Flight Price Prediction

BIA 678-B 2024 Fall

Aradhana Ramamoorthy

Tzu Hsuan Lin

Varsha Abraham

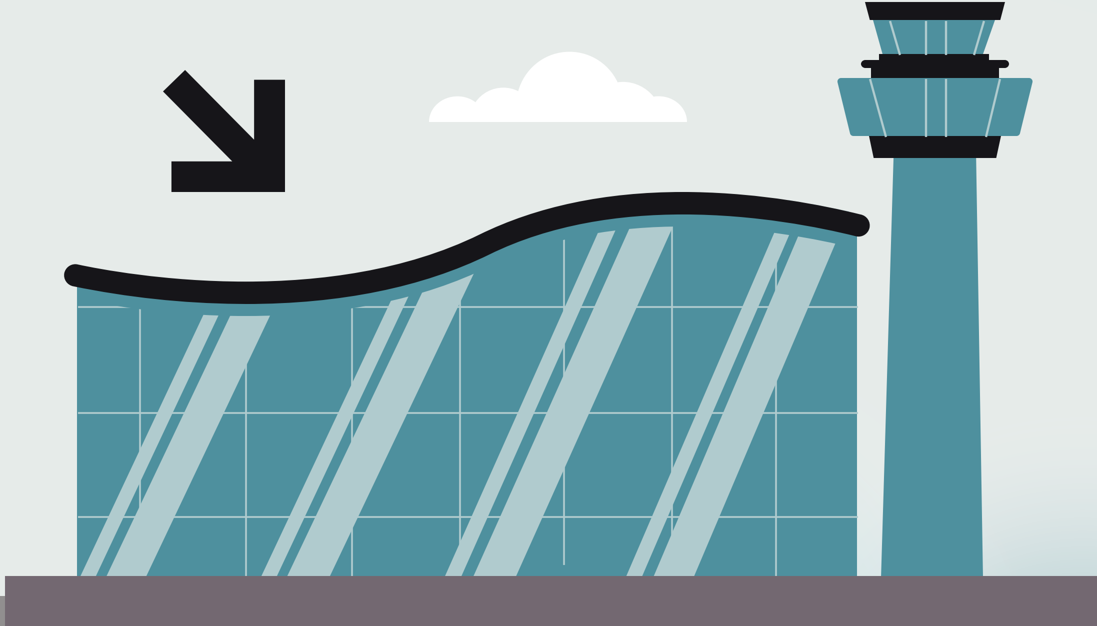
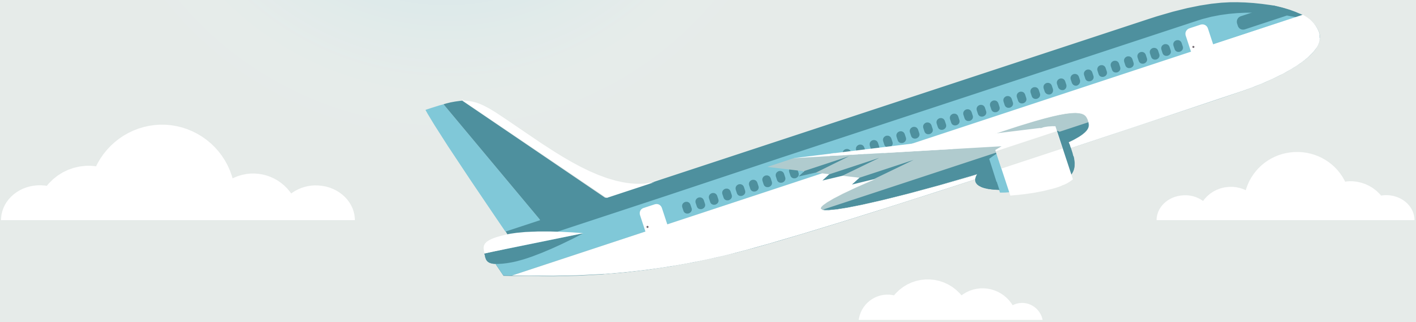


Table of *contents*

Table of *contents*

01

Problem Statement



Background

The flight booking industry is highly dynamic, with prices fluctuating based on demand, seasonality, and external factors. Predicting flight prices can empower consumers to make informed booking decisions and help airlines optimize revenue.

Problem statement

To develop a scalable regression-based predictive model to accurately predict flight prices based on historical data trends, enabling consumers to make informed decisions about flight bookings.

