NYC Airbnb's: Travel Stay Recommendations

1st Siju Chacko 2nd Mythri Shivakumar 3rd Rebecca Abraham

Phase 3

1. Data Cleaning Steps

The following pre-processing and cleaning steps were performed on the Airbnb dataset:

- **Handling Duplicates:**
  - Identified the number of duplicate rows before cleaning.
  - Removed duplicate rows using the dropDuplicates() method.
- **Type Conversion:**
  - Converted the following columns to appropriate numeric data types using the withColumn() method along with the cast() method:
    - price (float)
    - minimum_nights (integer)
    - number_of_reviews (integer)
    - reviews_per_month (float)
    - calculated_host_listings_count (double)
    - availability_365 (double)
    - latitude (double)
    - longitude (double)

  All of these conversions are required for the distributed processing when using the Machine learning algorithms to train the dataset.
- **Handling Missing Values:**
  Handled the missing values first finding the missing values using the filter() and col functions within the withColumn() method.
  - Filled missing values in the name column with "No Name Provided".
  - Filled missing values in the host_name column with "Unknown Host".
  - Replaced missing dates in the last_review column with the default value "2000-01-01".
  - Filled missing values in the reviews_per_month column with 0.
  - Computed the median value for the price column and filled missing values with the median.

  This step will help improve the training of the modeling. Empty/missing data can cause the model to perform poorly.
- **Feature Binning:**
  In this step similarly used the withColumn() along with when to categorize the data.
  - Categorized the price column into:
    - Low: Listings with price ≤ 100.
    - Medium: Listings with $100 <$ price ≤ 300.
    - High: Listings with price > 300.
  - Categorized the reviews_per_month column into:
    - Rarely Reviewed: reviews_per_month ≤ 1.
    - Moderately Reviewed: $1 <$ reviews_per_month ≤ 3.
    - Frequently Reviewed: reviews_per_month > 3.

  This pre-processiong helps with visualizing and categorizing the airbnb's based on price. The three categories give a more simple view of the overall dataset. Travelers/Users can narrow down on specific category as required and accordingly get further data to make their informed decisions. Similarly the the number if reviews categorized will help grouping the listings that people are interested in based on how many times they are reviewed. Overall this cleaning process in further help in visualizing and model creation.
- **Removing Inconsistent Data:**
  Used col() within filter() to remove the invalid data.
  - Removed rows with invalid or nonsensical values:
    - Listings with price ≤ 0 or minimum_nights ≤ 0.
    - Listings with zero availability (availability_365 = 0) but reviews_per_month > 0.

  This helps with keep only the data that can benefit the users when viewed or when creating models to train the system.
- **Text Pre-processing:**
  Used the withColumn():
  - Removed unwanted characters and punctuations (e.g., !, *,) from the name column with the regexp_replace() function.
  - Stripped leading and trailing spaces from string columns using trim().
  - Standardized string columns (name, host_name, neighbourhood_group, etc.) to title case using the initcap() function.

  The step helps clean the name column so that during modeling its cleaner to represent the data. Also display more clear details of the place. Along with it made all the text consistent so that when modeling it does not repeat similar data and affect model performance due to name inconsistency.

- **Data Aggregation:**
  Used groupBy() for data aggregation.
  - Aggregated data by host_id to calculate:
    * Average price (avg_price) using avg() and alias().
    * Total number of reviews (total_reviews) using sum() and alias().
  - Joined aggregated data back to the main dataset using join().

- **Ranking by Neighborhood:**
  To visualize the data in the tabular form in a more categorized manner using the different neighbourhoods and the different price ranges.
  - Ranked listings within each neighborhood by price using the PySpark Window function using partitionBy(). Then using the orderBy() to order based on price and last creating the column price_rank using withColumn()

  This gives the relative price position in a neighbourhood. And the ranking can help view if a certain area is more expensive than another. And then using the ranks it can be used to recommend areas to users/travelers.

- **Feature Encoding:**
  Created feature encoding to be used for modeling. utilized the StringIndexer() function, along with the fit() and transform() functions to apply it to the whole dataset for the specific column.
  - room_type encoding to room_type_index
  - neighbourhood_group encoding to neighbourhood_group_index

  This pre-processing step helps with using the specific features when creating models, since models like classification and clustering handle numerical data better and cannot handle strings directly.

- **Neighborhood Filtering:**
  Using the filter() and isin() methods to remove the samples that did not fit into the main neighbourhood groups. Since those had only one sample so it could have been an error when entering that data. To keep the data consistent and not have those individual outliers in the neighbourhood column.
  - Kept only rows where the neighbourhood_group was one of the five valid groups using the isin(): Manhattan, Brooklyn, Queens, Staten Island, Bronx.
  - Verified the filtered groups using a groupBy() count.

  This step keeps the data clean for analysis so that when modeling the individual samples that did not fit in the group will not affect the performance.

- **Handling Outliers:**
  This method helps handle the extreme values and making the model more reliable and keeping the analysis clear. Also having outliers can skew the data metrics and can lead to wrong scenarios or unwanted noise that can lead to wrong analysis.
  - Utilized the aproxQuantile() method to set the quantiles
  - then using filter() applied the change to the whole price column.

  Also the step helps with enhancing the model performance, since outliers can wrongly affect the model weights or the decision boundaries if not removed/handled. Then it help further with overfititng to extreme cases and not learn to generalize to most of the data. The price feature is also a very important feature of our dataset, which helps the users/travelers base their decisions for the room type, neighbourhood selection and even the hosts that have listings put up.

```
Decision Tree Classifier Metrics:
Accuracy: 0.64
Precision: 0.62
Recall: 0.64
F1 Score: 0.63
Execution Time: 7.58 seconds
+------------------+-----+----------+
|assembled_features|label|prediction|
+------------------+-----+----------+
|   [1.0,1.0,141.0]|  0.0|       0.0|
|   [0.0,1.0,203.0]|  0.0|       0.0|
|    [0.0,1.0,89.0]|  0.0|       0.0|
|   [0.0,3.0,211.0]|  0.0|       1.0|
|   [0.0,1.0,257.0]|  0.0|       0.0|
|   [1.0,1.0,343.0]|  1.0|       0.0|
|   [2.0,5.0,262.0]|  0.0|       1.0|
|   [2.0,6.0,361.0]|  1.0|       1.0|
|    [0.0,1.0,69.0]|  0.0|       0.0|
|     [0.0,1.0,3.0]|  0.0|       0.0|
|   [0.0,2.0,255.0]|  1.0|       1.0|
|   [0.0,3.0,347.0]|  0.0|       1.0|
|   [0.0,3.0,358.0]|  0.0|       1.0|
|   [2.0,4.0,346.0]|  1.0|       1.0|
|    [4.0,1.0,40.0]|  0.0|       0.0|
|   [2.0,1.0,335.0]|  0.0|       0.0|
|   [0.0,1.0,187.0]|  1.0|       0.0|
|   [1.0,4.0,170.0]|  1.0|       1.0|
|   [0.0,3.0,132.0]|  1.0|       1.0|
|   [2.0,2.0,174.0]|  0.0|       1.0|
+------------------+-----+----------+
```

Figure 1. decision tree

PART 2: MACHINE LEARNING ALGORITHMS

The following machine learning algorithms were implemented as part of the Airbnb dataset analysis:

1. Decision Tree Classifier

1) **Data Preparation:**
   - Features selected neighbourhood_group, calculated_host_listings_count, and availability_365 as features.
   - Utilized the pre-processed `room_type_index` as the target label.

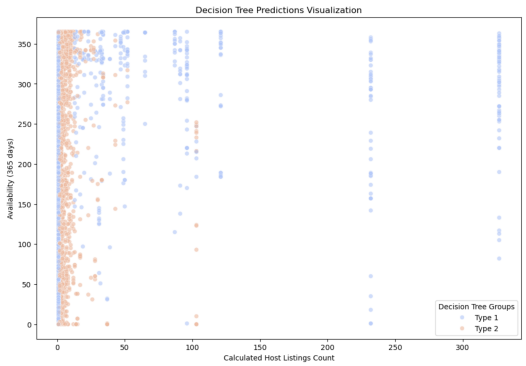


Figure 2. decision tree visual

2) **Model Training:**
   - Trained a decision tree classifier on the selected features using default parameters.
   - The features were assembled into a single vector using `VectorAssembler`.
3) **Evaluation:**
   - The dataset was split into 80% training and 20% testing sets.
   - Achieved an accuracy of:

Decision Tree Accuracy: 64%

   - Evaluated model performance using metrics such as accuracy and confusion matrix.

COMPARISONS

Accuracy::

- **Non-Distributed:** 0.81
- **Distributed:** 0.64

The distributed model had a noticeable drop in accuracy. This could be due to differences in how splits are calculated across partitions, leading to suboptimal tree construction.
Precision and Recall:: The non-distributed model had strong precision and recall (weighted avg precision: 0.81, recall: 0.81). The distributed model performed worse, which indicates potential issues with distributed split computations or parameter tuning.
Execution Time:: Distributed decision trees benefit from Spark's parallel split evaluation, reducing computational time for large datasets. However, for smaller datasets, the non-distributed version may be faster due to the absence of overhead.
DAG Insights:: The DAG for distributed decision tree training showed stages for evaluating splits, pruning nodes, and constructing the tree hierarchy. Each stage involved shuffling data between nodes to compute split criteria.

2. Logistic Regression

1) **Data Preparation:**

```
Logistic Regression Accuracy: 0.81
F1-Score: 0.80
Precision: 0.85
Recall: 0.75
Execution Time: 13.49 seconds
Classification Report:
```

	precision	recall	f1-score	support
Not Entire Home/Apt	0.78	0.87	0.82	3334
Entire Home/Apt	0.85	0.75	0.80	3302
accuracy			0.81	6636
macro avg	0.81	0.81	0.81	6636
weighted avg	0.81	0.81	0.81	6636

Figure 3. Logistic Regression Metrics

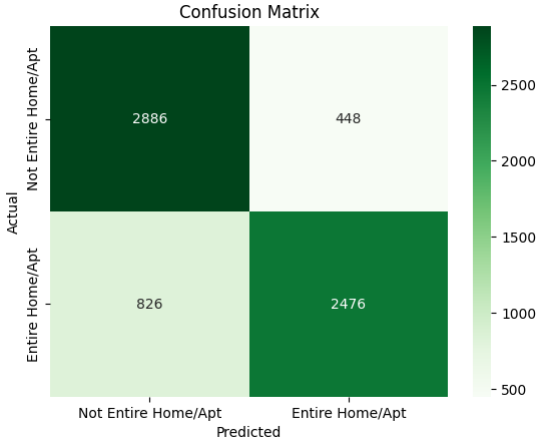


Figure 4. Logistic Regression Matrix

   - Created the column room_binary using the room_type_index created during pre-processing, to categorize the data as either Entire Apt or Not Entire Apartment as logistic regression functions better for binary classification.
   - Used neighbourhood_group and price features.
   - Split the data into 80% and 20% for training and testing sets.
2) **Model Training:**
   - Trained a logistic regression model with:
     - Regularization parameter (`regParam`): 0.01
     - Elastic net parameter (`elasticNetParam`): 0.0
   - The features were assembled into a single vector using `VectorAssembler`.
3) **Evaluation:**
   - Calcaulted the accuracy using the MulticlassClassificationEvaluator() method
   - Achieved an accuracy of:

