23rd October 2009 or after 23rd October 2014 are filtered out of the dataframe and only the records which are dated within this time period are retained.

```
In [ ]: from datetime import datetime
        stocks = stocks.filter(fun.col('Date') >= datetime(2009, 10, 23)).\
            filter(fun.col('Date') <= datetime(2014, 10, 23))
```

Similarly, the string values in the 'Date' column of the factors table are converted to date types and the records are filtered to retain only those which are dated within the given time frame.

```
In [ ]: factors = factors.withColumn('Date',
        fun.to_date(fun.to_timestamp(fun.col('Date'),'dd-MMM-yy')))
        factors = factors.filter(fun.col('Date') >= datetime(2009, 10, 23)).\
            filter(fun.col('Date') <= datetime(2014, 10, 23))
```

The stocks and the factors dataframes are converted to Pandas dataframes using the toPandas() function. This is done becuase certain data manipulations are more efficiently done on Pandas dataframes than Spark Dataframes.

```
In [ ]: stocks_pd_df = stocks.toPandas()
        factors_pd_df = factors.toPandas()
        factors_pd_df.head(5)
```

Determining Factor Weights

- A function is defined to calculate the return over a time period as (final_value-initial_value)/initial_value
- The stocks table is grouped by the distinct values in the 'symbol' column. In each group, a rolling window is created. A rolling window allows a function to be performed on certain number of rows at a time, in this case 10. For each set of 10 rows in the dataframe, the returns on the 'Close' column are calculated using the defined function and added to the record of the 10th row in the window. This calculates the return for every set of 10-day periods. The window moves 1 step at a time and performs the function. Similarly, the returns on the 'Close' column are calculated for the factors table
- The index of the tables are reset (to be the default values of 0, 1, 2...) and the values are sorted in ascending order.

In [ ]:
```
n_steps = 10
def my_fun(x):
    return ((x.iloc[-1] - x.iloc[0]) / x.iloc[0])
stock_returns = stocks_pd_df.groupby('Symbol').Close.\
    rolling(window=n_steps).apply(my_fun)
factors_returns = factors_pd_df.groupby('Symbol').Close.\
    rolling(window=n_steps).apply(my_fun)
stock_returns = stock_returns.reset_index().\
    sort_values('level_1').\
    reset_index()
factors_returns = factors_returns.reset_index().\
    sort_values('level_1').\
    reset_index()
```

- The stock_returns and factor_returns columns created with the rollong window function are added to the original stocks and factors dataframe. Additionally, in the factors dataframe, an additional column is added, consisting of the square of the returns for each row.
- The factors dataframe is reshaped (pivoted) to use the 'date' column as the new index and the values of the 'symbol' column are used to make the columns of the new dataframe. Values from the 'factor_returns' and 'factors_returned_squred' columns are used to populate the new dataframe
- The columns of the new factors_with_returns dataframe are renamed to convert each column name to a string joined with its group name (here, th evalue from the 'symbol' column it falls under), and the index is reset to the default values

In [ ]:
```
stocks_pd_df_with_returns = stocks_pd_df.\
    assign(stock_returns = \
    stock_returns['Close'])
# Create combined factors DF
factors_pd_df_with_returns = factors_pd_df.\
    assign(factors_returns = \
    factors_returns['Close'],factors_returns_squared = \
    factors_returns['Close']**2)

factors_pd_df_with_returns = factors_pd_df_with_returns.\
    pivot(index='Date',
    columns='Symbol',
    values=['factors_returns', \
        'factors_returns_squared'])
factors_pd_df_with_returns.columns = factors_pd_df_with_returns.\
    columns.\
    to_series().\
    str.\
    join('_').\
    reset_index()[0]

factors_pd_df_with_returns = factors_pd_df_with_returns.\
    reset_index()
print(factors_pd_df_with_returns.head(1))
```

The column names of the factors_with_returns dataframe are displayed. Since the index was reset, the 'date' column is added to the list of columns in the dataframe

