**Analyzing Hotel Booking Cancellations with HDFS, Hive, and Spark**

J. Ryan Weaver

Bellevue University

DSC-650: Big Data

Nasheb Ismaily

In this comprehensive report, we journeyed through the realms of Hadoop Distributed File System (HDFS), Apache Hive, and Apache Spark to dissect, analyze, and predict hotel booking cancellations using the Hotel Bookings dataset. Each component is pivotal in handling big data, enabling efficient querying, and empowering machine learning capabilities.

**1. Data Loading:**

Our journey commences with acquiring the Hotel Bookings dataset from GitHub, a task accomplished by utilizing wget to fetch the data and hdfs dfs -put to transfer it to HDFS. The orchestration of our Hadoop ecosystem is simplified with Docker, facilitated by docker-compose up -d and accessing the master node using docker-compose exec master bash.

**2. Hive – A Data Warehousing Solution:**

As we navigate through vast datasets, Hive emerges as our first ally. With its SQL-like interface, Hive facilitates the creation of structured data tables. We initiate our journey by crafting a table through the CREATE TABLE statement, dictating the schema of our dataset.

The decision to CLUSTER BY and BUCKET the table by country is strategic. Clustering boosts data locality, optimizing query performance. Binning data into eight buckets enhances parallelism and further accelerates query processing. Our choice of ROW FORMAT DELIMITED and FIELDS TERMINATED BY ',' ensures seamless parsing of the CSV data, with STORED AS TEXTFILE aligning it conveniently for Hadoop's file handling. To ensure data quality, tblproperties("skip.header.line.count"="1") skips the header line during ingestion.

Subsequently, we import our data into the Hive table, employing LOAD DATA INPATH to seamlessly transfer the contents of 'hotel\_bookings.csv' into our structured Hive table.

**3. In-Depth Analysis with Hive:**

Harnessing Hive's analytical capabilities, we delve into the intricacies of room allocation dynamics. By investigating the correlation between reserved and assigned room types, we unearth patterns that go beyond mere booking statistics. Filtering the results to consider only room types below the average reservation frequency provides a nuanced understanding of room allocation trends.

Taking a step further, we construct a view to calculate the average duration of stay based on matching reserved and assigned room types. This materialized view optimizes performance and provides a persistent snapshot, which is crucial for time-sensitive analyses.

**4. Spark – Unleashing the Power of Distributed Computing:**

The second leg of our journey introduces Spark, a robust distributed computing engine. As our dataset grows, Spark accelerates data processing, fosters iterative algorithms, and facilitates machine learning operations.

Creating a Spark session bridges the gap between Spark and Hive, enabling seamless integration. Our choice to enable Hive support ensures fluid data interchange between the two components.

**5. Machine Learning with Spark:**

Our analytical journey takes a machine learning detour as we harness the predictive capabilities of Spark. Transferring data from Hive to Spark for model training, we prepare the dataset using the Vector Assembler. This transformation aggregates features into a single vector, readying the data for machine learning algorithms.

A pivotal point is partitioning our data into training and testing sets. With an 80/20 split, we ensure adequate training while preserving a substantial dataset for robust model evaluation.

**6. Logistic Regression and Decision Trees:**

The logistic regression model steps onto the stage first. Trained on our prepared dataset, the Binary Classification Evaluator evaluates the model. Calculating the area under the ROC curve provides insights into the model's predictive accuracy. With a resulting score of 0.538, we understand the model's discrimination capabilities.

Next in line is the Decision Tree model, following a similar evaluation metric. The resultant area under the ROC curve, pegged at 0.486, paints a distinct picture. A comparative analysis between logistic regression and decision trees unfolds, shedding light on their relative strengths and weaknesses.

**7. Conclusion:**

Our journey through HDFS, Hive, and Spark encapsulates the entire data lifecycle, from acquisition and storage to analysis and prediction. The strategic use of each component highlights its unique capabilities, catering to specific needs within the data analytics spectrum.

In conclusion, the harmonious collaboration of HDFS, Hive, and Spark provides a robust infrastructure for handling big data analytics and machine learning. The intricate dance between structured storage, SQL-like querying, and distributed computing has unveiled patterns, trends, and predictive insights in the realm of hotel booking cancellations, empowering decision-makers with actionable intelligence.

**Hotel Booking Dataset Explanation:**

Summary: This data set contains booking information for a city hotel and a resort hotel, and includes information such as when the booking was made, length of stay, the number of adults, children, and/or babies, and the number of available parking spaces, among other things.

• **Hotel:** Hotel (H1 = Resort Hotel or H2 = City Hotel)

• **Is\_canceled:** Value indicating if the booking was canceled (1) or not (0)

• **Lead\_time:** Number of days that elapsed between the entering date of the booking into the PMS and the arrival date

• **Arrival\_date\_year:** Year of arrival date

• **Arrival\_date\_month:** Month of arrival date

• **arrival\_date\_week\_number:** Week number of year for arrival date

• **arrival\_date\_day\_of\_month:** Day of arrival date

• **stays\_in\_weekend\_nights:** Number of weekend nights (Saturday or Sunday) the guest stayed or booked to stay at the hotel

• **stays\_in\_week\_nights:** Number of week nights (Monday to Friday) the guest stayed or booked to stay at the hotel

• **adults:** Number of adults

• **children:** Number of children

• **babies:** Number of babies

• **meal:** Type of meal booked. Categories are presented in standard hospitality meal packages: Undefined/SC – no meal package; BB – Bed & Breakfast; HB – Half board (breakfast and one other meal – usually dinner); FB – Full board (breakfast, lunch and dinner)

• **country:** Country of origin. Categories are represented in the ISO 3155–3:2013 format

• **market\_segment:** Market segment designation. In categories, the term “TA” means “Travel Agents” and “TO” means “Tour Operators”

• **distribution\_channel:** Booking distribution channel. The term “TA” means “Travel Agents” and “TO” means “Tour Operators”

• **is\_repeated\_guest:** Value indicating if the booking name was from a repeated guest (1) or not (0)

• **previous\_cancellations:** Number of previous bookings that were cancelled by the customer prior to the current booking

• **previous\_bookings\_not\_canceled:** Number of previous bookings not cancelled by the customer prior to the current booking

• **reserved\_room\_type:** Code of room type reserved. Code is presented instead of designation for anonymity reasons.

• **assigned\_room\_type:** Code for the type of room assigned to the booking. Sometimes the assigned room type differs from the reserved room type due to hotel operation reasons (e.g. overbooking) or by customer request. Code is presented instead of designation for anonymity reasons.

• **booking\_changes:** Number of changes/amendments made to the booking from the moment the booking was entered on the PMS until the moment of check-in or cancellation

• **deposit\_type:** Indication on if the customer made a deposit to guarantee the booking. This variable can assume three categories: No Deposit – no deposit was made; Non Refund – a deposit was made in the value of the total stay cost; Refundable – a deposit was made with a value under the total cost of stay.

• **Agent:** ID of the travel agency that made the booking

• **Company:** ID of the company/entity that made the booking or responsible for paying the booking. ID is presented instead of designation for anonymity reasons

• **days\_in\_waiting\_list:** Number of days the booking was in the waiting list before it was confirmed to the customer

• **customer\_type:** Type of booking, assuming one of four categories: Contract - when the booking has an allotment or other type of contract associated to it; Group – when the booking is associated to a group; Transient – when the booking is not part of a group or contract, and is not associated to other transient booking; Transient-party – when the booking is transient, but is associated to at least other transient booking

• **adr:** Average Daily Rate as defined by dividing the sum of all lodging transactions by the total number of staying nights

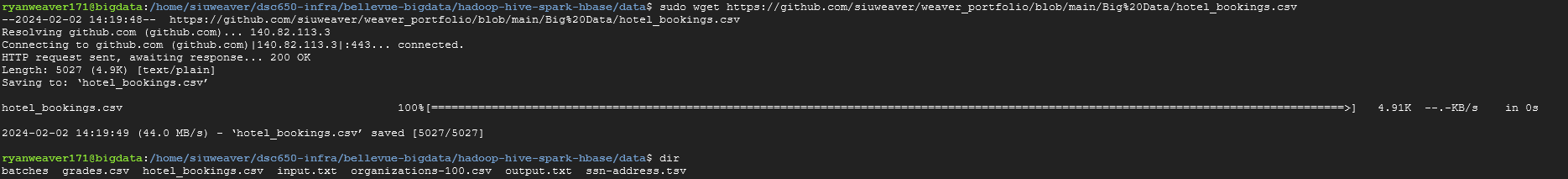
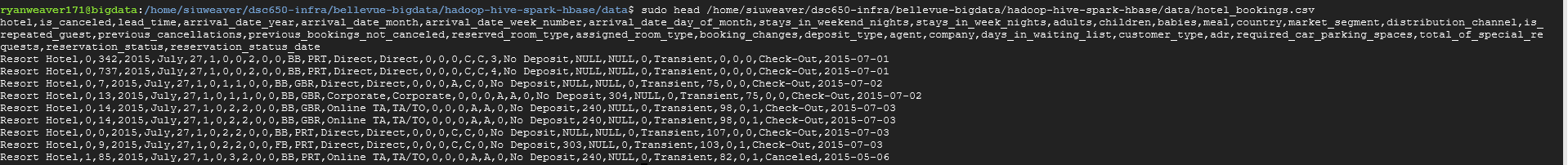
• **required\_car\_parking\_spaces:** Number of car parking spaces required by the customer

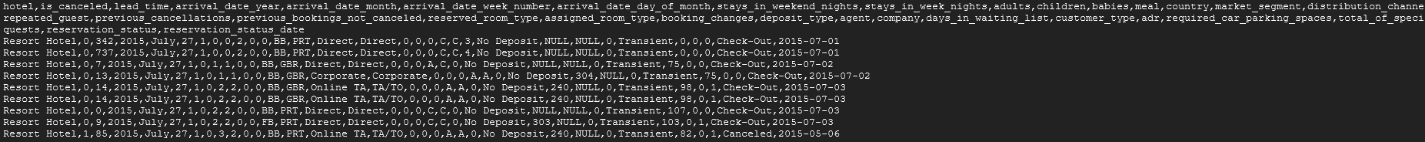
• **total\_of\_special\_requests:** Number of special requests made by the customer (e.g. twin bed or high floor)

• **reservation\_status:** Reservation last status, assuming one of three categories: Canceled – booking was canceled by the customer; Check-Out – customer has checked in but already departed; No-Show – customer did not check-in and did inform the hotel of the reason why

• **reservation\_status\_date:** Date at which the last status was set. This variable can be used in conjunction with the ReservationStatus to understand when was the booking canceled or when did the customer checked-out of the hotel

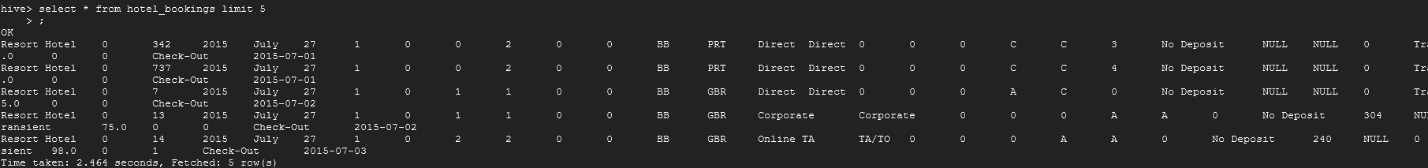
**Walkthrough With Screenshots:**

1. Load Data
   1. cd /home/siuweaver/dsc650-infra/bellevue-bigdata/hadoop-hive-spark-hbase/data
   2. sudo wget https://raw.githubusercontent.com/siuweaver/weaver\_portfolio/main/Big%20Data/hotel\_bookings.csv
   3. 
   4. sudo head /home/siuweaver/dsc650-infra/bellevue-bigdata/hadoop-hive-spark-hbase/data/hotel\_bookings.csv
   5. 
2. Environment Initialization
   1. cd ..
      1. Return to hadoop-hive-spark-hbase
   2. sudo docker-compose up -d
   3. sudo docker-compose exec master bash
3. Load
   1. hdfs dfs -put /data/hotel\_bookings.csv /
4. Confirm data is loaded
   1. hdfs dfs -ls /
   2. A black screen with white text

      Description automatically generated
   3. hdfs dfs -cat /hotel\_bookings.csv | head
   4. 
5. Start Hive
   1. hive
6. Create a Hive Table

|  |
| --- |
| CREATE TABLE hotel\_bookings (  hotel STRING,  is\_canceled INT,  lead\_time INT,  arrival\_date\_year INT,  arrival\_date\_month STRING,  arrival\_date\_week\_number INT,  arrival\_date\_day\_of\_month INT,  stays\_in\_weekend\_nights INT,  stays\_in\_week\_nights INT,  adults INT,  children INT,  babies INT,  meal STRING,  country STRING,  market\_segment STRING,  distribution\_channel STRING,  is\_repeated\_guest INT,  previous\_cancellations INT,  previous\_bookings\_not\_canceled INT,  reserved\_room\_type STRING,  assigned\_room\_type STRING,  booking\_changes INT,  deposit\_type STRING,  agent INT,  company INT,  days\_in\_waiting\_list INT,  customer\_type STRING,  adr DOUBLE,  required\_car\_parking\_spaces INT,  total\_of\_special\_requests INT,  reservation\_status STRING,  reservation\_status\_date STRING  )  CLUSTERED BY (country) INTO 8 BUCKETS  ROW FORMAT DELIMITED  FIELDS TERMINATED BY ','  STORED AS TEXTFILE  tblproperties("skip.header.line.count"="1"); |

1. Confirm table
   1. show tables;
   2. A black background with white text

      Description automatically generated
2. Load Data into table
   1. LOAD DATA INPATH '/hotel\_bookings.csv' INTO TABLE hotel\_bookings;
3. Confirm Loading
   1. select \* from hotel\_bookings limit 5;
   2. 
4. Create View for Average Room Type Match
   1. View stores the percent of the time the reserved room type matches the assigned room type for all reservations that weren’t cancelled. This will allow us to compare the overall average to the value for each individual room type

|  |
| --- |
| CREATE VIEW IF NOT EXISTS avg\_reserve\_room\_type\_match AS  SELECT sum(case when reserved\_room\_type = assigned\_room\_type then 1 else 0 end) / count(\*) AS perc  FROM hotel\_bookings  WHERE is\_canceled = 0; |

* 1. Validate view
     1. select \* from avg\_reserve\_room\_type\_match;
     2. A screenshot of a computer

        Description automatically generated
  2. Return reserved room types below average

|  |
| --- |
| select reserved\_room\_type, count(\*) reserve\_count, sum(case when reserved\_room\_type = assigned\_room\_type then 1 else 0 end) / count(\*) AS room\_type\_perc, arm.perc  from hotel\_bookings hb  join avg\_reserve\_room\_type\_match arm  where is\_canceled = 0  group by reserved\_room\_type, arm.perc  having sum(case when reserved\_room\_type = assigned\_room\_type then 1 else 0 end) / count(\*) < arm.perc; |

* + 1. A screenshot of a computer

       Description automatically generated
  1. Create Materialized View

|  |
| --- |
| CREATE MATERIALIZED VIEW avg\_stay\_duration  AS  SELECT hotel, AVG(stays\_in\_weekend\_nights + stays\_in\_week\_nights) AS avg\_duration  FROM hotel\_bookings  GROUP BY hotel; |

* + - 1. Validate results
         1. Select \* from avg\_stay\_duration;
         2. A screen shot of a computer

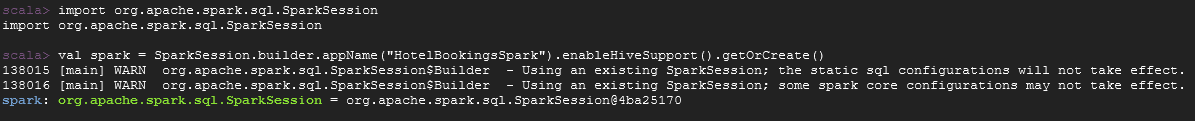
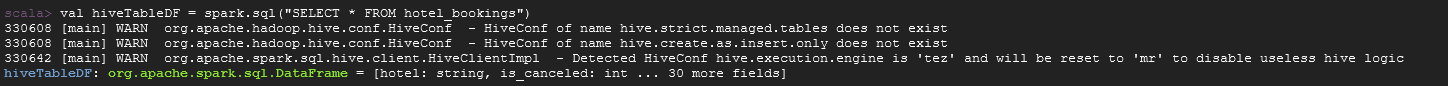
            Description automatically generated
    1. View Trending over Time

|  |
| --- |
| SELECT  arrival\_date\_year,  arrival\_date\_month,  COUNT(\*) AS bookings\_count  FROM hotel\_bookings  GROUP BY arrival\_date\_year, arrival\_date\_month  ORDER BY arrival\_date\_year, arrival\_date\_month; |

* + 1. A screenshot of a computer

       Description automatically generated

Spark-Shell

1. spark\_shell
   1. 
2. import org.apache.spark.sql.SparkSession
   1. This line of code imports the SparkSession class from the Apache Spark SQL library, allowing the creation of a unified entry point to access data and execute Spark functionalities in a Spark application.
3. val spark = SparkSession.builder.appName("HotelBookingsSpark").enableHiveSupport().getOrCreate()
   1. This code initializes a SparkSession named "HotelBookingsSpark" with Hive support enabled, allowing seamless integration with Hive and creating a unified environment for executing Spark operations.
   2. 
4. Read Hive Table into Spark DataFrame:
   1. val hiveTableDF = spark.sql("SELECT \* FROM hotel\_bookings")
   2. 
   3. Confirm data loaded
      1. hiveTableDF.show(10)
      2. A black and white screen with white dots

         Description automatically generated
5. Prepare data for model
   1. Import Vector Assembler
      1. import org.apache.spark.ml.feature.VectorAssembler
   2. Identify Feature Columns into an array
      1. val featureCols = Array("lead\_time", "stays\_in\_weekend\_nights", "stays\_in\_week\_nights", "adults", "children", "babies")
   3. Vector Assembler
      1. val assembler = new VectorAssembler().setInputCols(featureCols).setOutputCol("features").setHandleInvalid("keep")
         1. This code defines a VectorAssembler named "assembler" that combines specified input columns into a single feature column named "features," preserving invalid values during the assembly process.
      2. val assembledData = assembler.transform(data)
         1. This code applies the previously defined VectorAssembler ("assembler") to transform the input DataFrame "data," creating a new DataFrame named "assembledData" with the assembled features.
   4. validate assembled data
      1. assembledData.show(10)
      2. A black screen with white text

         Description automatically generated
   5. Train/Test Split
      1. val Array(trainData, testData) = assembledData.randomSplit(Array(0.8, 0.2), seed = 123)
6. Train Logistic Regression
   1. Import and Train
      1. import org.apache.spark.ml.classification.LogisticRegression
      2. val lr = new LogisticRegression().setLabelCol("is\_canceled").setFeaturesCol("features")
         1. This code initializes a Logistic Regression model ("lr") and specifies the label and features columns for training, with the label column set to "is\_canceled" and the features column set to "features."
      3. val model = lr.fit(trainData)
   2. Predict on Test Data
      1. val predictions = model.transform(testData)
      2. Validate Output
         1. Predictions.show(10)
         2. A black screen with white text

            Description automatically generated
   3. Evaluate Model
      1. import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
      2. val evaluator = new BinaryClassificationEvaluator().setLabelCol("is\_canceled").setRawPredictionCol("rawPrediction").setMetricName("areaUnderROC")
         1. This code creates a Binary Classification Evaluator ("evaluator") for evaluating the Logistic Regression model, specifying the label column as "is\_canceled," raw prediction column as "rawPrediction," and metric name as "areaUnderROC."
      3. val auc = evaluator.evaluate(predictions)
      4. show evaluation
         1. println(s"Area under ROC curve: $auc")
         2. A black screen with white text

            Description automatically generated
7. Train Decision Tree
   1. Import and Train
      1. import org.apache.spark.ml.classification.DecisionTreeClassifier
      2. val dt = new DecisionTreeClassifier().setLabelCol("is\_canceled").setFeaturesCol("features")
      3. val dt\_model = dt.fit(trainData)
   2. Predict
      1. val dt\_predictions = dt\_model.transform(testData)
      2. A black and white screen with white text

         Description automatically generated
   3. Evaluate
      1. val dt\_evaluator = new BinaryClassificationEvaluator().setLabelCol("is\_canceled").setRawPredictionCol("rawPrediction").setMetricName("areaUnderROC")
      2. val dt\_auc = dt\_evaluator.evaluate(dt\_predictions)
      3. println(s"Area under ROC curve: $dt\_auc")

**References:**

Antonio, N. (2019, February). Hotel Booking Demand, Version 1. Retrieved February 4, 2024 from https://www.kaggle.com/datasets/jessemostipak/hotel-booking-demand/data.