Logistic Regression Accuracy: 81%

   - Evaluated model performance using the below metrics and plotted a confusion matrix as shown in Fig. 3 and Fig. 4

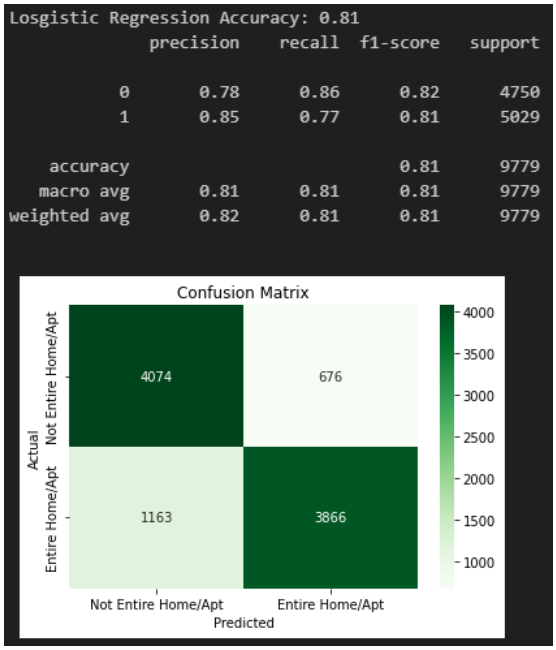


Figure 5. Logistic Regression Metrics using Python

COMPARISONS

To dive into some of the differences between the two ways of processing the dataset and using machine learning models to understand the data:

1) Execution:
   - Fig. 3 and Fig. 4 are performance collected using PySpark, where the computation is handled parallel which makes it efficient for our dataset.
   - Fig. 5 is performance of the same model using Python libraries. This approach is usually slower and the execution time can be higher.
2) Performance:
   - Accuracy: For both the models trained using PySpark and regular python libraries we can see the accuracy still remains at 81% showing that Logistic Regression is good model for this dataset and to understand and make informed decision about the type of room to select.
   - Precision, Recall, F1-Score: Here we can see some difference in the recall values which shows 75% for `Entire Home/Apt` using PySpark and 77% using Python libraries, but on the other hand precision is at 85% which is consistent in both approaches. Similarly we can see that variation for `Not Entire Home/Apt`. These slight variations can be due to the way the computations are done in each framework. Also this can be due to the way PySpark handles its data processing that might affect how it treats the specific category of `Entire Home/Apt` as that is the one with slight variations between both performances.

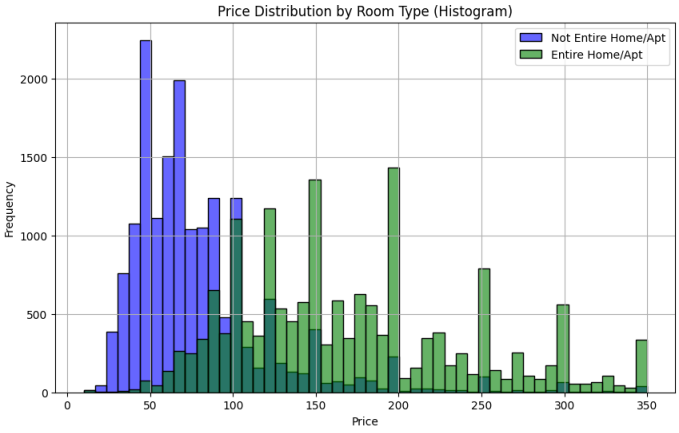


Figure 6. Price Distribution Based on Room Type

3) Training Time: Here we see that PySpark does not seem to be as beneficial with training time compared to using Python libraries. The training time using Python Sckit-learn library is 0.31 seconds whereas for PySpark it was 13.49 seconds. The sckit-learn is set up to operate on a single machine and is highly optimized for that machine when the data fits the memory. Whereas with Pyspark, since it is distributed computing it means that the dataset us split across multiple nodes and in this case the overhead computation is more to distribute the data. hence for this particular model the training time is better while using Sckit-learn.
4) Scalability Parallelization: PySpark is highly scalable and can handle big datasets, provide distributed computation. However can be sometimes hard to setup. On the other hand using libraries like Sckit-learn can work well and easy to setup, but might not be as beneficial when handling huge datasets and performance might be affected. It also might cause memory or computation limitations.

Overall comparison we can say that PySpark is beneficial when we want optimization of scalability and handle distributed computing. Python libraries even though easier to setup up and work with it, at the end it depends on the task at hand and the complexity of data being processed.

Finally to view the data in more visualize(Fig. 6) manner to see and understand what the model has trained on, is a graph showing the "Price Distribution by room type". The graph shows the trend of the two categories: `Entire Home/Apt` and `Not Entire Home/Apt`. The main decisions or observations we can get from this graph is the pricing based on the room and what to choose based on the cost. `Not Entire Home/Apt` can be seen as more budget friendly options for those who have cost restrictions. And `Entire Home/Apt` can be useful to those who prefer space or higher cost based on other factors such as the neighbourhood and host listings.

3. K-Means Clustering

In this PySpark implementation, we perform K-Means clustering to categorize Airbnb listings based on their average price. The goal is to assign each listing into one of three price categories: **Affordable**, **Mid-Range**, and **Luxurious**.

Data Preparation

Feature Engineering

The first step is to extract the relevant feature, `price`, from the dataset `df` for clustering. A `VectorAssembler` is used to assemble this feature into a feature vector required for the K-Means algorithm.

Scaling the Data

Normalization: Since K-Means relies on calculating distances between data points, it is important to normalize the data to avoid features with larger numerical ranges (like price) dominating the clustering process. Here, a `MinMaxScaler` is applied to scale the `price_vector` column.

K-Means Clustering

Model Training

The K-Means algorithm is applied to the scaled price data to form 3 clusters ($k = 3$). The model is initialized with a random seed (`seed=42`) for reproducibility.

Cluster Prediction

Assigning Clusters: After training the model, we use it to predict the cluster assignment for each data point (listing). The resulting cluster assignments are added as a new column (`prediction`) in the DataFrame.

Labeling Clusters

Human-readable Labels: The model produces cluster numbers, which are not very informative. We map these cluster numbers to human-readable labels: `'Affordable'`, `'Mid-Range'`, and `'Luxurious'`, based on the cluster center values.

Silhouette Score Evaluation

Cluster Quality: To evaluate the clustering performance, we use the Silhouette Score, which measures how similar each point is to its own cluster compared to other clusters. A higher silhouette score indicates better-defined clusters.

Comparison Table

Metric	**Python K-Means Implementation**	**PySpark K-Means Implementation**
Execution Time	34.01 seconds	1.011 seconds
Silhouette Score	0.3425	0.7715
Clustering Quality	Low	High
Parallelization	Single Node	Distributed
Cluster Assignments	Manually performed	Automatically mapped to meaningful labels

Table I
Comparison of Python and PySpark K-Means Implementations

Figure 7. Scatter Plot

```
+-------+--------------------+-----+----------+--------------+
|     id|                name|price|prediction|price_category|
+-------+--------------------+-----+----------+--------------+
|  66275|Lower East Side M...| 95.0|         2|    Affordable|
| 944755|Studio Apt 1. In ...|100.0|         2|    Affordable|
| 997404|Charming Apt. In ...|145.0|         0|     Mid-Range|
|1220548|Cozy 2 Br Apartme...| 90.0|         2|    Affordable|
|1656254|Habitat 101. Amaz...|149.0|         0|     Mid-Range|
|1677887|Charming Lg Bedro...| 80.0|         2|    Affordable|
|1945115|Beautiful Luxury ...| 90.0|         2|    Affordable|
|1981410|Spacious 1 Bed Ne...|135.0|         0|     Mid-Range|
|3268008|West Village Desi...|200.0|         0|     Mid-Range|
|3674757|Cozy Vintage Arti...| 77.0|         2|    Affordable|
|3704067|Garden Apartment ...|175.0|         0|     Mid-Range|
|4060526|      Private Room A| 39.0|         2|    Affordable|
|4586880|Vintage Room  In ...| 50.0|         2|    Affordable|
|4632852|Spacious, Rustic ...| 85.0|         2|    Affordable|
|4930683|This Cozy Space I...|100.0|         2|    Affordable|
|5410897|Fab Park Slope Ga...|169.0|         0|     Mid-Range|
|5504568|Spacious European...| 45.0|         2|    Affordable|
|5811254|A Beautiful Brown...|255.0|         1|     Luxurious|
|6107289|Beautiful Brownst...|350.0|         1|     Luxurious|
|6141561|Beautiful Room & ...|125.0|         0|     Mid-Range|
+-------+--------------------+-----+----------+--------------+
only showing top 20 rows

Execution Time: 1.01 seconds
Silhouette Score: 0.7715
```

Figure 8. Silhouette Score

Comparisons

Based on Table 1

- Execution Time: PySpark excels in execution time for large datasets due to its distributed processing capabilities. The Python implementation is much slower because it processes the data on a single machine.
- Silhouette Score: The PySpark model yields a higher silhouette score, indicating better-defined and more accurate clusters.
- Clustering Quality: PySpark's ability to perform distributed calculations ensures better clustering quality,

evident in its higher silhouette score and clearer visual clusters.

4. Random Forest Classification

Data Preparation

Feature Engineering

The first step is to index the `price_category` column, which is a string, using `StringIndexer`. This converts the price categories into numeric values for the classifier to process. Then, relevant features (`latitude`, `longitude`, `minimum_nights`, `reviews_per_month`, `host_listings_count`, and `availability`) are assembled into a feature vector using `VectorAssembler`. These features are required as input for the Random Forest model.

Splitting the Data

The dataset is split into training and test sets using `randomSplit` (80% training, 20% test), ensuring a proper evaluation mechanism for the model.

Model Training

Random Forest Classifier:
The `RandomForestClassifier` is initialized with 100 trees (`numTrees=100`) and trained on the training data (`train_df`). The `price_category_index` is used as the target variable (`labelCol`), and the assembled feature vector (`featuresCol`) is used as the input.

Model Evaluation

After training, the model predicts on the test dataset (`test_df`). The `MulticlassClassificationEvaluator` calculates the accuracy, providing insight into model performance.

Performance Metrics

- **Accuracy:** The accuracy measures the proportion of correct predictions. Accuracy = 70.16%.
- **Precision:** Proportion of true positives out of all positive predictions:
  - Affordable: 0.69
  - Mid-Range: 0.71
  - Luxurious: 0.00
- **Recall:** Proportion of true positives out of all actual positives:
  - Affordable: 0.79
  - Mid-Range: 0.63
  - Luxurious: 0.00
- **F1 Score:** Harmonic mean of precision and recall:
  - Affordable: 0.74
  - Mid-Range: 0.67
  - Luxurious: 0.00

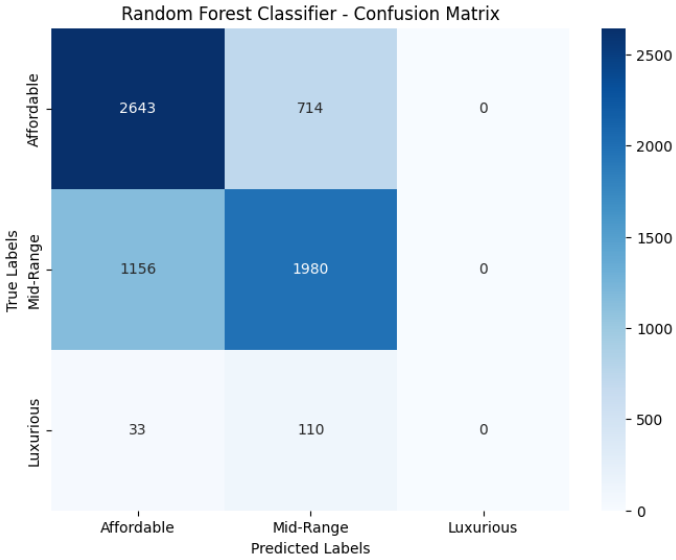


Figure 9. Confusion Matrix

Confusion Matrix

The confusion matrix visualizes the classifier's performance. It highlights areas where the model struggles, particularly with Luxurious listings, which are rare in the dataset.

