Term Project

CS 777 – Big Data Analysis

AWS SPOT PRICE PREDICTION

SARVESH KRISHNAN RAJENDRAN - U86908171

SHLOK MANDLOI - U45746761

Boston University Metropolitan College

Under the guidance of

Dr. Farshid Alizadeh-Shabdiz

The **AWS Spot Price Prediction System** is a cloud-based solution designed to leverage historical AWS spot pricing data for predictive analytics. AWS spot pricing refers to the dynamic pricing model used by Amazon Web Services (AWS) for its **spot instances**, which are spare computing capacity available at significantly reduced rates compared to on-demand instances. These prices fluctuate based on real-time supply and demand in the AWS ecosystem.

Spot instances allow users to access AWS compute resources at highly discounted rates, often up to 90% cheaper than standard on-demand prices. However, the trade-off is that AWS may terminate spot instances with short notice (typically two minutes) when the demand for on-demand instances increases or the spot price exceeds the user-defined maximum bid. This dynamic pricing mechanism creates a challenge for users who need to plan their workloads cost-effectively without interruptions.

This project aims to address these challenges by predicting spot instance prices using machine learning models, including advanced neural networks. Accurate spot price predictions enable AWS users to make informed decisions, such as selecting optimal bidding strategies, scheduling workloads during low-price periods, and minimizing the risk of instance termination due to price spikes. By forecasting prices, the project empowers organizations to utilize AWS resources with greater confidence and cost-efficiency.

To accomplish these objectives, the system integrates a robust preprocessing pipeline and advanced machine learning techniques. The preprocessing pipeline includes data ingestion, cleaning, and feature engineering, transforming raw historical data into a clean, structured format suitable for modeling. Distributed data processing using PySpark ensures that the system can handle large volumes of AWS spot price data efficiently, making the pipeline both scalable and reliable.

The core of the system's predictive capability lies in its machine learning models, which include regression algorithms and a distributed deep learning model powered by Elephas and Keras. Regression models such as Linear Regression, Decision Tree, and Random Forest serve as benchmarks, capturing linear and non-linear relationships within the data. The neural network, with its multi-layered architecture, leverages the power of deep learning to model complex dependencies between features and the target variable (price).

This project is designed to address both scalability and real-world applicability. By leveraging tools like Google Cloud Storage and Dataproc, the system is capable of processing large-scale datasets in a distributed environment. The use of advanced modeling techniques ensures high predictive accuracy, while the flexibility of PySpark allows seamless integration into real-time or batch processing pipelines.

The insights generated by this system extend beyond mere cost optimization. They enable businesses to plan and execute their computational tasks more strategically, reducing the risks associated with unpredictable price fluctuations. Ultimately, the project demonstrates how cutting-edge data science and cloud technologies can be combined to solve practical challenges in dynamically priced environments like AWS.

Data Sources

- **Source**: Historical AWS spot price data was collected from https://www.kaggle.com/datasets/noqcks/aws-spot-pricing-market
- **Format**: Multiple CSV files.
- **Schema**:
    - datetime: Timestamp of the recorded price.
    - instance_type: Type of AWS instance (e.g., t2.micro, m5.large).
    - os: Operating system type (e.g., Linux, Windows).
    - region: AWS region (e.g., us-east-1, us-west-2).
    - price: Spot price of the instance in USD.

Data Preprocessing

1. **Remove Duplicates**:
    - Duplicate rows, which may result from data collection or merging errors, are removed. This step reduces noise and ensures that each record represents unique information, thereby improving the reliability of downstream predictions.
2. **Handle Missing Values**:
    - Rows containing null or missing values in critical columns (e.g., price, instance_type, region) are dropped. Retaining incomplete data can lead to inaccuracies in machine learning models, so this step maintains data integrity and quality.
3. **Filter Outliers**:
    - Outlier records, such as those with zero or negative prices, are filtered out. These values are invalid in the context of AWS spot prices and could distort the model's learning process. By removing them, the dataset better reflects the true range of prices.