Objective

To develop a scalable regression-based predictive model to estimate flight prices using historical data, helping consumers make cost-effective booking decisions and evaluating scalability metrics for big data processing.

02

Data Overview

Data Collection

Dataset: Flight prediction Prices
Sourced: Expedia, through Kaggle
Rows: 82 million rows
Storage: GCS and processed using Apache Spark
Time frame: April 2022 to October 2022
Key features: arrival and destination airports, travel duration, travel dates, seat availability, and ticket prices.

03

Data Processing

Data Cleaning

1. **Dropping Irrelevant Columns** - fareBasisCode, elapsedDays
2. **Handling Missing Values** - missing totalTravelDistance values
3. **Dropping Temporary Columns** - flight_month and flight_day
4. **Filtering Data** – Bottom 50% of routes
5. **Schema Validation**
6. **Removing Redundant Rows** - filtering on legId

Feature Engineering

1. **Adding** `days_until_flight` **between** `flightDate` **and** `searchDate`
2. **Adding** `is_peak_season`
3. **Filling Missing Values with Averages**
   Average `totalTravelDistance`

04

Exploratory Data Analysis

EDA - Basic Statistics

- **3,915,873** unique flights, non-refundable
- **125** distinct routes(starting – destination airport)

Total Fare

$344.3	$202.6	$311.1	$457.1
Mean Min: 19.59/ Max: 8260.61	25th Percentile	50th Percentile	75th Percentile

