ETL Report

Group 2

November 10, 2021

Extraction

A complete list of all our data sources can be accessed through our group's GitHub repository. The sources are located in the file named *DataSources.pdf*. All of our files were downloaded as CSV files and uploaded directly into our group's container for storage and easy access among group members. Our *group2-capstone* container is located in the Microsoft Azure storage account named *gen10dbcdatalake*. Once we uploaded all of our data files into our container, we were able to read each CSV file into an Azure Databrick for cleaning and transformation. Also, for each of the bank stock data CSVs, we extracted data going back to July 2, 2007.

Transformation

This section will be broken down into three lists: one for the census data, one for our stock data, and one specifically for our ML model (this part will follow the Load section of this report). For a clearer explanation of these transformations, please look towards our code folder within GitHub.

Bank Stock Data:

1) We initially uploaded all of our bank stock CSVs in a folder labeled *Banks*. Then, we created a mount point to our *group2-capstone* container to read our data into a Spark data frame. We read all our bank CSVs at once using *. In our line of code, we also used *header = True.*

2) Next, we imported Pandas. From our Spark data frame, we used *toPandas()* to create a Pandas data frame.

3) With our Pandas data frame, we converted the *Open, Close, High, Low,* and *Volume* columns into float types using *pd.to_numeric.* We also converted the *Date* string into a datetime object. For the date conversion, we used *pd.to_datetime* and *dt.date.*

4) Now that all our bank stock data was in one data frame, we were now able to read in our S&P index data. This data was located in a folder titled *S&P.* We used the mount point we created earlier and read the S&P CSV into a Spark data frame. In our line of code, we also used *header =True.*

5) From our S&P Spark data frame, we used *toPandas()* to create a Pandas data frame.

6) Following this step, the column headers had awkward leading spaces, so we removed these spaces to make the headers the same as the headers in our bank data frame. However, we eventually changed these again for merging purposes.

7) Using our S&P Pandas data frame, we converted the *Open, Close, High,* and *Low* columns into float types using *pd.to_numeric*. We also converted the *Date* string into a datetime object. For the date conversion, we used *pd.to_datetime* and *dt.date*.

8) After these conversions, we sorted the S&P data frame by *Date* with *ascending = True.* We sorted this data frame by *Date* because we wanted to create a new column for percent change between each day using the closing price. To accomplish this, we created a new column named *SP_Perc* and used the function *pct_change(periods = 1).* This created a null value which we dealt with later.

9) Like the changes we performed for the S&P data frame, we sorted the bank data frame by *Date* with *ascending = True,* and we grouped the rows of the data frame by *Ticker*. This allowed us to find the daily percent change for each bank. Again, we used the *pct_change()* function to obtain our new *Perc* column.

10) After creating the new column, we once again sorted the bank data frame by *Date*. This allowed us to confirm that the percent change function worked correctly for each stock.

11) The next step was to merge the S&P and banks data frames. Before we could do this, we needed to rename some of the S&P data frame's columns. For this, we changed the S&P's *Open, Close, High, Low,* and *Ticker* to *SP_Open, SP_Close, SP_High, SP_Low,* and *SP_Ticker,* respectively.

12) To accomplish the actual merge of the data frames, we used the *merge* function and performed a left join on *Date*.

13) To save our newly created/merged data frame, we created a new mount point. We then converted our final Pandas data frame into a Spark data frame. Finally, we wrote the data frame as a CSV with *header = True*.

14) After saving our data frame as a CSV in our *2021-11-08-comparison* folder, we moved on to making our Producer.

15) We once again ran the code to create our mount point (same as step 1). We then read in all the CSV files from the *2021-11-08-comparison* folder using Spark. Once again, we used *header = True* in the read-in code. (For clarification, we have provided our code files on GitHub).

16) We imported *StringType, FloatType, and DoubleType* from *pyspark.sql.types*. We used the *withColumn*() function and *cast()* function to change some of our column types. We changed columns *Close, High, Date, Low, Open, Volume, SP_Close, SP_Open, SP_High, SP_Low, SP_Perc,* and *Perc* to *FloatType, FloatType, StringType, FloatType, FloatType, DoubleType, FloatType, FloatType, FloatType, FloatType, FloatType,* and *FloatType,* respectively.

