monte-carlo

April 7, 2024

`python import pyspark import os import sys os.environ['PYSPARK_PYTHON'] = sys.executable os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable` Imports necessary libraries and sets environment variables for PySpark to use the correct Python executable. `python from pyspark.sql import SparkSession` Imports the SparkSession class from the pyspark.sql module, which is used to create a Spark session. `python spark = SparkSession.builder.config("spark.driver.memory", "16g").appName('chapter_8').getOrCreate()` Creates a SparkSession object named `spark` with the following configurations: - `config("spark.driver.memory", "16g")`: Sets the driver memory to 16GB. - `appName('chapter_8')`: Sets the application name to "chapter_8". - `getOrCreate()`: Gets an existing SparkSession or creates a new one if it doesn't exist.

[ ]:
```python
import pyspark
import os
import sys
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_8').getOrCreate()
```

`python stocks = spark.read.csv(["data/stocksA/ABAX.csv","data/stocksA/AAME.csv","data/stocksA/` `header='true', inferSchema='true')` This line reads multiple CSV files from the specified paths into a Spark DataFrame named `stocks`. The `header='true'` option tells Spark to use the first row of the CSV files as column names, and `inferSchema='true'` automatically infers the data types of the columns based on the data. `python stocks.show(2)` This line displays the first two rows of the `stocks` DataFrame.

[ ]:
```python
stocks = spark.read.csv(["data/stocksA/ABAX.csv","data/stocksA/AAME.csv","data/stocksA/AEPI.csv"], header='true', inferSchema='true')
stocks.show(2)
```

This code is written in PySpark, a Python library for working with Apache Spark, a distributed computing framework for big data processing. `python from pyspark.sql import functions as fun` This line imports the `functions` module from the `pyspark.sql` package and assigns it an alias `fun`. `python 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))` This line operates on a DataFrame named `stocks`. It creates a new column named

"Symbol" and populates it with the file name from which each row was read. It then splits the file name on the "/" character and takes the last element (which should be the stock symbol). Finally, it splits the resulting string on the "." character and takes the first element (removing any file extension). `python factors = spark.read.csv(["data/stocksA/ABAX.csv","data/stocksA/AAME.csv","data/stocksA/AEPI.csv"], header='true', inferSchema='true')` This line reads multiple CSV files from the specified paths into a new DataFrame named `factors`. The `header='true'` option tells Spark to use the first row as the column names, and `inferSchema='true'` tells Spark to infer the data types of the columns automatically. `python 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))` This line is similar to the one for `stocks`, but it operates on the `factors` DataFrame. It creates a new column named "Symbol" and populates it with the file name from which each row was read, then extracts the stock symbol from the file name using the same logic as before.

[ ]:
```python
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))

factors = spark.read.csv(["data/stocksA/ABAX.csv","data/stocksA/AAME.csv","data/
 ↪stocksA/AEPI.csv"], 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))
```

`python from pyspark.sql import Window` Imports the `Window` class from the `pyspark.sql` module, which is used for performing window functions in PySpark. `python stocks = stocks.withColumn('count', fun.count('Symbol').over(Window.partitionBy('Symbol'))).filter(fun.col('count') > 260*5 + 10)` Adds a new column 'count' to the `stocks` DataFrame, which counts the number of occurrences of each 'Symbol' using the `count` window function. The `Window.partitionBy('Symbol')` partitions the data by 'Symbol' before applying the count. The resulting DataFrame is then filtered to keep only rows where the 'count' is greater than 260*5 + 10 (1310). `python spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")` Sets the `spark.sql.legacy.timeParserPolicy` configuration to 'LEGACY' in the SparkSession. `python stocks = stocks.withColumn('Date',fun.to_date(fun.to_timestamp(fun.col('Date'),'dd-MMM-yy'))) stocks.printSchema()` Converts the 'Date' column in the `stocks` DataFrame from a string in the format 'dd-MMM-yy' to a date type using `to_date` and `to_timestamp` functions. The `printSchema()` method is then called to print the schema of the updated `stocks` DataFrame. `python from datetime import datetime stocks = stocks.filter(fun.col('Date') >= datetime(2009, 10, 23)).filter(fun.col('Date') <= datetime(2014, 10, 23))` Imports the `datetime` class from the `datetime` module. The `stocks` DataFrame is then filtered to keep only rows where the 'Date' is between October 23, 2009, and October 23, 2014, inclusive. "'python factors = factors.withColumn('Date', fun.to_date(fun.to_timestamp(fun.col