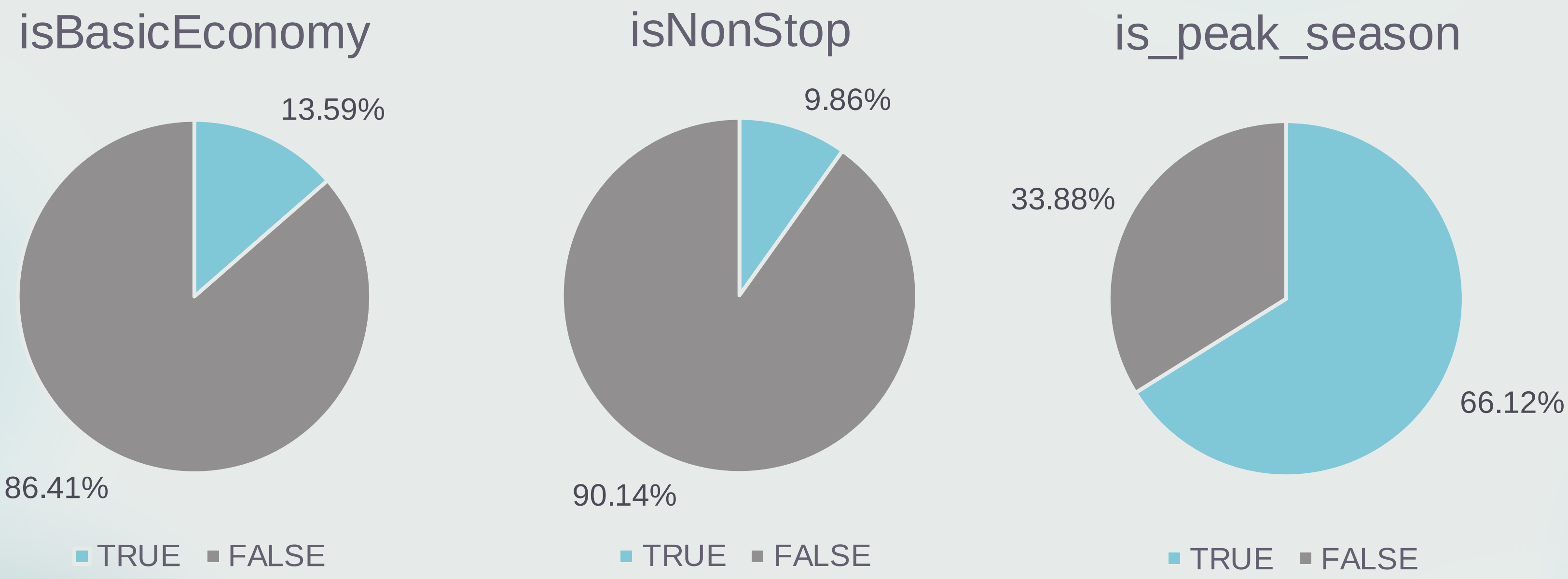
643,726

Outliers
exceeding $835.085

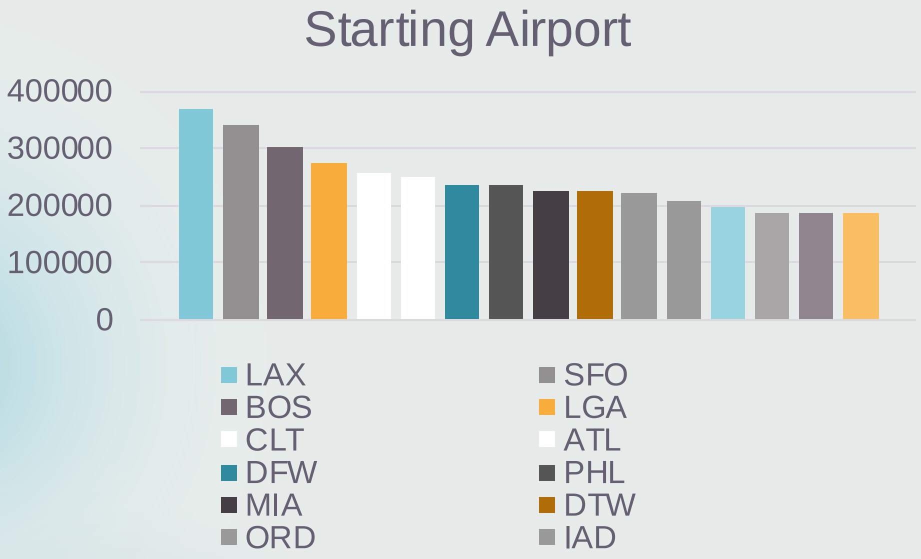
- Scenarios such as high-demand seasons, exceptional booking behavior, or airline-specific pricing strategies
- Improve Model Generalization to handle a wide range of scenarios

EDA - Basic Statistics

- TotalTravelDistance: ranges from 89 to 7,252 miles, showing a mix of short and long-haul flights



EDA – Airport Distribution



EDA - Correlation Analysis

- Correlation Between totalFare and the numerical features

0.04	0.4	-0.05
seatsRemaining	**totalTravelDistance**	**days_until_flight**

- Correlation between totalTravelDistance and totalFare for each route

startingAirport	destinationAirport	correlation
JFK	BOS	0.7089998621891911
BOS	LGA	0.5255915934690403
IAD	BOS	0.5143842681361767
DTW	JFK	0.477477396245983
OAK	ORD	0.45898514946458524
MIA	LGA	0.43267881243554984
JFK	DFW	0.41106129473904596
OAK	DEN	0.3966772483310056
OAK	LAX	0.3844373485319693
JFK	ORD	0.37331433055258884
LGA	IAD	0.35588517032179295
IAD	LGA	0.35432847612316565
IAD	EWR	0.3244639255171254
PHL	LGA	0.32319192661651547
ORD	JFK	0.3206619258593134
DEN	SFO	0.32010566788474076
CLT	MIA	0.2804493239435498
DFW	LGA	0.2516080963493985
MIA	CLT	0.24616551791871233
LAX	ORD	0.24544673587618937

- Route-specific trends
- Variability in Correlation Strength
- Potential Influencing Factors: Airline competition, demand variability, etc

05

Modeling

Computing Resource and Platform used

GCP

Storage & compute

Apache Spark

Distributed data processing

Dataproc

Spark cluster management

Jupyter

Development & testing

Parquet

Data storage format

Predictor & Target Variables

Predictor

IsBasicEconomy
IsRefundable
IsNonStop
seatsRemaining
totalTravelDistance
daysUntilFlight
Is_peak_season
travelDurationMinutes
StartingAirportIndex
destinationAirportIndex

Target

totalFare

Encoding

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder

# Step 1: StringIndexer for starting and destination airports
airport_indexer = StringIndexer(inputCol="startingAirport", outputCol="startingAirportIndex")
dest_airport_indexer = StringIndexer(inputCol="destinationAirport", outputCol="destinationAirportIndex")

# Transform the data
df_enc = airport_indexer.fit(df_all).transform(df_all)
df_enc = dest_airport_indexer.fit(df_enc).transform(df_enc)

# Step 2: OneHotEncoder for starting and destination airports
airport_encoder = OneHotEncoder(inputCol="startingAirportIndex", outputCol="startingAirportVec")
dest_airport_encoder = OneHotEncoder(inputCol="destinationAirportIndex", outputCol="destinationAirportVec")