Feature Engineering

1. **Categorical Encoding**:
    - **StringIndexer**:
        - Converts textual categorical columns (instance_type, os, region) into numerical indices.
        - Assigns a unique integer to each category, enabling compatibility with numerical models.
    - **OneHotEncoder**:
        - Further transforms the indexed features into **sparse vectors** that represent each category as a binary value. This approach avoids assigning unintended ordinal relationships to categories, which could mislead machine learning algorithms.
2. **Column Management**:
    - To streamline the dataset, columns that are no longer necessary (datetime, original categorical columns, and their numerical indices) are dropped.

- The final dataset retains only the relevant encoded features and the target variable (price), ensuring efficient processing and storage.

```
+------+-------------------+--------+------------+-----------------+-------------+--------------+
| price|instance_type_index|os_index|region_index|instance_type_vec|       os_vec|    region_vec|
+------+-------------------+--------+------------+-----------------+-------------+--------------+
|0.1635|                7.0|     0.0|         8.0|   (67,[7],[1.0])|(2,[0],[1.0])|(27,[8],[1.0])|
|0.4016|               43.0|     2.0|         7.0|  (67,[43],[1.0])|    (2,[],[])|(27,[7],[1.0])|
|0.1236|               24.0|     1.0|         7.0|  (67,[24],[1.0])|(2,[1],[1.0])|(27,[7],[1.0])|
|0.4034|                5.0|     0.0|         7.0|   (67,[5],[1.0])|(2,[0],[1.0])|(27,[7],[1.0])|
|0.4578|               12.0|     0.0|         7.0|  (67,[12],[1.0])|(2,[0],[1.0])|(27,[7],[1.0])|
|0.2644|                7.0|     1.0|         8.0|   (67,[7],[1.0])|(2,[1],[1.0])|(27,[8],[1.0])|
|0.2498|               47.0|     1.0|         7.0|  (67,[47],[1.0])|(2,[1],[1.0])|(27,[7],[1.0])|
| 0.564|               12.0|     1.0|         8.0|  (67,[12],[1.0])|(2,[1],[1.0])|(27,[8],[1.0])|
|0.1846|                7.0|     2.0|         8.0|   (67,[7],[1.0])|    (2,[],[])|(27,[8],[1.0])|
|0.4028|                5.0|     0.0|         7.0|   (67,[5],[1.0])|(2,[0],[1.0])|(27,[7],[1.0])|
|0.5088|                5.0|     1.0|         8.0|   (67,[5],[1.0])|(2,[1],[1.0])|(27,[8],[1.0])|
|0.1129|               29.0|     0.0|         8.0|  (67,[29],[1.0])|(2,[0],[1.0])|(27,[8],[1.0])|
|0.2016|               13.0|     0.0|         7.0|  (67,[13],[1.0])|(2,[0],[1.0])|(27,[7],[1.0])|
|0.8774|               25.0|     0.0|         8.0|  (67,[25],[1.0])|(2,[0],[1.0])|(27,[8],[1.0])|
|0.0703|                8.0|     0.0|         7.0|   (67,[8],[1.0])|(2,[0],[1.0])|(27,[7],[1.0])|
|0.1648|                7.0|     0.0|         8.0|   (67,[7],[1.0])|(2,[0],[1.0])|(27,[8],[1.0])|
|  1.36|                4.0|     1.0|         8.0|   (67,[4],[1.0])|(2,[1],[1.0])|(27,[8],[1.0])|
|0.1639|                7.0|     0.0|         7.0|   (67,[7],[1.0])|(2,[0],[1.0])|(27,[7],[1.0])|
|0.2583|                7.0|     1.0|         8.0|   (67,[7],[1.0])|(2,[1],[1.0])|(27,[8],[1.0])|
| 0.451|               38.0|     2.0|         7.0|  (67,[38],[1.0])|    (2,[],[])|(27,[7],[1.0])|
+------+-------------------+--------+------------+-----------------+-------------+--------------+
only showing top 20 rows
```

Preprocessed Data

Data Export

1. **Exporting Data**:
   - The preprocessed dataset is saved in **Parquet format** to a specified GCS
2. **Rationale for Parquet**:
   - **Efficiency**:
     - Parquet is a columnar storage format that significantly reduces storage requirements while enhancing query performance.
   - **Schema Preservation**:
     - Parquet retains the schema, making it easier to reload and process the data in subsequent stages without re-specifying data types or structure.
   - **Compatibility**:
     - It is widely supported by big data tools and frameworks, ensuring smooth integration with downstream processing and analytics.