17) Next, we built our Producer with the use of code provided to us in the shared Bootcamp folder in Azure Databricks. The file was named kafka_producer. Our confluentTopicName was *group2_pipelinerun.* We sent the data through the Producer one row at a time and with 0.5 seconds between sends. These rows get sent to a dictionary named *df_dict.*

18) We then moved on to our Consumer. Working off the sample code that was provided to us through the shared Bootcamp folder in Azure Databricks (kafka_consumer_get_all_*messages*), we once again made our confluentTopicName *group2_pipelinerun.* Following the Kafka Class setup, we input our Topic name in a *c.subscribe()* function.

19) We then use a while loop (based on code provided in *kafka_consumer_get_all_messages*) to append the components of *df_dict* to a list named *kafkaDictionaries.*

20) We then create a new mount point to send the data to the data lake.

21) We needed to change the types once more before we converted the data back into a Spark data frame. From *pyspark.sql.types,* we imported *StructType, StructField, StringType, FloatType, LongType,* and *DoubleType.* The schema for this data frame was created using the *StructType()* and *StructField()* functions. We made *Date, Ticker,* and *SP_Ticker* all *StringType, Volume* became a *DoubleType, timestamp* became a *LongType,* and all other variables became *FloatType.*

22) After step 21, we wrote the data frame as a CSV to our pipeline folder in our storage container. For this line of code, we also used *header = True.*

Census Data:

1) Most of the cleaning for the census data can be completed in Power Query. To start, after downloading the Finance and Insurance: Summary Statistics for the U.S., States, and Selected Geographies: 2017 from the Census data website and saving it as a CSV, we brought it into Excel by going to the Get and Transform Data tab in the Data ribbon. We selected from Text/CSV and selected our downloaded table.

2) We brought the table into Power Query. We removed the Geo_ID, Geo_ID_Footnote, Name, NAICS_Footnote columns.

3) We made the first row our headers. This is done in Power Query by going to the Transform tab in the Home ribbon.

4) Now, we selected the NAICS2017_Label column. Still in the Home ribbon, we selected Remove Rows à Remove Duplicates in the Reduce Rows tab.

5) Next, we filtered the Firm column to have the value “D” at the top. We counted the number of rows with the “D” values, selected Remove Rows in the Home ribbon, selected Remove Rows from Top, and inputted the number of rows with “D.”

6) While playing with some potential visualizations, we realized we needed to condense our rows. We decided that it would be best to view the smallest subcategories within each of the four major categories. To do so, we scanned through each row to see if its NAICS code had a smaller branch. If it did have a smaller branch, we removed the row. At the end we were left with 35 rows including headers.

7) After narrowing our dataset, we decided it would be a good idea to include the major category that each subcategory branched from. To do so, we manually made changes to our dataset, but for ETL purposes we suggest the following method.

8) We selected the NASICS2017 column. Then in the Add Column ribbon, we went to the From Text tab and selected Extract. Here we clicked First Characters and input three. This provides us with the primary category coded for each subcategory NAICS2017_Label. We named this column LabelCategory

9) To ultimately get the name of all the major categories and not just the codes, we added a conditional column (Add Column à General à Conditional Column). We named the column LabelName. For our conditions, we input that if the LabelCategory equals 521 to output Monetary authorities (central bank). The rest were as follows: 522 equals Credit intermediation and related activities, 523 equals Securities, commodity contracts, and other financial investments and related activities, and 524 equals Insurance carriers and related activities.

10) We now closed and applied our changes.

Load

This section will discuss how we moved our datasets into a SQL database. As you may have noticed we have not mentioned the transformations used for our Machine Learning model or an ERD. Our ERD will be in an attached file within our GitHub repository. Our ML model transformations will be discussed following this Load section.

SQL Loading:

1) We once again started by creating a new mount point for our SQL data.

2) We read in the data frame created in the Consumer process (We are reading in data from the pipeline folder mentioned above).

3) After reading in our Spark data frame, we realized that we once again needed to correct the types for our columns. From *pyspark.sql.types,* we imported *StringType, FloatType, LongType,* and *DateType.* Using the *withColumn()* function, we made *Date* a *DateType, timestamp* became a *LongType,* and all other variables became *FloatType.* We then ordered the data frame by *Date* and *Ticker.*

4) We removed the first day of trading because the percent change column in the data frame had null values for the first day.

5) Now, we created our SQL tables. The first table we created was the S&P data table. From *pyspark.sql.functions,* we imported *col* to select our columns (*Date, SP_Close, SP_Open, SP_High, SP_Low, SP_Perc).* We dropped the duplicates because each day of S&P data was duplicated six times in the original dataset (per stock). We also ordered the table by *Date*.

6) To create an S&P ID for our S&P table, we used the following code: *sp_data.withColumn('SP_ID', row_number().over(Window.orderBy(monotonically_increasing_id())))*. Then we rearranged our columns using the *select()* function listing *SP_ID* first.