In [ ]:
```
print(factors_pd_df_with_returns.columns)
```

- The stock and factors dataframes are merged to combine the rows with same value in the 'Date' column. Since a left join is used, all the rows of the stocks tabel are included and only the matching rows of the factors table are added.
- The last 6 columns of the combined dataframe are selected as feature columns
- For the selected feature columns and the 'stock_returns' columns, all null values are dropped using the dropna() function. For this instruction, the pandas context is set to the mode 'use_inf_as_na' which considers any infinity values also to be null.
- A function is defined to find the Ordinary Least Squares Co-efficients for each stock.
    - This creates a linear regression module and fits it on the input dataframe to estimate the stock_returns using the values in the feature columns
    - Using the groupby() function, the dataframe is grouped by the distinct values of the stocks, and on each group, the function is applied to calculate the coefficients.
- A dataframe of coefficients per stock is created, which lists the coefficients for each feature in a separate column. Each row represents all the coefficients for a single type of stock.

In [ ]:
```
import pandas as pd
from sklearn.linear_model import LinearRegression

stocks_factors_combined_df = pd.merge(stocks_pd_df_with_returns,factors_pd_df_with_
feature_columns = list(stocks_factors_combined_df.columns[-6:])

with pd.option_context('mode.use_inf_as_na', True):
    stocks_factors_combined_df = stocks_factors_combined_df.\
        dropna(subset=feature_columns \
        + ['stock_returns'])

def find_ols_coef(df):
    y = df[['stock_returns']].values
    X = df[feature_columns]
    regr = LinearRegression()
    regr_output = regr.fit(X, y)
    return list(df[['Symbol']].values[0]) + \
        list(regr_output.coef_[0])

coefs_per_stock = stocks_factors_combined_df.\
    groupby('Symbol').\
    apply(find_ols_coef)

coefs_per_stock = pd.DataFrame(coefs_per_stock).reset_index()
coefs_per_stock.columns = ['symbol', 'factor_coef_list']
```

```
coefs_per_stock = pd.DataFrame(coefs_per_stock.\
        factor_coef_list.tolist(),index=coefs_per_stock.index,columns = ['Symbol']
coefs_per_stock
```

The factors_returns dataframe is filtered to select the values from the 'Close' column for only a single type of stock.

A Kernel Density Estimate plot is then created for this sample, which is a visualisation technique offering a view of the probability density of the values in the series.

In [ ]:
```
samples = factors_returns.loc[factors_returns.Symbol == \
    factors_returns.Symbol.unique()[0]]['Close']
samples.plot.kde()
```

Three separate series are created, consisting of the values from the 'Close' column for each separate kind of stock, by selecting only the rows which match the selected stock symbol.

This is displayed as a dataframe, with each series representing a column of the dataframe

In [ ]:
```
f_1 = factors_returns.loc[factors_returns.Symbol == \
    factors_returns.Symbol.unique()[0]]['Close']
f_2 = factors_returns.loc[factors_returns.Symbol == \
    factors_returns.Symbol.unique()[1]]['Close']
f_3 = factors_returns.loc[factors_returns.Symbol == \
    factors_returns.Symbol.unique()[2]]['Close']

print(f_1.size,len(f_2),f_3.size)

pd.DataFrame({'f1': list(f_1)[1:1040], 'f2': list(f_2)[1:1040], 'f3': list(f_3)}).c
```

- A covariance matrix is created using the .cov() function on the dataframe to compute the covariance of each column with the other.
- The means of each column are also calculated using the mean() function

In [ ]:
```
factors_returns_cov = pd.DataFrame({'f1': list(f_1)[1:1040],'f2': list(f_2)[1:1040]
        .cov().to_numpy()
factors_returns_mean = pd.DataFrame({'f1': list(f_1)[1:1040],'f2': list(f_2)[1:104
        mean()
```

The multivariate_normal method uses the means and covariances of multivariate variables (in this case, the 3 stock types) to draw random samples of these mutlivariate variables from a normal distribution.

Running the multivariate_normal method on the mean and covariance matrix created earlier will produce an array of 3 variables drawn from the normal distribution of theses variables.