Simple Regression Models

Three regression models were implemented to predict AWS spot prices: Linear Regression, Decision Tree Regressor, and Random Forest Regressor. Each model is uniquely suited to capture different types of relationships between the features and the target variable.

1. **Linear Regression**: Linear regression models the relationship between features and the target variable as a linear equation. It serves as a baseline for comparison. Regularization parameters such as **regParam** and **elasticNetParam** are tuned to prevent overfitting and to balance L1 (Lasso) and L2 (Ridge) penalties.
2. **Decision Tree Regressor**: The decision tree regressor splits the data into subsets based on feature values, capturing non-linear relationships. Hyperparameters such as **maxDepth** (limiting tree depth) and **maxBins** (controlling feature splitting granularity) are optimized to enhance performance.
3. **Random Forest Regressor**: Random Forest, an ensemble method, combines predictions from multiple decision trees to improve accuracy and robustness. Key parameters include **numTrees** (number of trees in the forest), **maxDepth**, and **maxBins**. Random Forest is particularly effective for handling complex, high-dimensional data.

Hyperparameter Tuning

Hyperparameter tuning was performed using **ParamGridBuilder** and **TrainValidationSplit**. For each model, a grid of parameter values was created, and the model was trained and validated on different subsets of the training data. The best model for each algorithm was selected based on the Root Mean Squared Error (RMSE), ensuring optimal performance. This systematic approach balances model complexity and prediction accuracy.

NEURAL NETWORK:

Using a feed-forward neural network, this neural network model aimed to create a scalable, distributed deep learning pipeline that could forecast AWS spot instance prices using past data. This model is a component of a bigger pipeline for predicting AWS spot pricing.Since PySpark does not support custom neural network architectures, Elephas was used to facilitate distributed deep learning training. By combining Elephas, which enables Keras models to be trained in a distributed environment using Spark, and PySpark, which handles large-scale data processing, the distributed nature of the model was accomplished.

For training the deeper neural network, the preprocessed dataset was sampled to 50% of its original size, reducing it to approximately 13 million rows. This decision was driven by the computational intensity of neural networks and the limitations of the cluster configuration, which lacked GPU support due to quota constraints, making it impractical to handle the full dataset effectively.

Elephas: Distributed Deep Learning Made Simple

Elephas is a robust package that enables distributed training across Spark clusters by integrating Keras deep learning models with Apache Spark. For large-scale neural network training, it offers a smooth method of utilizing Spark's parallel computing capabilities. Elephas works by sharing the training data and model among several worker nodes in the cluster. After processing a portion of the data and updating the weights locally, each worker synchronizes the output with the master model. This method makes deep learning scalable and drastically cuts down on training time for big datasets. Elephas was set up in asynchronous mode for this project, meaning that model updates are carried out independently by each worker and aggregated on a regular basis.
As seen in this example, this approach works especially well for managing high-dimensional data and sizable batches. In order to achieve effective and scalable training for the AWS spot pricing prediction challenge, Elephas had to combine the distributed power of Spark with the ease of model-building of Keras.