7) After setting up our data frame accordingly, we wrote the table to our server using the connection code provided to us. This table is named *dbo.SP_data.*

8) The second table we set up was the stock data table. Using *col* from *pyspark.sql.functions*, we selected our columns (*Date, Close, Open, High, Low, Perc, Ticker, Volume).* We ordered the table by *Date* and *Ticker*.

9) To create an ID column for our stock table, we used the following code: *bank_data.withColumn('Stock_ID', row_number().over(Window.orderBy(monotonically_increasing_id())))*. This gives us an ID for each row of the table. Then we rearranged our columns using the *select()* function listing *Stock_ID* first.

10) The third table we set up was the ticker table. Using the *select()* function, we selected our *Ticker* column. Next, we dropped any duplicate rows. To create an ID column for our ticker table, we used the following code: *ticker.withColumn('Ticker_ID', row_number().over(Window.orderBy(monotonically_increasing_id())))*. Then we rearranged our columns using the *select()* function listing *Ticker_ID* first.

11) Having set up the ticker table and stock data table, we now needed to join them before sending them to SQL with our S&P data table. We joined the two tables using the *join()* function, and we joined them on *Ticker*.

12) The newly combined table named *bank_data2* is written into our SQL server as *dbo.stock_data.*

13) We also wanted to add the ticker table to our SQL server. The table was written to SQL as *dbo.ticker_data.*

14) Finally, we wanted to send our census data table to SQL as well. First, we read our census table into the databrick as a Spark data frame. We again used *header = True* in our code,

15) We needed to change our column types. From *pyspark.sql.types*, we imported *IntegerType.* Using the *withColumn()* function, we made *FIRM, ESTAB, PAYANN, EMP, RCPTOT, PAYQTR1,* and *NAICS2017* all *IntegerType.* We then selected all of the columns we needed to send to SQL.

16) We wrote the table to our SQL server as *dbo.census_data.*

17) Once in SQL, we needed to make some changes to our tables. For *dbo.ticker_data*, we used *ALTER* to make *Ticker_ID* not null and then made it the primary key. For *dbo.stock_data*, we used *ALTER* to make *Stock_ID* not null and then made it the primary key. For *dbo.SP_data*, we used *ALTER* to make *SP_ID* not null and then made it the primary key. For *dbo.census_data*, we used *ALTER* to make *NAICS2017* not null and then made it the primary key.

18) In SQL, a view was created that joined *dbo.stock_data* and *dbo.SP_data* on the column *Date*, and *dbo.ticker_data* was joined on the *Ticker_ID* column in order to get the Ticker name joined to the view. This view was then used for the dashboard and the machine learning model.

ML Transformations:

1) After creating all of our tables and views in our SQL server, it was time to set up our ML model. In a Jupyter Notebook, we imported Pandas as pd, Pymssql, and numpy as np. We connected to our server using the resources provided by Tom and selected all columns from our dbo.stock_table view. This table was read as a Pandas data frame.

2) We then broke the date column into *Year*, *Month*, and *Day* columns and converted them to datetime objects using *pd.to_datetime().*

3) We also needed to add a new column named *C_Weighted* for the exponentially weighted moving average of the daily closing stock price. This was performed by grouping the rows by *Ticker* and using the *ewm* function (a Pandas function).

4) We also added a column named *Comparison* which compared each stock's percent change with the S&P percent change. This is the column we are looking to predict in our model. This column yielded a 1 or 0 depending on if the stock change was higher or lower, respectively.

5) For each stock, we also added a Market Capitalization column. For JPM and GS, the value was Large, for NYCB and BOH, the value was small, and for DFS and SNV, the value was Mid.

6) We created a column to show the actual percent difference between the stock percent change and S&P percent change. This column was named *Perc_diff.*

7) We then added five more columns to see the percent change for the previous five days for each stock. For this we used a *groupby()* function and a *shift()* function.

8) We then located where the null values were and worked with a dataset that did not include the nulls. We ultimately did not remove null values from the original data frame.

9) When working with the actual ML model, we dropped different columns in our testing process. For our finalized ML model, we dropped *High, Low, Close, Open, SP_Close, SP_Open, SP_High, SP_Low, Perc_diff, Volume, Year, Ticker, Ticker_ID, Market,* and *Date.* However, before dropping the market column, we created dummy columns for the market values (Large, Mid, Small). For this we used the *get_dummies()* function associated with Pandas.

10) After these changes, we were ready to run our ML model.