In [ ]:
```
from numpy.random import multivariate_normal
multivariate_normal(factors_returns_mean, factors_returns_cov)
```

The dataframe containing the co-efficients fro each stock, the selected feature columns and the covariance matrix and mean values for the stocks are broadcasted so that they are cached in the memory of every machine on the cluster. This will avoid them having to be sent repeatedly to every machine that requires it.

In [ ]:
```
b_coefs_per_stock = spark.sparkContext.broadcast(coefs_per_stock)
b_feature_columns = spark.sparkContext.broadcast(feature_columns)
b_factors_returns_mean = spark.sparkContext.broadcast(factors_returns_mean)
b_factors_returns_cov = spark.sparkContext.broadcast(factors_returns_cov)
```

A list of 1000 integers starting from the base_seed value is created and converted to a dataframe. This is repartitioned to divide/distribute it among 1000 partitions, so each partition contains a set of seed values

In [ ]:
```
from pyspark.sql.types import IntegerType
parallelism = 1000
num_trials = 1000000
base_seed = 1496

seeds = [b for b in range(base_seed,base_seed + parallelism)]
seedsDF = spark.createDataFrame(seeds, IntegerType())
seedsDF = seedsDF.repartition(parallelism)
```

- A function is defined to calculate the returns for each trial:
  - A random integer is drawn using the randint() function and the seed is set to the input parameter.
  - A random sample of factors is drawn from the multivariate_normal function using the covariance matrix and means defined and broadcast previously
  - The returns for each stock are estimated by multiplying the coefficient for each feature with the value of the feature (the close value for each stock and the square of the close value for each stock, derived from the random sample drawn) and summing them for each stock
  - This is averaged for all the stocks by summing the returns for each and dividing by the number of stocks.
    - This is repeated for 1000 times on each partition (num_trials/parallelism) and a list consisting of the average returns from each trial is returned
    - This function is registered as a User-Defined function, returning an array of double values

In [ ]:
```
import random
from numpy.random import seed
from pyspark.sql.types import LongType, ArrayType, DoubleType
from pyspark.sql.functions import udf
def calculate_trial_return(x):
    trial_return_list = []
    for i in range(int(num_trials/parallelism)):
        random_int = random.randint(0, num_trials*num_trials)
        seed(x)
        random_factors = multivariate_normal(b_factors_returns_mean.value,b_factors
        coefs_per_stock_df = b_coefs_per_stock.value
        returns_per_stock = (coefs_per_stock_df[b_feature_columns.value] *(list(ran
        trial_return_list.append(float(returns_per_stock.sum(axis=1).sum()/b_coefs_
    return trial_return_list
udf_return = udf(calculate_trial_return, ArrayType(DoubleType()))
```

For each seed value in the seeds dataframe, the function defined in the previous cell is applied. Since there are 1000 values in this dataframe and the function conducts 1000 trials for each seed value, a total of 1000000 trials are carried out. The list of values returned by this function is flattened using teh explode function, to create a separate record for each value in the list with its corresponding seed value.

```
In [ ]: from pyspark.sql.functions import col, explode
        trials = seedsDF.withColumn("trial_return", udf_return(col("value")))
        trials = trials.select('value', explode('trial_return').alias('trial_return'))
        trials.cache()
```

Using all the values of the returns from the trials, the appoxQuantile() method can estimate the quantiles of the numerical values in the list. In this case, it approximates the value of the 5th percentile in the values for trail_returns

```
In [ ]:  trials.approxQuantile('trial_return', [0.05], 0.0)
```

The records are sorted to store the trial returns in ascending order. The limit function is used to retain only 1/20th of the records. An aggregation is carried over these records using the agg() function to return the average value of the trail returns in these records.

```
In [ ]: trials.orderBy(col('trial_return').asc()).\
        limit(int(trials.count()/20)).\
        agg(fun.avg(col("trial_return"))).show()
```

A line plot of all the trial return values is made to show their distribution. For this, the trials dataframe is converted to pandas and the plot.line() function is called to create the graph

```
In [ ]: import pandas
        mytrials=trials.toPandas()
        mytrials.plot.line()
```