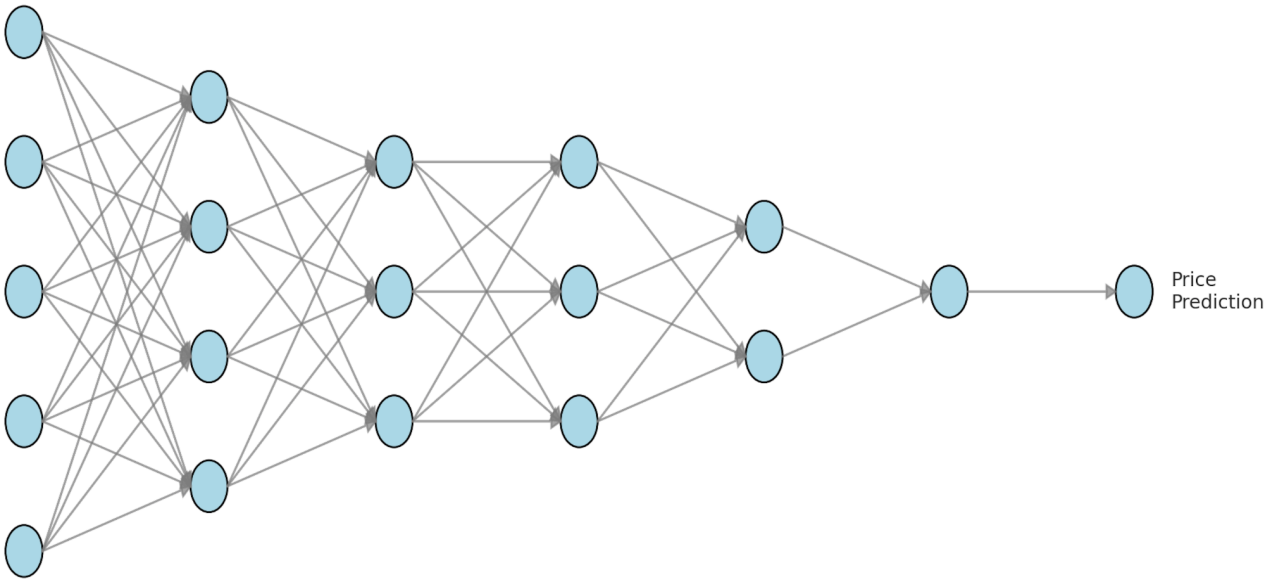
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from elephas.spark_model import SparkModel
from tensorflow.keras.optimizers.legacy import Adam
# Define the model
def create_model(input_dim):
    model = Sequential([
        Dense(512, activation='relu', input_dim=input_dim),
        Dropout(0.3),
        Dense(256, activation='relu'),
        Dropout(0.3),
        Dense(128, activation='relu'),
        Dropout(0.3),
        Dense(64, activation='relu'),
        Dropout(0.3),
        Dense(32, activation='relu'),
        Dense(1, activation='linear')  # For regression tasks
    ])
    model.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])
    return model

# Wrap the model with Elephas
input_dim = len(train_df.first()["features"])

keras_model = create_model(input_dim)
#
### Define early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

spark_model = SparkModel(model=keras_model, mode='asynchronous', num_workers=2)

# Fit the model with early stopping
spark_model.fit(train_data, epochs=50, batch_size=4096, verbose=1, validation_split=0.1, callbacks=[early_stopping])


# Prepare features RDD for prediction
features_rdd = test_df.rdd.map(lambda row: row["features"].toArray())
predictions = spark_model.predict(features_rdd)

num_partitions = test_df.rdd.getNumPartitions()

# # Convert predictions list to RDD with the same number of partitions as test_df
predictions_rdd = spark.sparkContext.parallelize(predictions, num_partitions)
```

Neural Network Code Using Elephas

Neural Network Architecture

A feed-forward neural network was designed to capture relationships between features and the target variable (price). The architecture is as follows:

1. **Input Layer**:
    - Number of neurons: Equal to the dimensionality of the input feature vector after one-hot encoding.
    - Activation: None (input fed directly into the hidden layers).
2. **Hidden Layers**:
    - **Layer 1**: 512 neurons, ReLU activation, 30% dropout.
    - **Layer 2**: 256 neurons, ReLU activation, 30% dropout.
    - **Layer 3**: 128 neurons, ReLU activation, 30% dropout.
    - **Layer 4**: 64 neurons, ReLU activation, 30% dropout.
    - **Layer 5**: 32 neurons, ReLU activation.
3. **Output Layer**:
    - Number of neurons: 1 (single output to predict `price`).
    - Activation: Linear, suitable for regression tasks.



```
+--------------------+--------------------+
|            features|               label|
+--------------------+--------------------+
|(96,[1,9,77],[1.0...|[0.00100174389473...|
|(96,[1,14,77],[1....|[0.00213728104930...|
|(96,[0,9,76],[1.0...|[6.20831061235457...|
|(96,[17,76],[1.0,...|[0.00221914891049...|
|(96,[0,21,77],[1....|[1.72453072499526...|
+--------------------+--------------------+
only showing top 5 rows