Distributed Processing

- **Execution Time:** 13.11 seconds, thanks to PySpark's parallel processing across multiple nodes.
- **Scalability:** PySpark efficiently scales to large datasets, making it suitable for handling vast data.
- **Fault Tolerance:** PySpark's distributed data ensures robustness. If a node fails, the task is reassigned to another node.

Summary

PySpark proves to be an efficient and scalable tool for machine learning on large datasets. The Random Forest Classifier achieved a reasonable accuracy of 70.16%, with strong performance on Affordable and Mid-Range listings. The model struggled with Luxurious listings due to class imbalance. PySpark's distributed nature enables faster model training and evaluation.

5. Support Vector Machine (SVM)

1) **Data Preparation:**
   - Normalized the `price` column using min-max scaling:

$$\text{normalized\_price} = \frac{\text{price} - \text{min\_price}}{\text{max\_price} - \text{min\_price}}$$

   - Selected `latitude`, `longitude`, `reviews_per_month`, and `normalized_price` as features.
   - Indexed `room_type` as the target label using `StringIndexer`.

```
Execution Time: 6.13 seconds
Random Forest Classifier Accuracy: 0.6967

Classification Report:
              precision    recall  f1-score   support

  Affordable       0.69      0.79      0.74      3357
   Mid-Range       0.71      0.63      0.67      3136
   Luxurious       0.00      0.00      0.00       143

    accuracy                           0.70      6636
   macro avg       0.47      0.47      0.47      6636
weighted avg       0.68      0.70      0.69      6636

+--------+---------+--------------+----------+
|latitude|longitude|price_category|prediction|
+--------+---------+--------------+----------+
|40.74768|-73.98723|        Medium|       1.0|
|40.68409|-73.96467|           Low|       1.0|
|40.76435|-73.99054|        Medium|       1.0|
|40.83844|-73.92489|           Low|       0.0|
|40.63255|-73.97324|           Low|       0.0|
|40.76525|-73.98058|        Medium|       1.0|
|40.76148|-73.99489|          High|       1.0|
|40.71515| -73.8158|           Low|       0.0|
|40.76704|-73.98646|        Medium|       1.0|
|40.77214|-73.89169|           Low|       0.0|
|40.77309|-73.89256|           Low|       0.0|
|40.77224|-73.89295|        Medium|       0.0|
|40.76492|-73.99128|        Medium|       1.0|
|40.83464|-73.92505|           Low|       0.0|
|40.75403| -73.9715|        Medium|       1.0|
|40.75784|-73.83181|           Low|       0.0|
| 40.7552| -73.9724|        Medium|       1.0|
|40.75832|-73.99179|        Medium|       1.0|
```

Figure 10. Report

2) **Model Training:**
   - Trained a multi-class SVM using `OneVsRest` and `LinearSVC`.
   - The features were assembled into a single vector using `VectorAssembler`.
   - Used the following parameters for `LinearSVC`:
     - Maximum iterations (`maxIter`): 100.
     - Regularization parameter (`regParam`): 0.1.
3) **Evaluation:**
   - The dataset was split into 80% training and 20% testing sets.
   - Achieved an accuracy of:

SVM Accuracy: 70%

   - Evaluated model performance using metrics such as accuracy and confusion matrix.

COMPARISONS

Accuracy::

- **Non-Distributed:** 0.52
- **Distributed:** 0.70

The distributed model achieved a significant improvement in accuracy. Spark's ability to handle large datasets efficiently

```
Multi-class SVM Metrics:
Accuracy: 0.70
Precision: 0.68
Recall: 0.70
F1 Score: 0.69
Execution Time: 54.25 seconds
+--------------------+-----+----------+
|            features|label|prediction|
+--------------------+-----+----------+
|[40.69356,-73.967...|  0.0|       0.0|
|[40.76247,-73.992...|  0.0|       0.0|
|[40.72138,-74.007...|  0.0|       0.0|
|[40.76195,-73.971...|  0.0|       0.0|
|[40.75487,-73.968...|  0.0|       0.0|
|[40.63255,-73.973...|  1.0|       1.0|
|[40.72988,-73.857...|  0.0|       1.0|
|[40.70857,-73.807...|  1.0|       1.0|
|[40.71732,-74.009...|  0.0|       0.0|
|[40.7718,-73.9942...|  0.0|       0.0|
|[40.76086,-73.989...|  1.0|       0.0|
|[40.76269,-73.989...|  0.0|       0.0|
|[40.76704,-73.986...|  0.0|       0.0|
|[40.77224,-73.892...|  1.0|       1.0|
|[40.62063,-74.130...|  0.0|       1.0|
|[40.72136,-73.817...|  0.0|       1.0|
|[40.76537,-73.987...|  1.0|       0.0|
|[40.64007,-73.980...|  1.0|       1.0|
|[40.76002,-73.989...|  1.0|       0.0|
|[40.76199,-73.794...|  0.0|       1.0|
+--------------------+-----+----------+
```

Figure 11. SVM

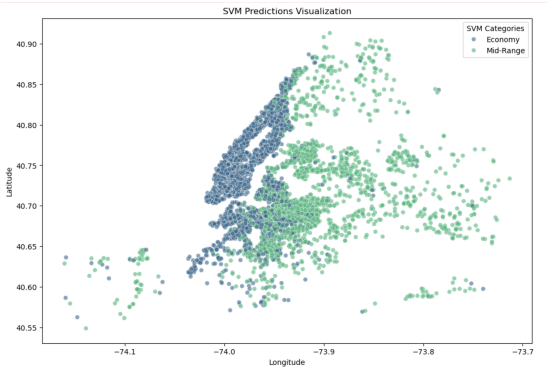


Figure 12. SVM visual

likely contributed to better convergence and margin optimization.

Precision and Recall:: The non-distributed model had low precision and recall (weighted avg precision: 0.62, recall: 0.52). The distributed version performed better, indicating that Spark effectively utilized distributed gradient descent to optimize the SVM model.

Execution Time:: SVM models typically require more computational resources due to their iterative nature. The distributed version likely required more time than the non-distributed one but scaled better for large datasets.

```
Naive Bayes Accuracy: 0.61
Precision: 0.61
Recall: 0.61
F1-Score: 0.60

Classification Report:
              precision    recall  f1-score   support

         0.0       0.63      0.70      0.66      3357
         1.0       0.61      0.53      0.56      3136
         2.0       0.08      0.08      0.08       143

    accuracy                           0.61      6636
   macro avg       0.44      0.44      0.44      6636
weighted avg       0.61      0.61      0.60      6636

Execution Time: 6.50 seconds
```

Figure 13. Naive Bayes Metrics

```
Naive Bayes Accuracy: 0.53
              precision    recall  f1-score   support

           0       0.52      0.98      0.68      4941
           1       0.77      0.07      0.13      4838

    accuracy                           0.53      9779
   macro avg       0.64      0.52      0.40      9779
weighted avg       0.64      0.53      0.41      9779

Execution Time: 0.19 seconds
```

Figure 14. Naive Bayes Python Libraries Implementation Metrics

DAG Insights:: The DAG for distributed SVM showed iterative stages for gradient updates. Each stage involved computing the margins for support vectors and updating the hyperplane parameters.

6. Naive Bayes

1) **Data Preparation:**
   - Using features calculated_host_listings_count and neighbourhood_group_index created during preprocessing. And selected the price_category_index as the target column for classification.
2) **Model Training:**
   - The dataset was split into 80% training and 20% testing sets.
   - The features were assembled into a single vector using `VectorAssembler`.
3) **Evaluation:**
   - Utilizing the MulticlassClassificationEvaluator() function achieved an accuracy of:

     Naive Bayes Accuracy: 61%

   - Evaluated model performance using the below metrics and plotted a confusion matrix as shown in Fig. 13 and Fig. 14

COMPARISONS

To now compare the performance metrics and see how they are different based on the framework used be it the PySpark approach or the Python library approach.

1) Execution
   - Fig. 13 is the metrics for the PySpark execution. This approach utilizes the distributed fashion of computation which means it split across different nodes, Includes the functions like Vector Assembler, model fitting and evaluation for multiple jobs which adds latency.
   - Fig. 14 this is metrics of python implementation of Naive Bayes. This execution utilizes a single nodes for computation and hence reduces the overhead distributed computation. However the memory constraint is a limitation in this case.
2) Performance
   - Accuracy: From both the metrics collected it very evident that there is an increase in the accuracy when using the PySpark framework. We can see that during the python library implementation the value is 53% and for PySpark we get 61%. Shows that the distributed handling of data is a better approach in this case. Now the reason that Python library implementation had lower value can be due to different points such as different pre-processing of the data or lack of proper handling of the imbalanced features. In this case its slightly different also since we only have two groups created during the Python implementation however we three categories which trained model better so it could classify data correctly.
   - Precision: In this model we see again that the value is higher for category 0 and less for 1 since we separated the data in that into two groups. And the value of group 2 is less since the data for that price group is comparatively lesser and not as many positives predictions. Also we can see that when using Python library of sckit-learn it perform a bit better for the weighted average, however this could be because it is leaning more towards the majority group i.e. 0 and avoids more false positives, but do so at the expense of other groups. But we can also say that PySpark handles/minimizes the misclassifications and thus provides balances precision across the groups.
   - Recall: For this we can see that values are different again as well have three groups in the PySpark approach and 2 in the regular Python implementation. However we can see the weighted average for PySpark is higher which shows it perform well across all the classes. whereas there is slight difference in Python implementation as it tries to capture all instances of actual positives.

- F1-Score: This gives the harmonic mean of precision and recall, hence giving a balance between the two. As we can see from the metrics that the weighted average in PySpark implementation outperforms the value in Python implementation. The values are 60% and 41% for PySpark and Python library implementations respectively.

Overall we can say that the PySaprk implementation of Naive Bayes performed better. It was able to handle the class imbalance.

3) Training Time: In this metric we can see that the training time was higher for the PySpark implementation which was 6.50 seconds. Now this is mostly due to the initial overhead to set up the distribution and manage the data partitioning. Usually this can be different for a much huger dataset if compared with the two different approaches of implementations. And the operations such as Vector Assembler and model fitting ad latency to the whole process increasing the time taken to execute.
4) Parallelization, Scalability and Efficiency
   - As noted before PySpark allows for parallel and distributed procession over multiple nodes. It process the data simultaneously. And it does not required user intervention to parallelize the tasks. However it depends on the size of the dataset. It is more ideal when you have train or work on huge dataset that perform better when distributed computations take place. It can turn out to be a challenge for smaller dataset as it would increase overhead setup computation time.
   - Python library implementation on the other hand as discussed before is a single threaded approach. which means all the computation happens on one single node. Now we can handle parallelism here too using other libraries like GridSearchCV. It also is beneficial if we are looking for lesser computation time if handling small to medium dataset.