[ ]:
```
from pyspark.sql import Window
stocks = stocks.withColumn('count', fun.count('Symbol').over(Window.
 ↪partitionBy('Symbol'))).filter(fun.col('count') > 260*5 + 10)
spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")

stocks = stocks.withColumn('Date',fun.to_date(fun.to_timestamp(fun.
 ↪col('Date'),'dd-MMM-yy')))
stocks.printSchema()

from datetime import datetime
stocks = stocks.filter(fun.col('Date') >= datetime(2009, 10, 23)).filter(fun.
 ↪col('Date') <= datetime(2014, 10, 23))

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))

stocks_pd_df = stocks.toPandas()
factors_pd_df = factors.toPandas()
factors_pd_df.head(5)
```

`n_steps = 10` sets the window size for rolling calculations to 10. `python def my_fun(x): return ((x.iloc[-1] - x.iloc[0]) / x.iloc[0])` This function calculates the percentage change between the last and first values in a given window `x`. It subtracts the first value from the last, divides by the first value, and returns the result. `python 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)` These lines calculate the rolling percentage change for the 'Close' column of `stocks_pd_df` and `factors_pd_df`, grouped by 'Symbol'. The `rolling` function creates a window of size `n_steps`, and `apply` applies the `my_fun` function to each window. `python stock_returns = stock_returns.reset_index().sort_values('level_1').reset_index() factors_returns = factors_returns.reset_index().sort_values('level_1').reset_index()` These lines reset the index of the resulting DataFrames, sort by the 'level_1' column (likely a time index), and reset the index again to create a new index column.

[ ]:
```
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()
```

`stocks_pd_df_with_returns = stocks_pd_df.assign(stock_returns = stock_returns['Close'])` adds a new column 'stock_returns' to the `stocks_pd_df` DataFrame, which contains the 'Close' values from the `stock_returns` DataFrame. `python factors_pd_df_with_returns = factors_pd_df.assign(factors_returns = factors_returns['Close'], factors_returns_squared = factors_returns['Close']**2)` This line adds two new columns to the `factors_pd_df` DataFrame: 'factors_returns' containing the 'Close' values from the `factors_returns` DataFrame, and 'factors_returns_squared' containing the squared values of 'factors_returns'. `python factors_pd_df_with_returns = factors_pd_df_with_returns.pivot(index='Date', columns='Symbol', values=['factors_returns', 'factors_returns_squared'])` This pivots the `factors_pd_df_with_returns` DataFrame, setting the 'Date' column as the index, the 'Symbol' column as the columns, and the 'factors_returns' and 'factors_returns_squared' columns as the values. `python factors_pd_df_with_returns.columns = factors_pd_df_with_returns.columns.to_series().str.join('_').reset_index()[0]` This line modifies the column names of the pivoted DataFrame by joining the multi-level column names with an underscore. `factors_pd_df_with_returns = factors_pd_df_with_returns.reset_index()` resets the index of the pivoted DataFrame, creating a new column for the former index values. The last two lines print the first row of the `factors_pd_df_with_returns` DataFrame and its column names.

[ ]:
```
stocks_pd_df_with_returns = stocks_pd_df.assign(stock_returns =␣
↪stock_returns['Close'])
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))
print(factors_pd_df_with_returns.columns)
```