>>> Fit model
 * Serving Flask app 'elephas.parameter.server'
 * Debug mode: on
>>> Initialize workers
>>> Distribute load
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server i
 * Running on http://10.150.0.7:4000
Press CTRL+C to quit
10.150.0.6 - - [04/Dec/2024 17:29:44] "GET /parameters HTTP/1.1" 200 -
10.150.0.5 - - [04/Dec/2024 17:29:45] "GET /parameters HTTP/1.1" 200 -
10.150.0.6 - - [04/Dec/2024 17:29:56] "POST /update HTTP/1.1" 200 -
10.150.0.6 - - [04/Dec/2024 17:29:56] "GET /parameters HTTP/1.1" 200 -
10.150.0.5 - - [04/Dec/2024 17:29:57] "POST /update HTTP/1.1" 200 -
10.150.0.5 - - [04/Dec/2024 17:29:57] "GET /parameters HTTP/1.1" 200 -
10.150.0.6 - - [04/Dec/2024 17:30:06] "POST /update HTTP/1.1" 200 -
10.150.0.6 - - [04/Dec/2024 17:30:06] "GET /parameters HTTP/1.1" 200 -
10.150.0.5 - - [04/Dec/2024 17:30:08] "POST /update HTTP/1.1" 200 -
10.150.0.5 - - [04/Dec/2024 17:30:08] "GET /parameters HTTP/1.1" 200 -
```

Compilation Details:

- **Loss Function**: Mean Squared Error (`mse`), appropriate for continuous target variables.
- **Optimizer**: Adam optimizer with a learning rate of 0.001, chosen for its adaptive learning rate and efficient convergence.

Data Preparation:

- The training dataset was converted into an RDD of tuples (features, label) to align with Elephas' input requirements for distributed training.For training the deeper neural network, the dataset was sampled to 50% of its original size, reducing it to approximately 27 million rows. This decision was driven by the computational intensity of neural networks and the limitations of the cluster configuration, which lacked GPU support due to quota constraints, making it impractical to handle the full dataset effectively.