Finally to visualize(Fig. 15) the data used in the Naive Bayes model and what it was trained on, here is a graph to show the categorization of the Price vs Listings. This graph separated the price into three categories and compares to the number of listings in each category.

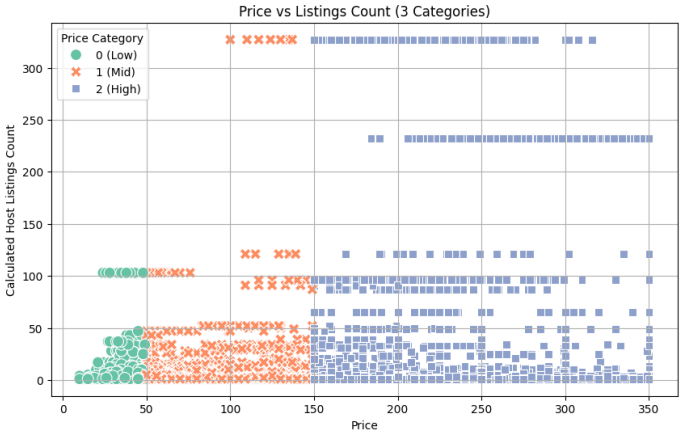
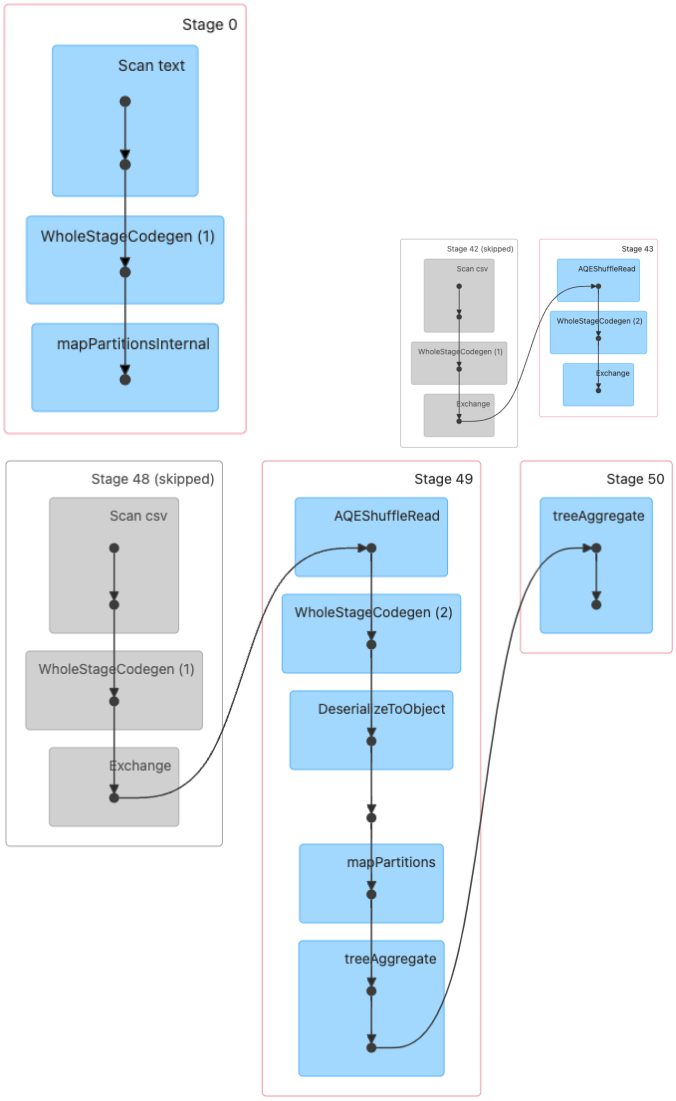
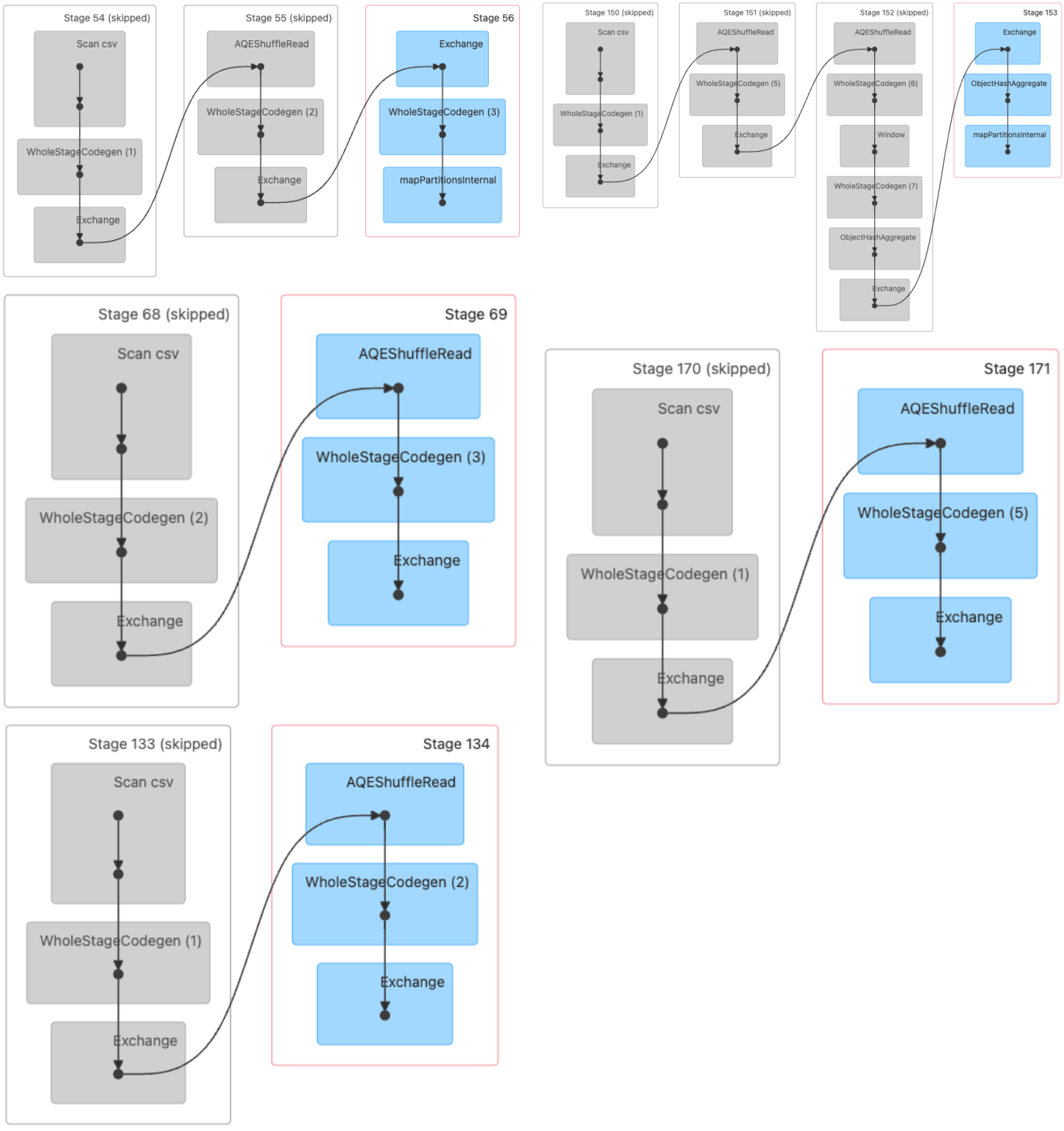