# Transform the data
df_enc = airport_encoder.fit(df_enc).transform(df_enc)
df_enc = dest_airport_encoder.fit(df_enc).transform(df_enc)
```

```
+---------------+--------------------+------------------+
|startingAirport|startingAirportIndex|startingAirportVec|
+---------------+--------------------+------------------+
|ATL            |6.0                 |(15,[6],[1.0])    |
|ATL            |6.0                 |(15,[6],[1.0])    |
|ATL            |6.0                 |(15,[6],[1.0])    |
|ATL            |6.0                 |(15,[6],[1.0])    |
|ATL            |6.0                 |(15,[6],[1.0])    |
+---------------+--------------------+------------------+
only showing top 5 rows

+------------------+-----------------------+---------------------+
|destinationAirport|destinationAirportIndex|destinationAirportVec|
+------------------+-----------------------+---------------------+
|BOS               |4.0                    |(15,[4],[1.0])       |
|BOS               |4.0                    |(15,[4],[1.0])       |
|BOS               |4.0                    |(15,[4],[1.0])       |
|BOS               |4.0                    |(15,[4],[1.0])       |
|BOS               |4.0                    |(15,[4],[1.0])       |
+------------------+-----------------------+---------------------+
```

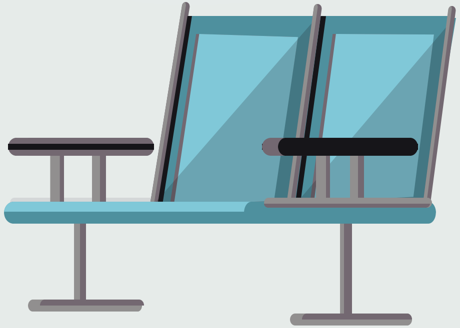
```
from pyspark.sql.functions import col

# Convert boolean columns to integer
df_enc = df_enc.withColumn("isBasicEconomy", col("isBasicEconomy").cast("integer"))
df_enc = df_enc.withColumn("isRefundable", col("isRefundable").cast("integer"))
df_enc = df_enc.withColumn("isNonStop", col("isNonStop").cast("integer"))
df_enc = df_enc.withColumn("is_peak_season", col("is_peak_season").cast("integer"))
```

Linear Regression

Modeling for Linear Regression

Chosen as a baseline to evaluate performance using models like GBT, Linear Regression predicts a target variable by modeling the relationship between features and the target as a straight line. It's simple, interpretable, and a good baseline for regression tasks.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Step 1: Define predictor columns and target column
predictor_columns = [
    "isBasicEconomy", "isRefundable", "isNonStop", "seatsRemaining",
    "totalTravelDistance", "days_until_flight", "is_peak_season",
    "travelDurationMinutes", "startingAirportIndex", "destinationAirportIndex"
]
target_column = "totalFare"

# Step 2: Vectorize predictors (combine all predictor columns into one feature vector)
vector_assembler = VectorAssembler(inputCols=predictor_columns, outputCol="features")
df_vectorized = vector_assembler.transform(df_enc)

# Step 3: Split Data into Training and Test Sets
train_data, test_data = df_vectorized.randomSplit([0.8, 0.2], seed=42)

# Step 4: Define and Train the Linear Regression Model
lr = LinearRegression(featuresCol="features", labelCol=target_column)

# Train the model
lr_model = lr.fit(train_data)

# Step 5: Evaluate the Model on the Test Set
predictions = lr_model.transform(test_data)
```



Gradient Boosted Trees

Modeling for Gradient Boosted Trees

Chosen for its ability to capture non-linear patterns in the flight price data and prediction accuracy, Gradient Boosting Trees build a series of decision trees where each tree corrects the errors of the previous ones, creating a robust predictive model. It handles complex relationships and nonlinear data well.

```
# Step 1: Define predictor columns and target column
predictor_columns = [
    "isBasicEconomy", "isRefundable", "isNonStop", "seatsRemaining",
    "totalTravelDistance", "days_until_flight", "is_peak_season",
    "travelDurationMinutes", "startingAirportIndex", "destinationAirportIndex"
]
target_column = "totalFare"

# Step 2: Vectorize predictors (combine all predictor columns into one feature vector)
vector_assembler = VectorAssembler(inputCols=predictor_columns, outputCol="features")
df_vectorized_10pct = vector_assembler.transform(df_10pct)

# Step 3: Split Data into Train and Test Sets
train_data_10pct, test_data_10pct = df_vectorized_10pct.randomSplit([0.8, 0.2], seed=42)

# Step 4: Define Gradient-Boosted Trees Regressor and parameters
gbt = GBTRegressor(featuresCol="features", labelCol=target_column, maxIter=100, maxDepth=6, seed=42)

# Step 5: Train the model and measure training time
start_time = time.time()
gbt_model_10pct = gbt.fit(train_data_10pct)
training_time_10pct = time.time() - start_time
print(f"Training Time (10% Dataset): {training_time_10pct:.2f} seconds")

# Step 6: Make predictions and measure inference time
start_time = time.time()
predictions_10pct = gbt_model_10pct.transform(test_data_10pct)
inference_time_10pct = time.time() - start_time
print(f"Inference Time (10% Dataset): {inference_time_10pct:.2f} seconds")
```