```
MSE: 0.00037115100464523587
RMSE: 0.019265279770749134
```

Model Evaluation

The models were evaluated on the test dataset using the following metrics:

1. **Root Mean Squared Error (RMSE)**: RMSE measures the average magnitude of prediction errors. A lower RMSE indicates better predictive accuracy.

2. **Mean Squared Error (MSE):**Measures the average squared difference between predicted and actual values.Lower MSE indicates better performance.

Each model was tested on the same test dataset, and their RMSE and R2 scores were recorded to compare their predictive power.

Metric	Linear Regression	Random Forest	Decision Tree	Neural Network
Mean Squared Error (MSE)	16.57466	12.8393	12.6515	0.000406
Root Mean Squared Error (RMSE)	4.0712	3.5832	3.5569	0.02102

Technologies and Tools Used:

1. **PySpark**:
    - Used for distributed data preprocessing, feature engineering, and splitting the dataset into training and testing subsets.
2. **Elephas**:
    - Enabled distributed training of the Keras model on a Spark cluster.
    - Asynchronous mode was particularly useful for parallel updates to model weights.
3. **Keras and TensorFlow**:
    - Designed the neural network architecture.
    - Provided high-level APIs for model compilation and evaluation.
4. **Google Cloud Storage**:
    - Hosted the preprocessed dataset as a Parquet file for scalability and accessibility.
5. **Google Dataproc**:
    - Managed the Spark cluster for distributed processing and training tasks.

This neural network model was successful in predicting AWS spot prices with high accuracy and efficiency. The combination of distributed data processing (PySpark) and distributed training (Elephas) enabled the model to handle large-scale data effectively. The use of early stopping ensured the model was robust against overfitting, and the low RMSE validated its predictive power.

Conclusion

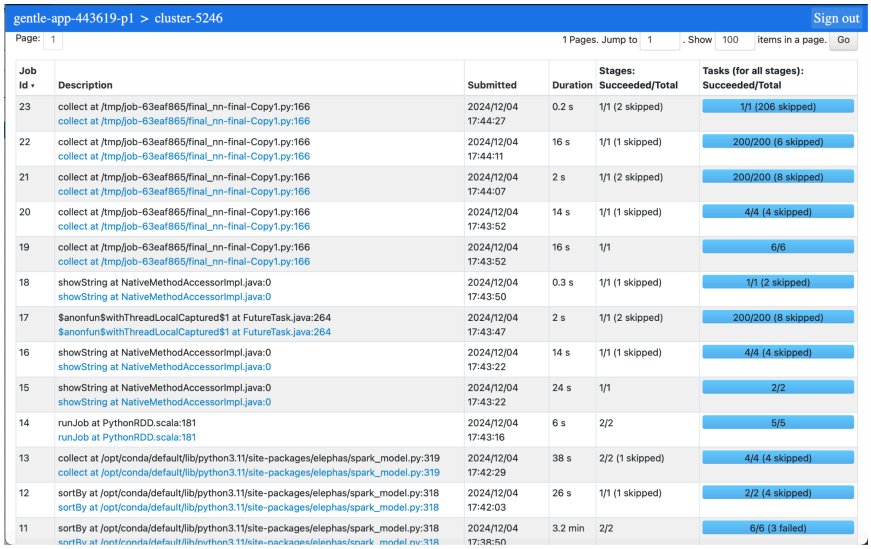
The AWS Spot Price Prediction System demonstrates the effective integration of distributed data processing and advanced machine learning techniques to solve a practical challenge in dynamic pricing environments. By leveraging historical spot pricing data, the system accurately predicts future price trends, empowering AWS users to optimize their resource utilization strategies.