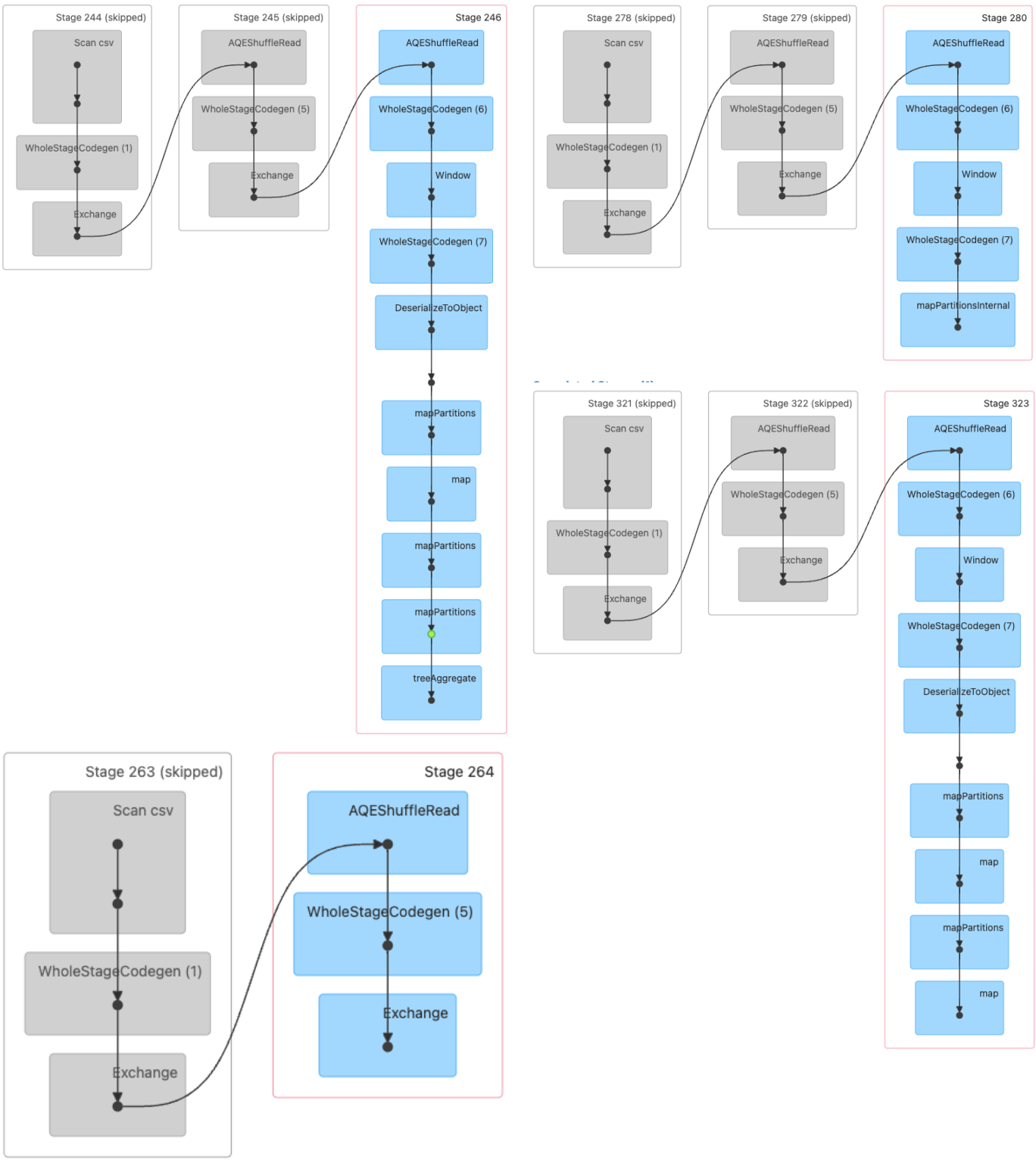
Figure 15. Price Vs Listings Count

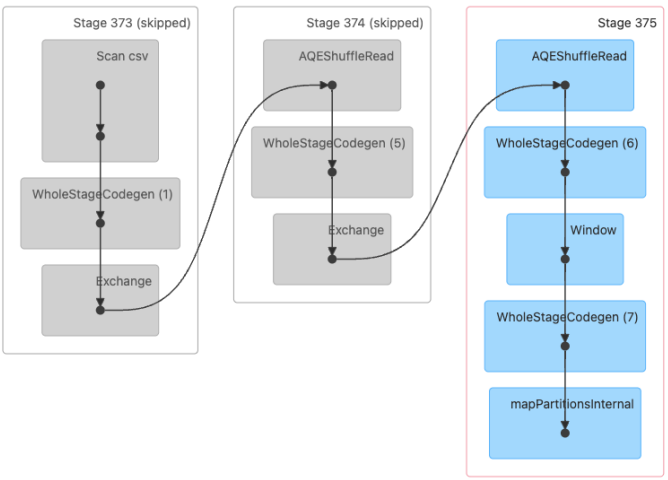
I. Explanation and Analysis

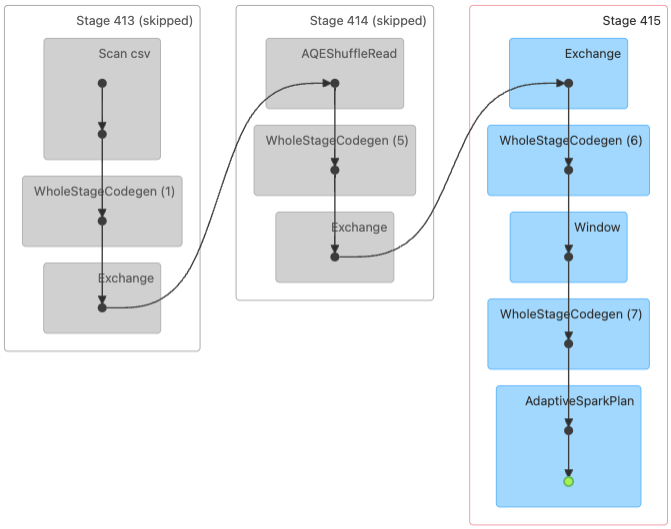
A. DAG visualization

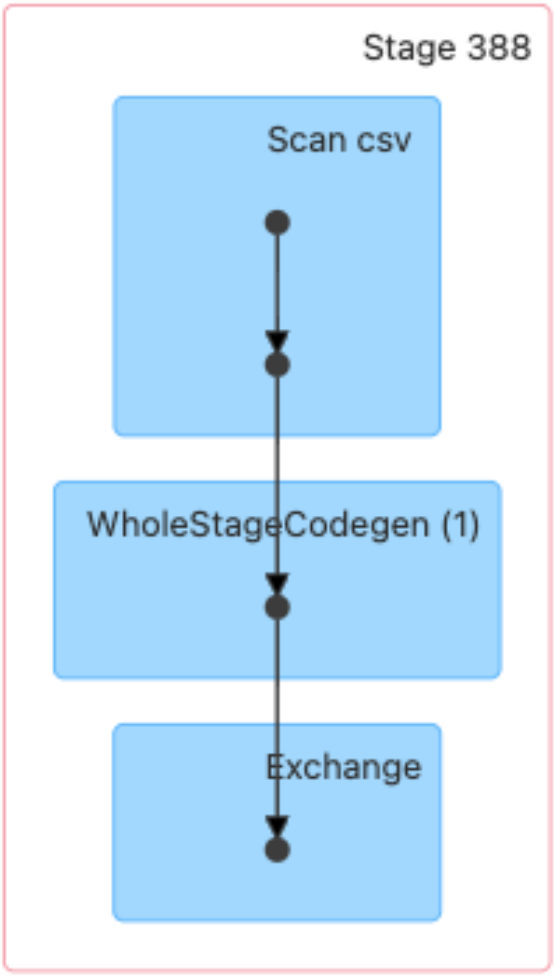


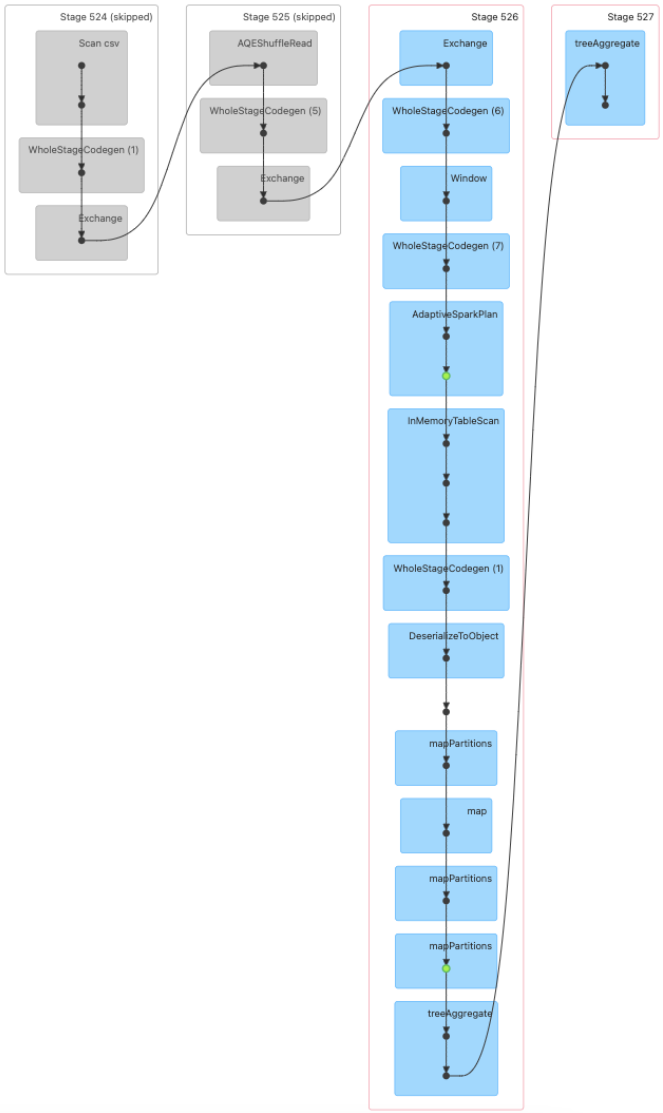

Stage 54 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 55 (skipped)
AQEShuffleRead
WholeStageCodegen (2)
Exchange
Stage 56
Exchange
WholeStageCodegen (3)
mapPartitionsInternal
Stage 150 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 151 (skipped)
AQEShuffleRead
WholeStageCodegen (5)
Exchange
Stage 152 (skipped)
AQEShuffleRead
WholeStageCodegen (6)
Window
WholeStageCodegen (7)
ObjectHashAggregate
Exchange
Stage 153
Exchange
ObjectHashAggregate
mapPartitionsInternal
Stage 68 (skipped)
Scan csv
WholeStageCodegen (2)
Exchange
Stage 69
AQEShuffleRead
WholeStageCodegen (3)
Exchange
Stage 170 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 171
AQEShuffleRead
WholeStageCodegen (5)
Exchange
Stage 133 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 134
AQEShuffleRead
WholeStageCodegen (2)
Exchange

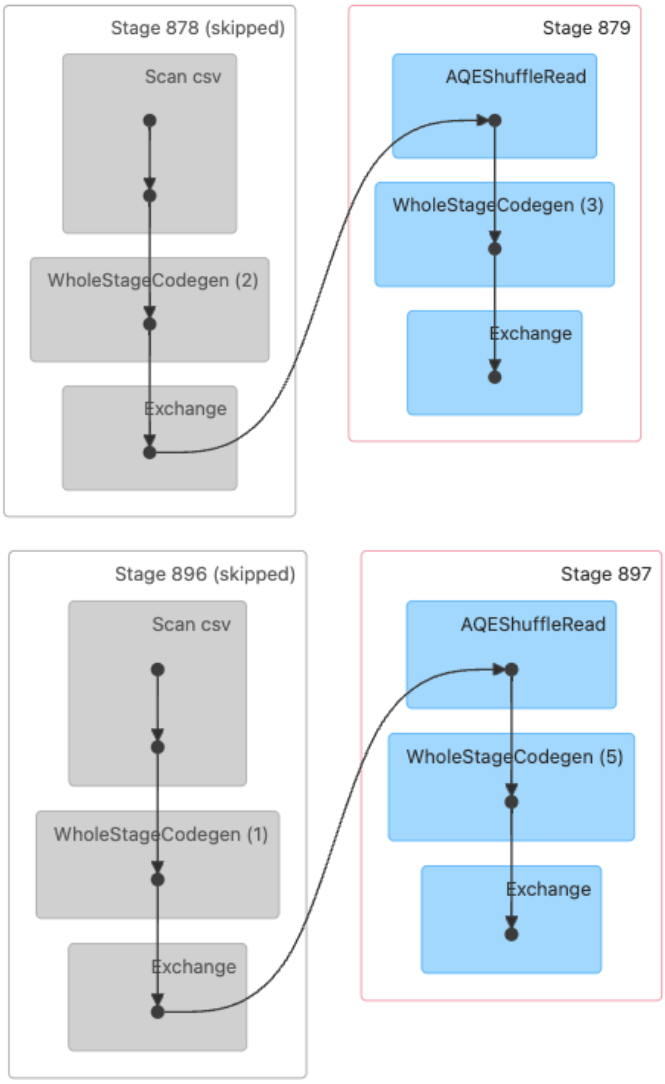

Stage 244 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 245 (skipped)
AQEShuffleRead
WholeStageCodegen (5)
Exchange
Stage 246
AQEShuffleRead
WholeStageCodegen (6)
Window
WholeStageCodegen (7)
DeserializeToObject
mapPartitions
map
mapPartitions
mapPartitions
treeAggregate
Stage 278 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 279 (skipped)
AQEShuffleRead
WholeStageCodegen (5)
Exchange
Stage 280
AQEShuffleRead
WholeStageCodegen (6)
Window
WholeStageCodegen (7)
mapPartitionsInternal
Stage 321 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 322 (skipped)
AQEShuffleRead
WholeStageCodegen (5)
Exchange
Stage 323
AQEShuffleRead
WholeStageCodegen (6)
Window
WholeStageCodegen (7)
DeserializeToObject
mapPartitions
map
mapPartitions
map
Stage 263 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 264
AQEShuffleRead
WholeStageCodegen (5)
Exchange


Stage 373 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 374 (skipped)
AQEShuffleRead
WholeStageCodegen (5)
Exchange
Stage 375
AQEShuffleRead
WholeStageCodegen (6)
Window
WholeStageCodegen (7)
mapPartitionsInternal


Stage 413 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 414 (skipped)
AQEShuffleRead
WholeStageCodegen (5)
Exchange
Stage 415
Exchange
WholeStageCodegen (6)
Window
WholeStageCodegen (7)
AdaptiveSparkPlan