06

Modeling Evaluation

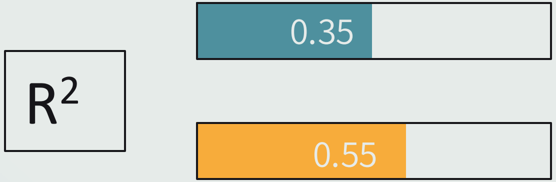
Linear Regression vs. Gradient Boosted Trees

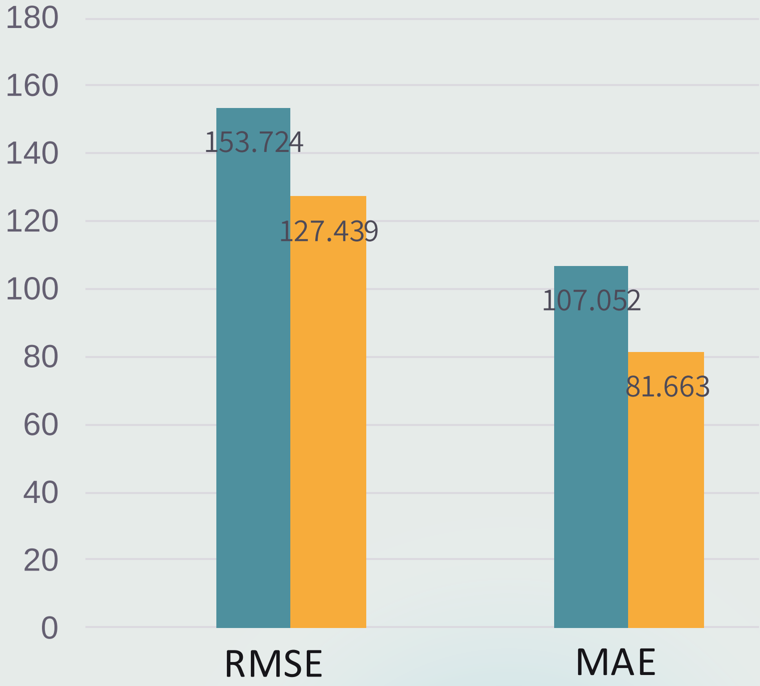
Linear Reg.

Relatively large errors (both RMSE and MAE) and the low R^2 explains a smaller portion of the variability in the data.

GBT

Performs better overall: smaller errors (lower RMSE and MAE) and a better fit (higher R^2)





07

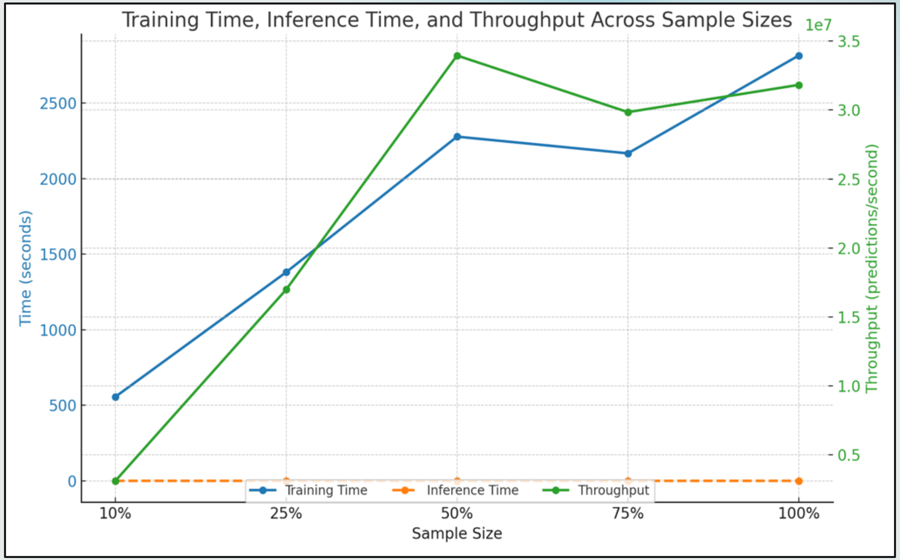
Scale Up & Scale Out

Cluster configuration

The cluster configuration varies with dataset size to optimize resource utilization, maintain performance, and balance between training/inference times and throughput.