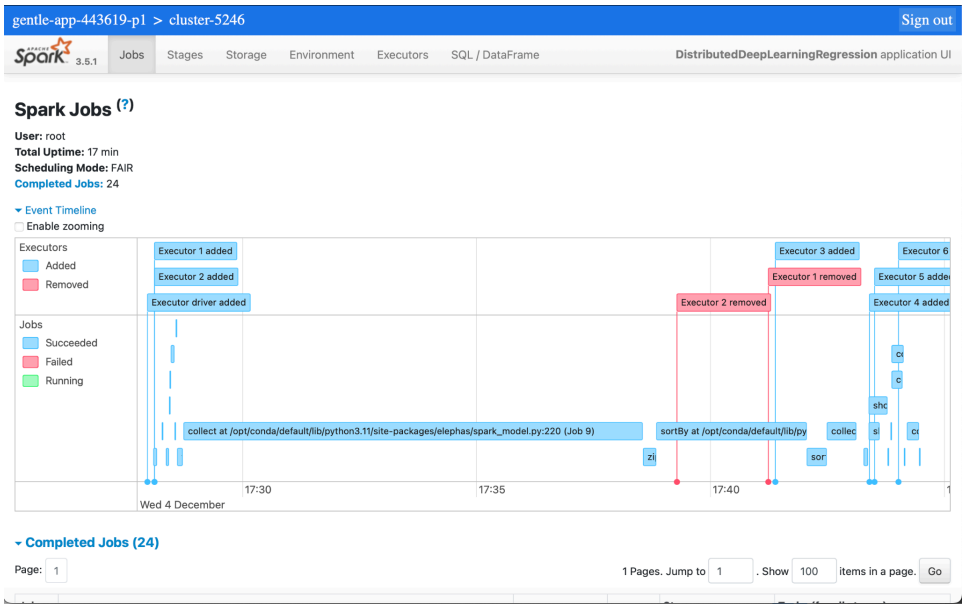
Through a robust preprocessing pipeline, regression models, and a distributed deep learning architecture, the project achieves high predictive accuracy while maintaining scalability. The implementation of tools such as PySpark, Elephas, and Google Cloud technologies ensures efficient handling of large-scale data, highlighting the potential of distributed systems in big data analytics.

This system not only addresses cost optimization but also enables businesses to mitigate risks associated with spot price volatility, fostering confidence in utilizing AWS spot instances. By successfully bridging data science and cloud technologies, the project underscores the value of innovative approaches in addressing real-world computational challenges.

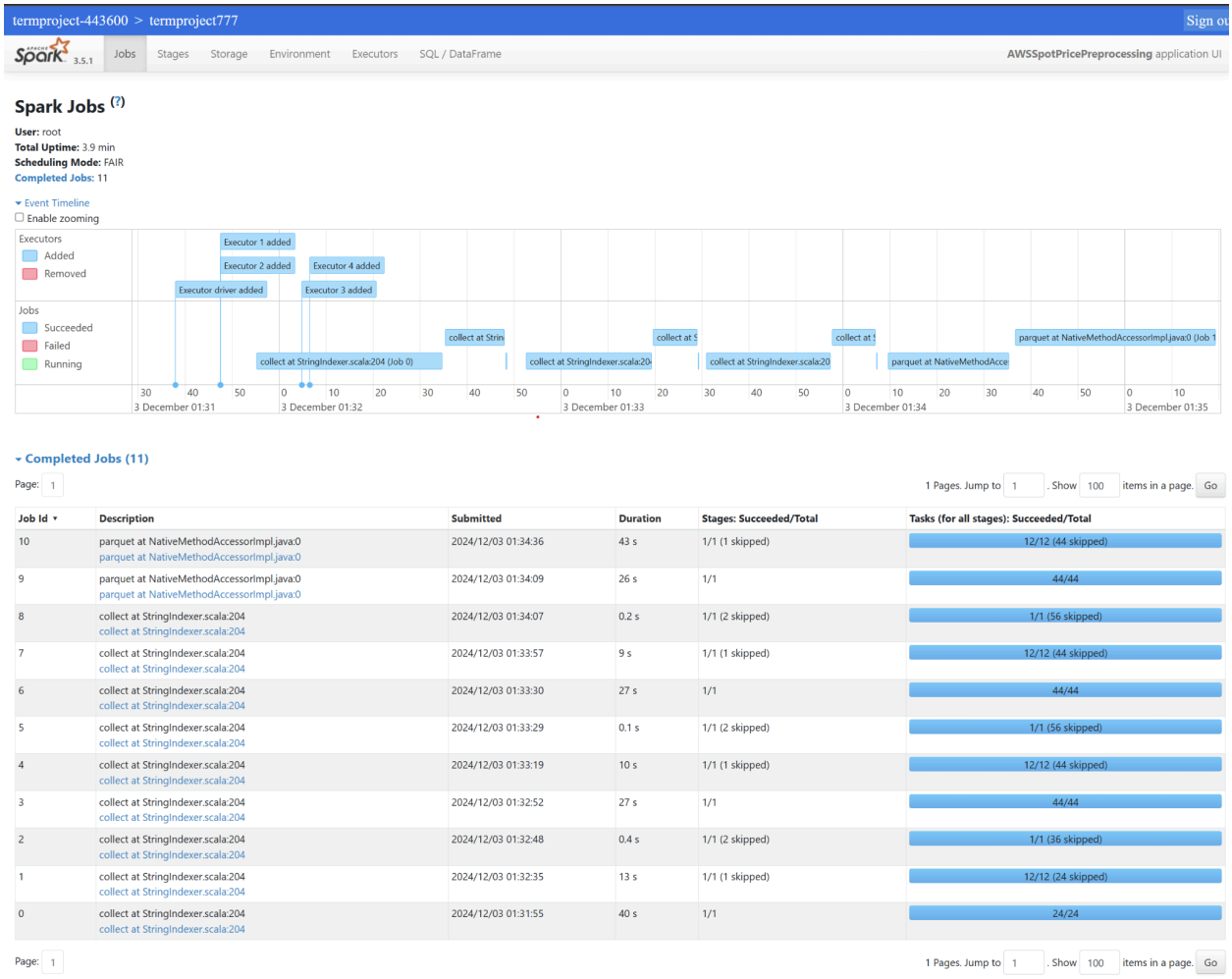
SPARK SERVER HISTORY:

FOR NN:





Data Preprocessing Spark Job:



Completed Jobs (11)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
10	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/12/03 01:34:36	43 s	1/1 (1 skipped)	12/12 (44 skipped)
9	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/12/03 01:34:09	26 s	1/1	44/44
8	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:34:07	0.2 s	1/1 (2 skipped)	1/1 (56 skipped)
7	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:33:57	9 s	1/1 (1 skipped)	12/12 (44 skipped)
6	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:33:30	27 s	1/1	44/44
5	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:33:29	0.1 s	1/1 (2 skipped)	1/1 (56 skipped)
4	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:33:19	10 s	1/1 (1 skipped)	12/12 (44 skipped)
3	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:32:52	27 s	1/1	44/44
2	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:32:48	0.4 s	1/1 (2 skipped)	1/1 (36 skipped)
1	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:32:35	13 s	1/1 (1 skipped)	12/12 (24 skipped)
0	collect at StringIndexer.scala:204 collect at StringIndexer.scala:204	2024/12/03 01:31:55	40 s	1/1	24/24

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Linear Regression, Ensemble Learning and Tree-Based Regressor Spark Server History: