Term Project

Lincoln Brown

DSC650-T301

Professor Ismaily

Introduction:

For this project, I will be taking an advertising dataset found on Kaggle that contains advertising spend vs sales for different forms of advertising (tv, radio, and newspaper). I will be saving this dataset to GitHub and then using the raw download URL to process the data in Nifi before sending it to HDFS for storage. From HDFS, the data will flow into Hive, where it can be queried from PySpark. Using a SparkSession, I will be running a LinearRegression on the dataset and then exporting the metrics to HBase.

Dataset:

<https://www.kaggle.com/datasets/brsahan/advertising-spend-vs-sales>

Pipeline Overview:

The data will be ingested as an HTTP request using an InvokeHTTP processor in NiFi. From there, the data will be sent to a PutHDFS processor also in my NiFi Flow. After the data has been stored in HDFS, I will create a table schema in Hive and then ingest the data into the table. After the table has been populated with data, I will use PySpark to access the Hive table and retrieve the records, storing them in a dataframe that will be converted into a vector to create a Linear Regression model. After the LR model has been created, trained, and evaluated, I will store the model’s performance metrics in HBase.

Issues Encountered:

I ran into a couple of different issues while working on this project. They stemmed from the limited resources available on the VM.

The first issue I ran into was when I was ingesting the data into NiFi. I wanted to confirm that the data was saving to HDFS before I closed NiFi. However, leaving NiFi open while trying to open HDFS proved to be too resource intensive for the VM and I encountered a freeze. This happened a couple of times before I realized my error and started closing out NiFi each time I was done ingesting the data.

The second issue I ran into was when I was in PySpark I was ingesting the data from the table in Hive using a SparkSession. I didn’t think it made sense to close out the SparkSession when I was done retrieving the data from Hive, since I was just going to be using another SparkSession to connect to HBase. However, this also caused the VM to freeze and I started closing out the first SparkSession and creating a second separate SparkSession to connect to HBase after I was done ingesting the data from Hive.

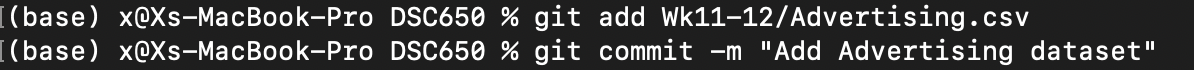
Overall, I ran into some issues because I would have to start from square one everytime I ran “docker-compose down -d” in Hadoop. I believe this is because the VM is using a Docker container and I am not rewriting the Docker container when I am going through the project. I found that if I just stop the VM, it will close all of the components except for Zookeeper, which when I ran “docker-compose up -d” in the Hadoop folder, it would have all of my data still intact.

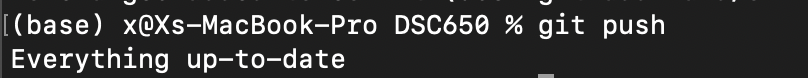
I still ended up going through the entire data flow multiple times, but I am grateful because it helped me get really familiar with the entire project more and I fully understood the commands as I typed them in. On this note, I did not use any copy/paste scripts in my project. I followed along with the examples and I used the NiFi template provided, making changes as I needed. However, I manually typed all commands so that I could better understand what exactly each line was doing and why it was needed.

Screenshot Walkthrough and Code:

Step 1: Upload data to GitHub for ingestion into NiFi

I uploaded the file to GitHub by downloading it first from Kaggle to my local computer, then using the commands:





Step 2: Ingest data into NiFi using InvokeHTTP to PutHDFS processors.

Data is ingested from invoking the raw URL for the Advertising.csv directly from GitHub, it is then processed by PutHDFS and added to the Hadoop Distributed File System (HDFS).

A screenshot of a computer

Description automatically generated

I used the template but customized it to my own needs.

In the InvokeHTTP processor, I changed the HTTP URL to point to the raw download URL for the dataset on my GitHub.

A screenshot of a computer

Description automatically generated

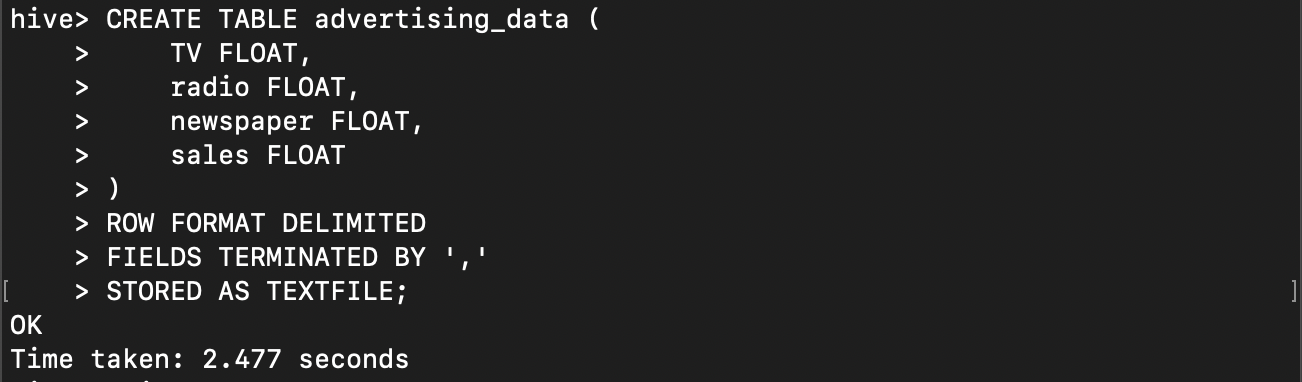
In the PutHDFS processor, I changed the Hadoop Configuration Resources path to reflect the correct user path. This required changing from the user “ubuntu” to “x” which then fixed the warning icon in the processor. I also changed from the /tmp directory to a new directory labeled /project. I used the command “hdfs dfs -mkdir /project” in bash of the master docker container to create the project folder.

A screenshot of a computer

Description automatically generated

Step 3: Create Hive Table and Ingest data from HDFS

After the data was stored on HDFS, I logged into hive and created a table through there.

  
Once the table was created, it was time to load data stored on HDFS into the table.

A black screen with white text

Description automatically generated

Step 4: SparkSession to import data from Hive  
  
Following that, I created a Spark Session to import the database from Hive. At first, I just wanted to be able to access the database to ensure it was working properly.   
A screenshot of a computer

Description automatically generated

Step 6: Create HBase table

Since, I was going to be performing ML on the dataset, I logged into HBase and created a new table:

A computer screen with white text

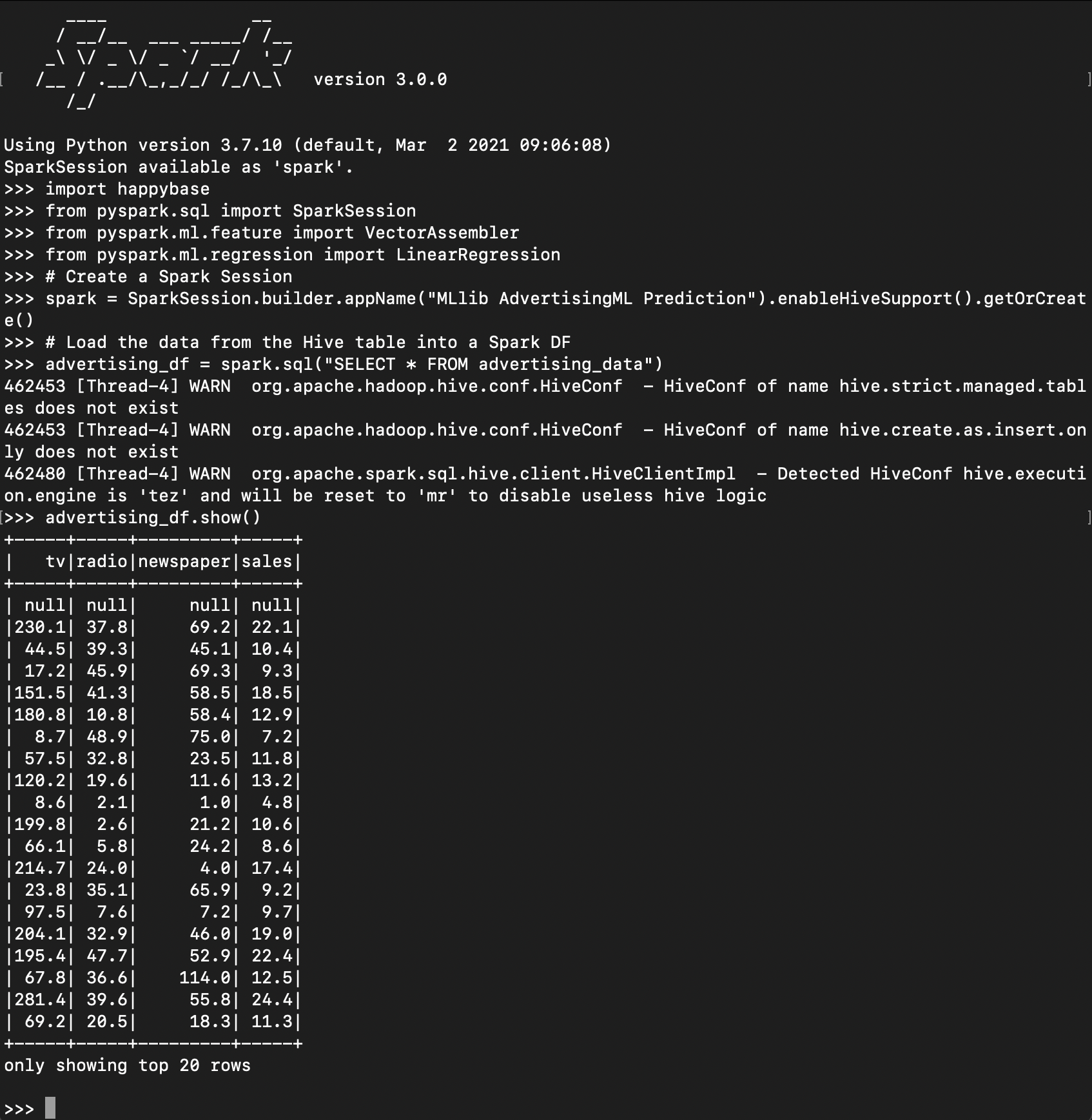
Description automatically generated

Step 7: SparkSession – Get data from Hive and run Linear Regression

I then logged back into Spark to continue.