Stage 388
Scan csv
WholeStageCodegen (1)
Exchange


Stage 524 (skipped)
Scan csv
WholeStageCodegen (1)
Exchange
Stage 525 (skipped)
AQEShuffleRead
WholeStageCodegen (5)
Exchange
Stage 526
Exchange
WholeStageCodegen (6)
Window
WholeStageCodegen (7)
AdaptiveSparkPlan
InMemoryTableScan
WholeStageCodegen (1)
DeserializeToObject
mapPartitions
map
mapPartitions
mapPartitions
treeAggregate
Stage 527
treeAggregate



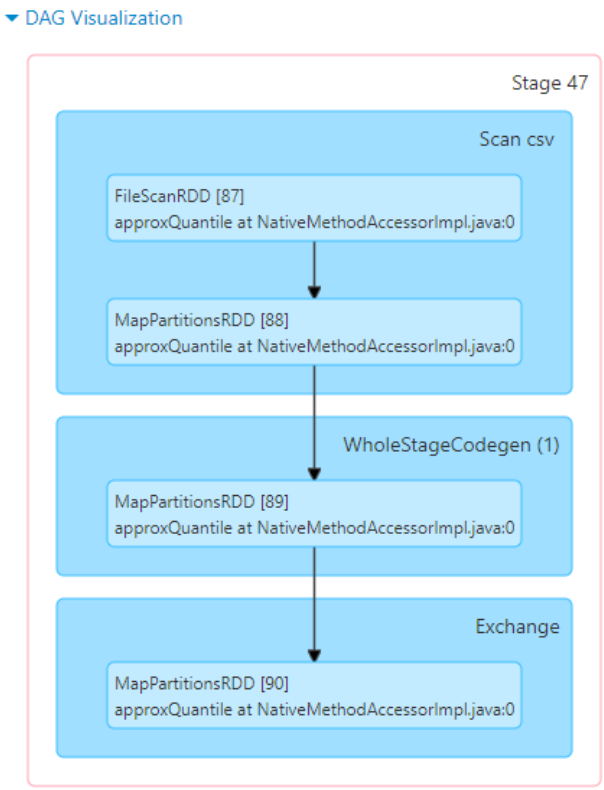


Figure 16. Action: approxQauntile method

Distributed Data Preprocessing (DAG Analysis)

In Spark, each preprocessing step is represented as a DAG, showing the transformations applied to the data and how Spark stages the execution. Below is the explanation of the preprocessing pipeline based on DAG visualization:

1. Reading Data

- **DAG Structure:** The initial stages of the DAG involve reading the CSV files. These stages distribute the file data across partitions to prepare for parallel processing.
- **Parallel Execution:** Each partition is processed independently, ensuring scalability for large datasets.
- **Node Representation:** In the DAG, this appears as the first node, labeled as the file source, followed by transformations such as `mapPartitions()`.

2. Transformations

- **Filtering Rows:** The DAG shows transformations where invalid rows (e.g., missing values) are filtered out. These transformations are represented as narrow dependencies since data within a partition is processed independently.
- **Type Casting:** Another transformation involves converting columns to appropriate data types (e.g., strings to integers or floats). This appears as a `map()` stage in the DAG.
- **Stages:** The DAG stages are optimized, ensuring that multiple transformations are combined into a single stage wherever possible.

3. Actions (Count Operations)

- **Purpose:** The `count()` operations trigger execution and allow validation of intermediate results. Each action starts a new stage in the DAG.
- **Representation:** In the DAG, these appear as nodes that depend on all previous transformations. Spark collects the results from all partitions to compute the final count.

4. Actions (approxQuantiles Operation)

- **Purpose**: The approxQuantiles() method as seen in Fig. 16is used during the pre-processing step of handling outliers. The method is used to create the quantiles so as to handle the outliers so it does not skew data and affect the model performance. We can see how the first step is to read the CSV file then apply the transformations. The method basically creates an RDD to represent the

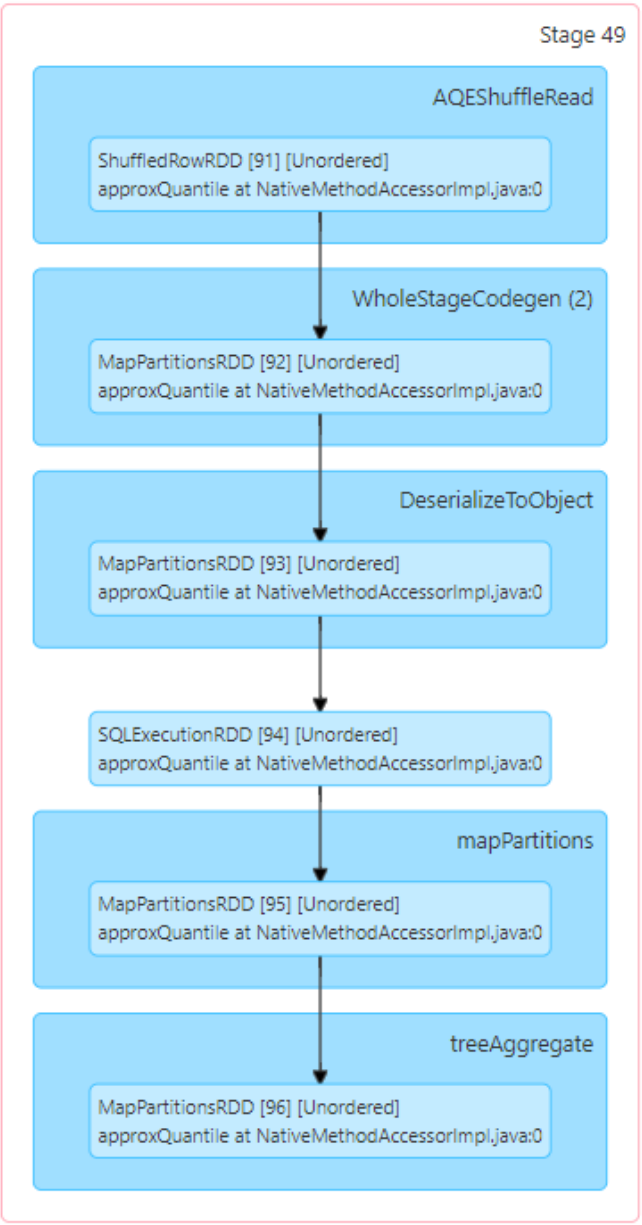


Figure 17. Next Stage of approxQuantile

data read from the file. After reading the mapPartitions take place independent. During the mapPartitioning process, the WholeStageCodegen takes place which is an optimization mechanism of Spark.

- The step moves to the next processing step as seen in Fig. 17. It performs the next steps of AQEShuffle that performs shuffling on the data. Then another step of WholeStageCodegen to optimize the execution, followed by a new step of DeserializeToObject that de-serializes the data to in-memory object representation. Following that it has the mapPartitions and a final step of treeAggregate to combine and summarize the data.

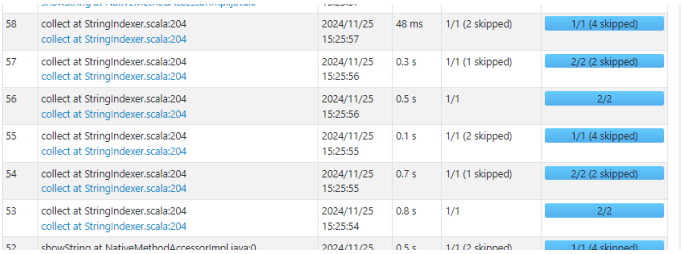


Figure 18. Collect Jobs

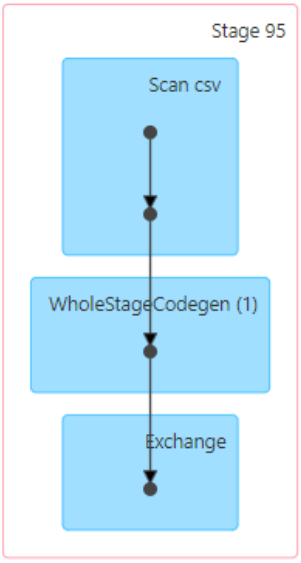


Figure 19. Collect Begin Stage

5. Actions (collect Operations)

- **Purpose**: Here the collect() operation takes place when we utilize the StringIndexer to create the encoding in the pre-processing step of room_type_index and neighbourhood_index as shown in Fig. 18
- In Fig. 19 we can see the steps or readign the CSV, followed by wholeStageCodegen for optimizing and then the exchange with the next job.
- Fig. 20 shows the final step of collect operation where we can see the exchange follows the ObjectHashAggregate step and then the mapPartitions processing.

Advantages of Spark for Preprocessing (DAG Perspective)

- The DAG ensures fault tolerance by tracking the lineage of transformations, allowing recomputation in case of failures.
- The DAG scheduler optimizes transformations into pipelines, reducing execution time by minimizing shuffles and redundant computations.
- Distributed execution of preprocessing steps increases scalability and efficiency.

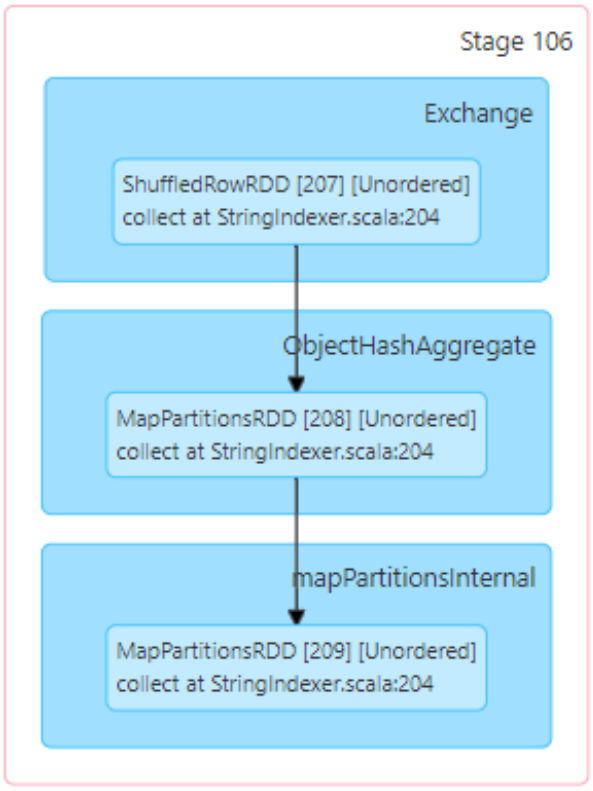


Figure 20. Collect End Stage

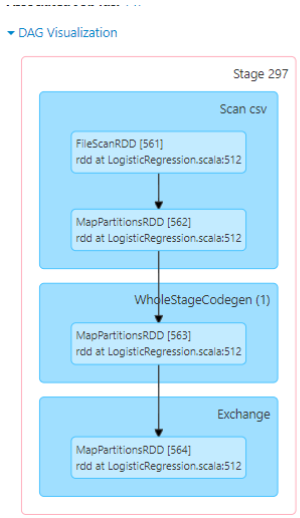


Figure 21. Logistic Regression Begin Stage

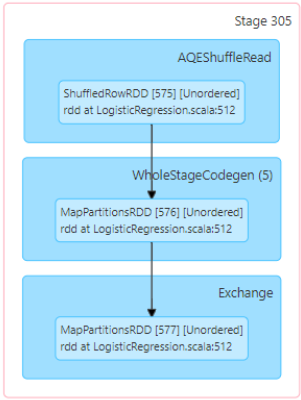


Figure 22. Logistic Regression Last Stage

Machine Learning Models (DAG Analysis)

For machine learning models, Spark DAG's represent the distributed nature of training. Each stage in the DAG corresponds to a specific step in the algorithm.