`import pandas as pd` and `from sklearn.linear_model import LinearRegression` import the necessary libraries for data manipulation and linear regression modeling. `python stocks_factors_combined_df = pd.merge(stocks_pd_df_with_returns, factors_pd_df_with_returns, how="left", on="Date")` This line merges two dataframes `stocks_pd_df_with_returns` and `factors_pd_df_with_returns` on the "Date" column using a left join. `python feature_columns = list(stocks_factors_combined_df.columns[-6:])` This line creates a list of the last 6 column names from the merged dataframe, which will be used as feature columns. `python with pd.option_context('mode.use_inf_as_na', True): stocks_factors_combined_df = stocks_factors_combined_df.dropna(subset=feature_columns + ['stock_returns'])` This block of code drops rows with missing values (NaN or inf) in the feature columns and the 'stock_returns' column from the merged dataframe. `python 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])` This function takes a dataframe `df` as input, extracts the 'stock_returns' column as the target variable `y`, and the feature columns as the predictor variables `X`. It then fits a linear regression model using `LinearRegression()` from scikit-learn and returns a list containing the stock symbol and the coefficients of the linear regression model. "'python 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_

[ ]:
```
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_returns,how="left", on="Date")
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'] + feature_columns)
coefs_per_stock
```

`python samples = factors_returns.loc[factors_returns.Symbol == factors_returns.Symbol.unique()[0]]['Close']` This line filters the `factors_returns` DataFrame to select the 'Close' column for the first unique symbol in the 'Symbol' column. `python samples.plot.kde()` This line plots a kernel density estimation (KDE) curve for the `samples` data using the `plot.kde()` method from Pandas. A KDE curve is a non-parametric way to estimate the probability density function of a random variable.

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

The code extracts closing prices for three different symbols from a DataFrame `factors_returns` and performs some operations on them. `python 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'] These lines extract the 'Close' column for the first three unique symbols in the DataFrame `factors_returns`. python print(f_1.size,len(f_2),f_3.size) This line prints the size of `f_1`, length of `f_2`, and size of `f_3`. python pd.DataFrame({'f1': list(f_1)[1:1040], 'f2': list(f_2)[1:1040], 'f3': list(f_3)}).corr() This line creates a new DataFrame with columns 'f1', 'f2', and 'f3' containing the closing prices from index 1 to 1039 for the three symbols. It then calculates the correlation between these columns. python factors_returns_cov = pd.DataFrame({'f1': list(f_1)[1:1040], 'f2': list(f_2)[1:1040], 'f3': list(f_3)}).cov().to_numpy() factors_returns_mean = pd.DataFrame({'f1': list(f_1)[1:1040], 'f2': list(f_2)[1:1040], 'f3': list(f_3)}).mean() These lines create a covariance matrix and a mean vector for the closing prices of the three symbols from index 1 to 1039.

[ ]:
```python
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)}).corr()

factors_returns_cov = pd.DataFrame({'f1': list(f_1)[1:1040], 'f2': list(f_2)[1:
 ↪1040], 'f3': list(f_3)}).cov().to_numpy()
factors_returns_mean = pd.DataFrame({'f1': list(f_1)[1:1040], 'f2': list(f_2)[1:
 ↪1040], 'f3': list(f_3)}).mean()
```

python 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) This code is broadcasting several variables (`coefs_per_stock`, `feature_columns`, `factors_returns_mean`, and `factors_returns_cov`) across the cluster using the `broadcast` method of the `SparkContext`. Broadcasting is a way to efficiently distribute large read-only values across the cluster. Instead of sending the entire data to each executor, Spark sends a single copy of the data to each executor node, which can then be cached and used by all tasks running on that node. This can significantly reduce the communication overhead and memory usage, especially when dealing with large datasets or models. The `broadcast` method returns a `Broadcast` object, which can be accessed on the executors using the same variable name (`b_coefs_per_stock`, `b_feature_columns`, etc.). This allows the broadcasted data to be accessed efficiently by all tasks running on the executors.

[ ]:
```python
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)
```

python from pyspark.sql.types import IntegerType This line imports the IntegerType class from the pyspark.sql.types module, which is used to define the data type of a column in a Spark DataFrame. python parallelism = 1000 num_trials = 1000000 base_seed = 1496 These lines set the values of three variables: parallelism (set to 1000), num_trials (set to 1000000), and base_seed (set to 1496). python seeds = [b for b in range(base_seed, base_seed + parallelism)] This line creates a list seeds containing integers from base_seed to base_seed + parallelism - 1. python seedsDF = spark.createDataFrame(seeds, IntegerType()) This line creates a Spark DataFrame seedsDF from the seeds list, with a single column of IntegerType. python seedsDF = seedsDF.repartition(parallelism) This line repartitions the seedsDF DataFrame into parallelism (1000) partitions, which can improve performance for certain operations by distributing the data across multiple partitions.