After confirming that I had successfully loaded the dataset into Spark, I decided to go further and run a Linear Regression on the data.

This was in between days, so my project had been closed and I had to reopen the Spark session.



Above, I have loaded the data and verified that it was loaded into Spark correctly.

Next, I assembled the features into a vector, split the data into a train and test set, created a linear regression model, fit the model onto the training set, and then evaluated it and printed the Root Mean Squared Error and the R^2 score.   
A screenshot of a computer program

Description automatically generated

A computer screen shot of a black screen

Description automatically generated

Step 8: Save Performance Metrics to HBase.

After printing the metrics, it was time to save them into a list and then write them to my ‘advertising\_table’ in HBase.

A screen shot of a computer

Description automatically generated

A screen shot of a computer screen

Description automatically generated

Finally, I needed to confirm that the metrics were successfully written to HBase:

A computer screen with white text

Description automatically generated

This concluded my Term Project.

Conclusion:

In summary, I took an advertising dataset that contained the different sales performance for different forms of advertising (tv, radio, newspaper). I retrieved this dataset from Kaggle, saved it my local PC, and then hosted it on GitHub. The reasoning for uploading it to GitHub was to provide a raw data URL that was easily accessible by the NiFi InvokeHTTP processor. This aspect of the problem is used to simulate getting raw HTTP requests from a data stream available on the Internet.

From the InvokeHTTP processor, the data flowed into the PutHDFS processor (also part of my NiFi flow). This processor actively listened for messages from the InvokeHTTP processor and then stored them on HDFS located on the Google Cloud VM. This required specifying the correct configuration resource and creating a new project directory on HDFS. To make the new directory I used the command:

“hdfs dfs -mkdir /project”

After the data was stored in HDFS, it was time to create a table in Hive and load the data from the file stored in HDFS. This is an important step because the data stored on HDFS is not a table that can be queried with SQL. HDFS is for data storage, not querying. That is where Hive comes into play. It was necessary to create a table schema in Hive and then load the data into that table so that we could interact with it through PySpark.

In between performing ML with PySpark, I knew that I would need a table to store the metrics from the machine learning in HBase, so at this point, I went ahead and created a table in HBase named advertising\_table.

Once the data was loaded into Hive, I started a PySpark session. Using SparkSession to create a SparkSession with Hive support enabled. Using this session, I pulled the data from Hive and then stored it in a dataframe. Using this dataframe, I was able to transform its features into a vector, which was subsequently split into train and test sets, that I then used to create a Linear Regression model.

The linear regression model was trained on the training set and evaluated using the test set. This generated test metrics. For the purpose of this project, I focused only on RMSE and R2 score. These metrics were then saved to a list variable named data that contained tuples with the relevant metrics information and the naming schema for the HBase table. I then opened a new Spark Session. At first, I was attempting to just reuse the SparkSession I had been using with Hive. However, this caused me lots of difficulties and crashed my VM several times. I decided to not reuse the SparkSession and just create a new one. After I did that, my issues were resolved.

I then created a function to write to performance metrics to the hbase partitions. This function handled connecting to HBase using the happybase library. After the function was defined, it was time to use a Resilient Distributed Dataset (RDD) to parallelize the data and write it to HBase.

Using rdd = spark.sparkContext.parallelize(data), I was able to parallelize the data variable into an rdd that split the data into multiple partitions that would be distributed across the modes in the cluster for parallel processing. Using foreachPartition(write\_to\_hbase\_partition) ensured that the function was applied to each partition of the RDD. Finally, I was able to verify that my metrics were stored correctly in HBase.

This project seemed very formidable at first, but as I began working on it, I became much more familiar with the different Apache components and how they work together. I had issues with freezing initially because I didn’t want to shut off NiFi until I had verified that my data was being stored in HDFS. This caused my VM to freeze and I would have to manually stop it and get reconnected. Eventually, I figured out that it was overloaded and I started just stopping NiFi before I logged into HDFS.

I also ran into issues with my SparkSessions and trying to write data to HBase. I was trying to use the same SparkSession that I had used with Hive and it kept causing error messages that would iterate endlessly, despite using Ctrl+C to cancel the process. This ended up with me also having to manually restart the VM before I realized that I needed to use a different SparkSession for reading from Hive/performing ML and writing the metrics to HBase.

I used the example template that you provided for NiFi, and then I adjusted it to fit my needs. Other than that, I just typed everything, I did not make scripts. When I am learning new things I like to manually type them in instead of copy/pasting because it helps me remember the commands better and I think more about what I am typing in rather than just waiting for an error to crop up because I forgot to change a variable when copy/pasting. Therefore, with my mistakes described above, I went through the full flow of data throughout the project **a lot.** It was very helpful for my learning and I’m actually glad that I ran into issues because after I had to rebuild everything multiple times, it made more sense after each iteration.