1. Decision Tree

- **DAG Structure:** Decision tree training involves hierarchical computations to find the best split for each node. Each split is a separate stage in the DAG.
- **Stages:**
  - Evaluating all possible splits for each feature is performed in parallel across partitions.
  - Pruning nodes based on stopping criteria, such as maximum depth or minimum node size.
- **Optimization:** The DAG scheduler combines computations for multiple splits into a single stage wherever possible, reducing shuffle operations.

2. Logistic Regression

- **DAG Structure:** Logistic regression involves iterative computation using gradient descent. Each iteration is represented as a separate stage in the DAG. Each stage has the same steps of reading the CSV, optimizing through WholeStageCodegen and then the exchange.
- **Stages:**

164	rdd at ClassificationSummary.scala:58 rdd at ClassificationSummary.scala:58	2024/11/25 15:27:02	0.2 s	1/1 (1 skipped)	2/2 (2 skipped)
163	rdd at ClassificationSummary.scala:58 rdd at ClassificationSummary.scala:58	2024/11/25 15:27:01	1 s	1/1	2/2
162	rdd at ClassificationSummary.scala:58 rdd at ClassificationSummary.scala:58	2024/11/25 15:27:00	0.6 s	1/1	2/2
161	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:27:00	61 ms	1/1 (2 skipped)	2/2 (4 skipped)
160	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:27:00	0.1 s	1/1 (2 skipped)	2/2 (4 skipped)
159	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:27:00	60 ms	1/1 (2 skipped)	2/2 (4 skipped)
158	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:27:00	67 ms	1/1 (2 skipped)	2/2 (4 skipped)
157	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:26:59	89 ms	1/1 (2 skipped)	2/2 (4 skipped)
156	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:26:59	59 ms	1/1 (2 skipped)	2/2 (4 skipped)
155	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:26:59	71 ms	1/1 (2 skipped)	2/2 (4 skipped)
154	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:26:59	0.1 s	1/1 (2 skipped)	2/2 (4 skipped)
153	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2024/11/25 15:26:58	1 s	1/1 (2 skipped)	2/2 (4 skipped)
152	treeAggregate at Summarizer.scala:233 treeAggregate at Summarizer.scala:233	2024/11/25 15:26:54	1 s	1/1 (2 skipped)	2/2 (4 skipped)
151	rdd at LogisticRegression.scala:512 rdd at LogisticRegression.scala:512	2024/11/25 15:26:53	0.5 s	1/1 (1 skipped)	2/2 (2 skipped)

Figure 23. Next Stages after Logistic Regression

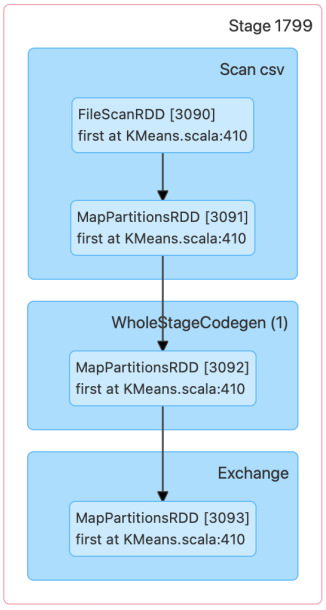


Figure 24. Clustering First Stage

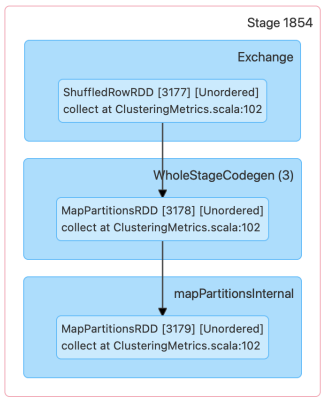


Figure 25. Clustering Last Stages

  - Computing partial gradients for each partition is performed in parallel.
  - Aggregating the gradients at the driver node to update model parameters.
  - Last stage of the logistic regression also include the AQEShuffleRead and follwoed by optimizing and exchange.

  Fig. 21 and Fig. 22 display the begin and last step of the use of Logistic Regression operation. That is then followed by the treeAggregate step and Classification-Summary operations as seen in Fig. 23
- **Iteration:** The DAG shows repeated stages for each iteration until the model converges.
- **Efficiency:** Spark optimizes the process by caching intermediate results, reducing redundant computations in each iteration.

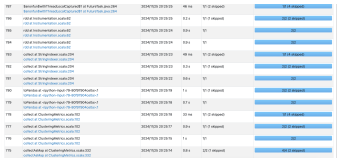


Figure 26. Clustering Stages

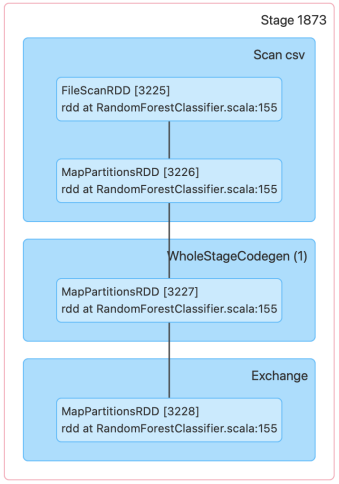


Figure 27. Random Forest Classification Start

3. K-Means

- **DAG Structure:** K-Means clustering involves iterative computations to update centroids and assign data points to clusters. Each iteration is represented as a separate stage in the DAG.
- **Stages:**
  - **Initialization:** Centroids are initialized (either randomly or using a predefined method).
  - **Distance Calculation:** Distances between each data point and the centroids are computed in parallel across partitions.
  - **Cluster Assignment:** Each data point is assigned to the nearest cluster based on the computed distances.
  - **Centroid Update:** New centroids are calculated as the mean of data points in each cluster.
- **Optimization:** The DAG scheduler combines computations for distance calculation and cluster assignment within a single stage where possible, reducing shuffle operations.

4. Random Forest Classification

- **DAG Structure:** Random Forest training involves parallel computations to build decision trees independently. Each tree is represented as a separate stage in the DAG, and within each tree, node splits are computed hierarchically.
- **Stages:**
  - **Feature Sampling:** A subset of features is selected for each tree in parallel.

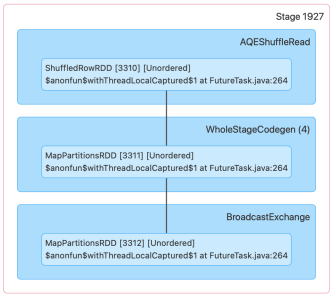


Figure 28. Random Forest Classification End

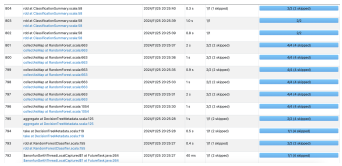


Figure 29. Random Forest Classification Middle Stages

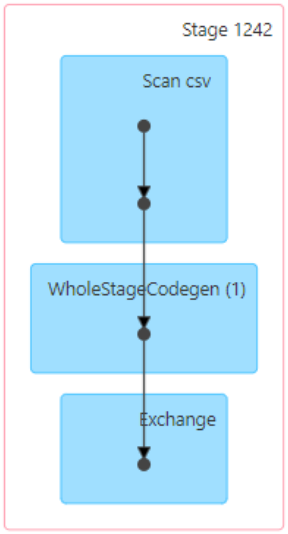


Figure 30. Naive Bayes Begin Stage

  - **Tree Training:**
    - **Split Evaluation:** Each split's potential improvement in classification is calculated across partitions in parallel.
    - **Node Splitting:** Data is split based on the best feature and threshold, creating child nodes.
    - **Pruning:** Nodes are pruned based on stopping criteria, such as maximum depth or minimum impurity.
  - **Tree Aggregation:** Predictions from all trees are aggregated to form the final output.
- **Optimization:** The DAG scheduler optimizes tasks by grouping computations for node splits and pruning to minimize data shuffles, ensuring efficient distributed execution.

5. Support Vector Machine (SVM)

- **DAG Structure:** SVM training uses gradient descent to maximize the margin between classes. The DAG shows repeated stages for margin computation and model updates.
- **Stages:**
  - Computing the margin for support vectors across all partitions.
  - Aggregating the results to update the parameters.
- **Iteration:** The DAG has multiple iterations for gradient descent, similar to logistic regression.

6. Naive Bayes

- **DAG Structure:** The model execution involves transformations like VectorAssembler, randomSplit and model.transform() which are added to the DAG but not execute immediately as they are part of the lazy execution. The training involves aggregating counts of features and classes in a distributed manner. The predictions are distributed and probabilities are computed in parallel. It also utilizes the optimization operations of shuffling. Lastly the actions such as fit(), show() and evaluate() are what trigger the execution.
- **Stages:**
  - Fig. 30 shows the same stages of reading the CSV, then optimizing using the WholeStageCodegen and then the exchange step.
  - Fig. 31 shows the last stage of Naive Bayes Operation which has the Shuffle step, the ObjectHashAggregate step to collect all the processing done into

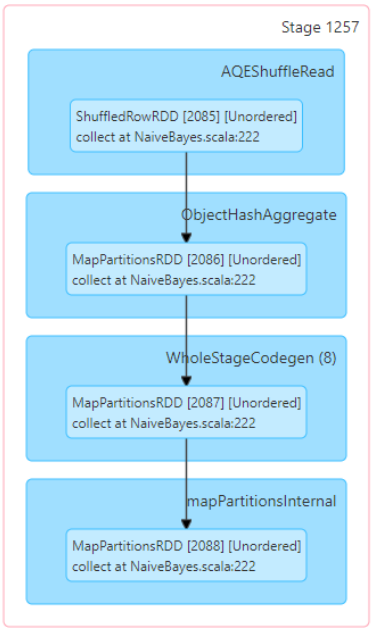
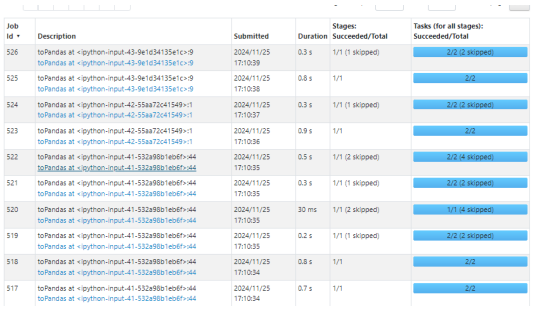