[ ]:
```
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)
```

This code defines a Python function calculate_trial_return and a Spark UDF udf_return based on that function. python import random from numpy.random import seed These lines import the random module from Python's standard library and the seed function from NumPy's random module. python from pyspark.sql.types import LongType, ArrayType, DoubleType from pyspark.sql.functions import udf These lines import various types and functions from PySpark's SQL module, which are used to define the UDF. python 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_returns_cov.value) coefs_per_stock_df = b_coefs_per_stock.value returns_per_stock = (coefs_per_stock_df[b_feature_columns.value] *(list(random_factors) + list(random_factors**2))) trial_return_list.append(float(returns_per_stock.sum(axis=1 return trial_return_list This function calculates trial returns based on various inputs (num_trials, parallelism, b_factors_returns_mean, b_factors_returns_cov, b_coefs_per_stock, b_feature_columns). It generates random factors using the multivariate_normal function, calculates returns per stock based on these factors and coefficients, and appends the sum of these returns to a list. The function returns this list of trial returns. python udf_return = udf(calculate_trial_return, ArrayType(DoubleType())) This line creates a Spark UDF udf_return from

[ ]:
```
import random
from numpy.random import seed

from pyspark.sql.types import LongType, ArrayType, DoubleType
from pyspark.sql.functions import udf
```

```python
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_returns_cov.value)
        coefs_per_stock_df = b_coefs_per_stock.value
        returns_per_stock = (coefs_per_stock_df[b_feature_columns.value] *(list(random_factors) + list(random_factors**2)))
        trial_return_list.append(float(returns_per_stock.sum(axis=1).sum()/b_coefs_per_stock.value.size))
    return trial_return_list
udf_return = udf(calculate_trial_return, ArrayType(DoubleType()))
```

This code is written in PySpark, a Python library for working with Apache Spark, a distributed computing framework for big data processing. `python from pyspark.sql.functions import col, explode` This line imports the `col` and `explode` functions from the `pyspark.sql.functions` module. `col` is used to reference a column in a DataFrame, and `explode` is used to create a new row for each element in an array or map column. `python trials = seedsDF.withColumn("trial_return", udf_return(col("value")))` This line creates a new DataFrame `trials` by adding a new column `"trial_return"` to the existing DataFrame `seedsDF`. The values in this new column are computed by applying the `udf_return` function (which is not shown) to the `"value"` column. `python trials = trials.select('value', explode('trial_return').alias('trial_return'))` This line modifies the `trials` DataFrame by selecting the `"value"` column and the exploded `"trial_return"` column, which is aliased as `"trial_return"`. `python trials.cache()` This line caches the `trials` DataFrame in memory for faster access. `python trials.approxQuantile('trial_return', [0.05], 0.0)` This line computes the approximate 5th percentile of the `"trial_return"` column in the `trials` DataFrame. `python trials.orderBy(col('trial_return').asc()).limit(int(trials.count()/20)).agg(fun.avg(col("trial_` This line performs the following operations: 1. Orders the `trials` DataFrame by the `"trial_return"` column in ascending order. 2. Limits the DataFrame to the first `trials.count()/20` rows (5% of the total rows). 3. Aggregates the `"trial_return"` column by computing the average value using the `avg` function from the `fun` module (not shown). 4.

[ ]:
```python
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()

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

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

`python import pandas` This line imports the pandas library, which is a popular data manipulation and analysis library for Python. `python mytrials=trials.toPandas()` This line assumes that `trials` is an object with a `toPandas()` method that converts it to a pandas DataFrame. The resulting DataFrame is assigned to the variable `mytrials`. `python mytrials.plot.line()` This line uses the pandas plotting functionality to create a line plot of the data in the `mytrials` DataFrame. The `plot.line()` method is a shortcut for creating a line plot, which is a common way to visualize data over time or other continuous variables.

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