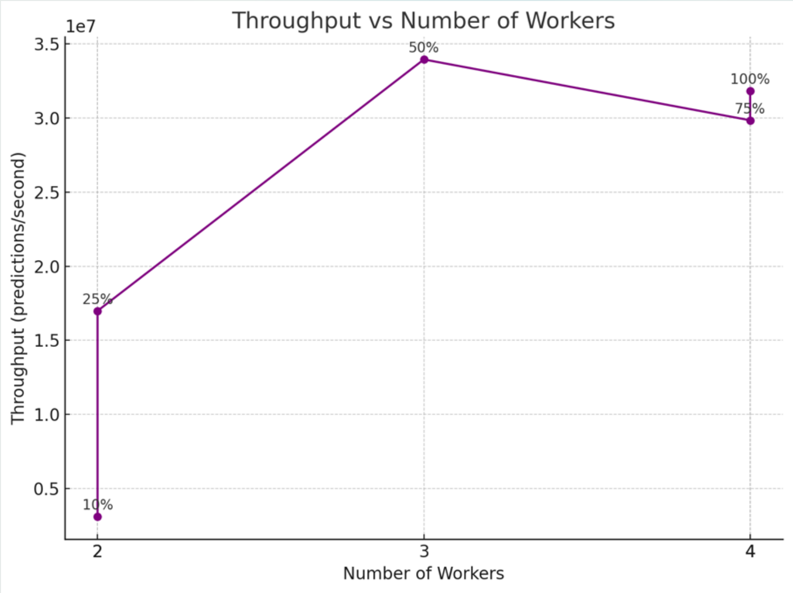
Dataset Sample	Machine Type	vCPUs per Node	Physical Cores per Node	Memory per Node	Number of Worker Nodes
10% Sample	n1-standard-4	4	2 (hyper-threaded)	15 GB	2
25% Sample	n1-standard-4	4	2 (hyper-threaded)	15 GB	2
50% Sample	n1-standard-4	4	2 (hyper-threaded)	15 GB	3
75% Sample	n1-standard-4	4	2 (hyper-threaded)	15 GB	4
100% Sample	n1-standard-4	4	2 (hyper-threaded)	15 GB	4

Scaling up



Sample Size	Training Time (s)	Inference Time (s)	Throughput (predictions/sec)
10%	556.41	0.34	3116812.72
25%	1380.63	0.16	16986634.29
50%	2277.56	0.16	33952669.61
75%	2166.97	0.27	29848675.93
100%	2814.22	0.33	31823632.0

Scaling out



Throughput vs Number of Workers
1e7
3.5
3.0
2.5
2.0
1.5
1.0
0.5
Throughput (predictions/second)
50%
100%
75%
25%
10%
2
3
4
Number of Workers

08

Conclusions

Conclusion

Metrics	10%	25%	50%	75%	100%
Training Time	556.41 sec	1380.63 sec	277.56 sec	2166.97 sec	2814.22 sec
Inference Time	0.34 sec	0.16 sec	0.16 sec	0.27 sec	0.33 sec
RMSE	28.061	127.524	127.593	127.398	127.439
MAE	81.859	81.999	81.830	81.733	81.664
R^2	0.549	0.550	0.552	0.553	0.553
Throughput	3116812.72 preds/sec	16986634.29 preds/sec	33952669.61 preds/sec	29848675.93 preds/sec	31823632.00 preds/sec

Conclusion

Scalability achieved

Training time increases with dataset size but remains manageable due to distributed computing.
Throughput consistently scales, demonstrating efficiency in handling larger datasets.

Model Performance

Metrics (RMSE, MAE, R^2) remain stable across dataset sizes, indicating the model's robustness.
R^2 improves slightly with larger datasets, reaching **0.553** for the full dataset.

Resource Utilization

Optimal configurations of worker nodes, executor memory, and cores ensured efficient processing.
Larger datasets required *higher instances and memory*, but training remained feasible **within** allocated resources.

Thanks

Any Questions?