Figure 31. Naive Bayes Operation Last Stage

498	collectAsMap at MulticlassMetrics.scala:61 collectAsMap at MulticlassMetrics.scala:61	2024/11/25 17:10:25	0.5 s	2/2 (2 skipped)	4/4 (4 skipped)
497	rdd at MulticlassClassificationEvaluator.scala:191 rdd at MulticlassClassificationEvaluator.scala:191	2024/11/25 17:10:25	0.3 s	1/1 (1 skipped)	2/2 (2 skipped)
496	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264 $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	2024/11/25 17:10:25	31 ms	1/1 (2 skipped)	1/1 (4 skipped)
495	rdd at MulticlassClassificationEvaluator.scala:191 rdd at MulticlassClassificationEvaluator.scala:191	2024/11/25 17:10:24	0.1 s	1/1 (1 skipped)	2/2 (2 skipped)
494	rdd at MulticlassClassificationEvaluator.scala:191 rdd at MulticlassClassificationEvaluator.scala:191	2024/11/25 17:10:24	0.7 s	1/1	2/2
493	rdd at MulticlassClassificationEvaluator.scala:191 rdd at MulticlassClassificationEvaluator.scala:191	2024/11/25 17:10:24	0.5 s	1/1	2/2
492	collect at NaiveBayes.scala:222 collect at NaiveBayes.scala:222	2024/11/25 17:10:23	61 ms	1/1 (3 skipped)	1/1 (6 skipped)
491	collect at NaiveBayes.scala:222 collect at NaiveBayes.scala:222	2024/11/25 17:10:22	0.6 s	1/1 (2 skipped)	2/2 (4 skipped)
490	collect at NaiveBayes.scala:222 collect at NaiveBayes.scala:222	2024/11/25 17:10:22	0.3 s	1/1 (1 skipped)	2/2 (2 skipped)
489	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264 $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	2024/11/25 17:10:22	31 ms	1/1 (2 skipped)	1/1 (4 skipped)
488	collect at NaiveBayes.scala:222 collect at NaiveBayes.scala:222	2024/11/25 17:10:22	0.2 s	1/1 (1 skipped)	2/2 (2 skipped)
487	collect at NaiveBayes.scala:222 collect at NaiveBayes.scala:222	2024/11/25 17:10:21	0.8 s	1/1	2/2
486	collect at NaiveBayes.scala:222 collect at NaiveBayes.scala:222	2024/11/25 17:10:21	0.7 s	1/1	2/2

Figure 32. Stages After Naive Bayes Operation



Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
526	toPandas at <ipython-input-43-9e1d34135e1c>:9 toPandas at <ipython-input-43-9e1d34135e1c>:9	2024/11/25 17:10:39	0.3 s	1/1 (1 skipped)	2/2 (2 skipped)
525	toPandas at <ipython-input-43-9e1d34135e1c>:9 toPandas at <ipython-input-43-9e1d34135e1c>:9	2024/11/25 17:10:38	0.8 s	1/1	2/2
524	toPandas at <ipython-input-42-55aa72c41549>:1 toPandas at <ipython-input-42-55aa72c41549>:1	2024/11/25 17:10:37	0.3 s	1/1 (1 skipped)	2/2 (2 skipped)
523	toPandas at <ipython-input-42-55aa72c41549>:1 toPandas at <ipython-input-42-55aa72c41549>:1	2024/11/25 17:10:36	0.9 s	1/1	2/2
522	toPandas at <ipython-input-41-532a98b1eb6f>:44 toPandas at <ipython-input-41-532a98b1eb6f>:44	2024/11/25 17:10:35	0.5 s	1/1 (2 skipped)	2/2 (4 skipped)
521	toPandas at <ipython-input-41-532a98b1eb6f>:44 toPandas at <ipython-input-41-532a98b1eb6f>:44	2024/11/25 17:10:35	0.3 s	1/1 (1 skipped)	2/2 (2 skipped)
520	toPandas at <ipython-input-41-532a98b1eb6f>:44 toPandas at <ipython-input-41-532a98b1eb6f>:44	2024/11/25 17:10:35	30 ms	1/1 (2 skipped)	1/1 (4 skipped)
519	toPandas at <ipython-input-41-532a98b1eb6f>:44 toPandas at <ipython-input-41-532a98b1eb6f>:44	2024/11/25 17:10:35	0.2 s	1/1 (1 skipped)	2/2 (2 skipped)
518	toPandas at <ipython-input-41-532a98b1eb6f>:44 toPandas at <ipython-input-41-532a98b1eb6f>:44	2024/11/25 17:10:34	0.8 s	1/1	2/2
517	toPandas at <ipython-input-41-532a98b1eb6f>:44 toPandas at <ipython-input-41-532a98b1eb6f>:44	2024/11/25 17:10:34	0.7 s	1/1	2/2

Figure 33. Last Few Stages of DAG

one RDD and the optimize it using the WholeStage-Codegen and finish off with the mapPartitions before the next stage.

- Next steps as seen in Fig. 32 are the next steps that run the evaluate functions to calculate and see how the the data behaves and then collect the metrics accordingly.

- **Iterations:** Here also we can see that the DAG has multiple iterations based on the distributed processing it does when running the model.

Final Stages

- Fig. 32 we can see the last stages of the DAG and we notice most of them are toPandas() operations which match with what we do when we create our visualizations of the graphs. It includes converting the spark dataframe to a Python dataframe so that we can plot the graphs using Seaborn and Matplotlib libraries.
- Fig. 34 and Fig. 35 show the beigning and end stages of the toPandas() operation. Where Stage 1327 is the final stage that performs the shuffling, optimizing and mapPartition operations.

In total there are 526 jobs and 1327 Stages.

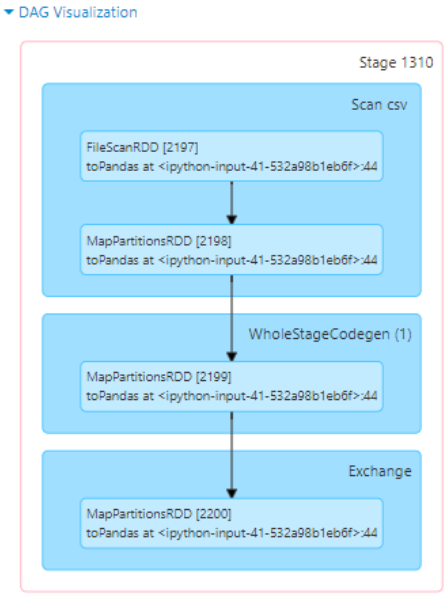


Figure 34. First Stage of toPandas() Operation

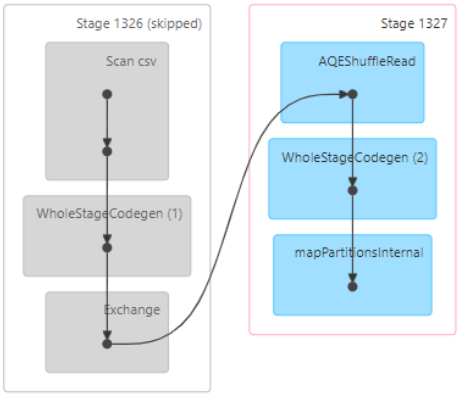


Figure 35. Final Stage of DAG

REFERENCES

[1] Dgomonov, New York City Airbnb Open Data, Kaggle. 2019, August 12. Available at: https://www.kaggle.com/datasets/dgomonov/new-york-city-airbnb-open-data

[2] XGBoost documentation. XGBoost Documentation - xgboost 2.1.1 documentation. (n.d.). Available at: https://xgboost.readthedocs.io/en/latest/

[3] Parth Shukla, Analytics Vidya. Naive Bayes Algorithms: A Complete Guide for Beginners. 2024, March 21. Available at: https://www.analyticsvidhya.com/blog/2023/01/naive-bayes-algorithms-a-complete-guide-for-beginners/#h-what-is-naive-bayes-algorithm

[4] Ajitesh Kumar, Analytics Yogi, Recommender Systems in Machine Learning: Examples. 2024, September 16. Available at: https://vitalflux.com/recommender-systems-in-machine-learning-examples/

[5] Scikit Learn. Naive Bayes. (n.d.). Available at: https://scikit-learn.org/stable/modules/naive_bayes.html

[6] Scikit Learn. KMeans. (n.d.). Available at: https://scikit-learn.org/1.5/modules/generated/sklearn.cluster.KMeans.html

[7] GeeksforGeeks. K means Clustering – Introduction. 2024, August 29. Available at: https://www.geeksforgeeks.org/k-means-clustering-introduction/

[8] Wikipedia. Decision tree learning. (n.d.). Available at: https://en.wikipedia.org/wiki/Decision_tree_learning

[9] Scikit Learn. Decision Trees. (n.d.). Available at: https://scikit-learn.org/1.5/modules/tree.html

[10] GeeksforGeeks. Decision Tree in Machine Learning. 2024, March 15. Available at: https://www.geeksforgeeks.org/decision-tree-introduction-example/

[11] Apache Spark RDD Operations - Javatpoint. Available at: www.javatpoint.com.(n.d.).https://www.javatpoint.com/apache-spark-rdd-operations

[12] RDD Programming Guide. RDD Programming Guide - Spark 3.5.3 Documentation. (n.d.). Available at: https://spark.apache.org/docs/latest/rdd-programming-guide.html

[13] Support Vector Machines. scikit. (n.d.). Available at: https://scikit-learn.org/1.5/modules/svm.html

[14] Analytics Vidhya. Data Preprocessing Using PySpark's DataFrame. 9 Jul 2024. Available at: https://www.analyticsvidhya.com/blog/2022/04/data-preprocessing-using-pysparks-dataframe/

[15] Medium. Natural Language Processing (NLP) with Spark (Python). 30 Jan 2024. Available at: https://medium.com/@mrunmayee.dhapre/natural-language-processing-nlp-with-spark-python-f67ac513616f

[16] Medium. Multi-class Text Classification using Spark ML in Python. 25 Nov 2022. Available at: https://ashokpalivela.medium.com/multi-class-text-classification-using-spark-ml-in-python-b8d2a6545cb

[17] Apache Spark. PySpark Functions. (n.d.) Available at: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/functions.html

[18] Machine Learning +. PySpark Outlier Detection and Treatment – A Comprehensive Guide How to handle Outlier in PySpark. (n.d.) Available at: https://www.machinelearningplus.com/pyspark/pyspark-outlier-detection-and-treatment/

[19] Data Minnow. Mastering Data Cleaning with PySpark. (n.d.) Available at: https://dataminnow.com/blog/mastering-data-cleaning-with-pyspark

[20] Medium. Apache Spark: Data cleaning using PySpark for beginners. 14 Jun 2021. Available at: https://medium.com/bazaar-tech/apache-spark-data-cleaning-using-pyspark-for-beginners-eeeced351ebf

[21] GeeksForgeeks. Cleaning Data with PySpark Python. 5 Feb 2023. Available at: https://www.geeksforgeeks.org/cleaning-data-with-pyspark-python/

[22] Medium. Machine learning with Pyspark MLlib: Part 1 Regression. 20 Jun 2023. Available at: https://sharmashorya1996.medium.com/machine-learning-with-pyspark-mllib-part-1-regression-e7ad4d9780af

[23] Notes By Louisa. Data Cleaning with Apache Spark. (n.d.) Available at: https://louisazhou.gitbook.io/notes/spark/data-cleaning-with-apache-spark

[24] GeeksForGeeks. PySpark Window Functions. 4 Aug 2022. Available at: https://www.geeksforgeeks.org/pyspark-window-functions/

Peer Evaluation Form for Final Group Work

CSE 487/587B

- Please write the names of your group members.

Group member 1 : Siju Chacko

Group member 2 : Mythri Shivakumar

Group member 3 : Rebecca Abraham

- Rate each groupmate on a scale of 5 on the following points, with 5 being HIGHEST and 1 being LOWEST.

Evaluation Criteria	Group member 1	Group member 2	Group member 3
How effectively did your group mate work with you?	5	5	5
Contribution in writing the report	5	5	5
Demonstrates a cooperative and supportive attitude.	5	5	5
Contributes significantly to the success of the project .	5	5	5
TOTAL	20	20	20

Also please state the overall contribution of your teammate in percentage below, with total of all the three members accounting for 100% (33.33+33.33+33.33 ~ 100%) :

Group member 1 : 33.33

Group member 2 : 33.33

Group member 3 : 33.33