

AWS data pipeline – Used Car price Analysis

Introduction

This project implements a comprehensive data pipeline leveraging AWS cloud services and PySpark to analyze and predict used car prices from a large-scale dataset. The pipeline orchestrates multiple components, starting with raw data storage in S3, followed by data processing using Jupyter Notebooks for cleaning and feature engineering. The processed data then flows through parallel paths - one for SQL analysis using Python scripts, and another for machine learning model development using Amazon SageMaker AutoML. The pipeline culminates with AWS QuickSight for data visualization and insights delivery to end users. This architecture ensures efficient data processing, scalable machine learning operations, and interactive visualization capabilities, all while maintaining data integrity and processing efficiency throughout the workflow. The implementation demonstrates the effective use of cloud-native services for handling complex data processing and machine learning tasks in a production environment.

Dataset overview

The dataset in question is a comprehensive collection of used car listings, encompassing a wide range of vehicle attributes and market information. With a size of 100 MB, it provides a rich source of data for analyzing the used car market and developing predictive models. The dataset captures various aspects of each vehicle, including basic information like make and model, technical specifications, aesthetic features, geographic details, and pricing information. This dataset serves as the foundation for building a machine learning model to predict the prices of used cars. It offers a diverse set of features that can potentially influence a vehicle's value, making it ideal for exploring complex relationships between car attributes and market prices. The inclusion of both structured data (such as numeric and categorical fields) and unstructured data (like descriptions) allows for the application of various data analysis and machine learning techniques,

enabling a comprehensive approach to price prediction in the used car market.

Project Objective

This project implements a comprehensive big data pipeline leveraging AWS cloud services and PySpark to analyze and predict used car prices from a large-scale dataset. The pipeline orchestrates multiple components to efficiently process, analyze, and visualize data from raw ingestion to actionable insights.

Methodology

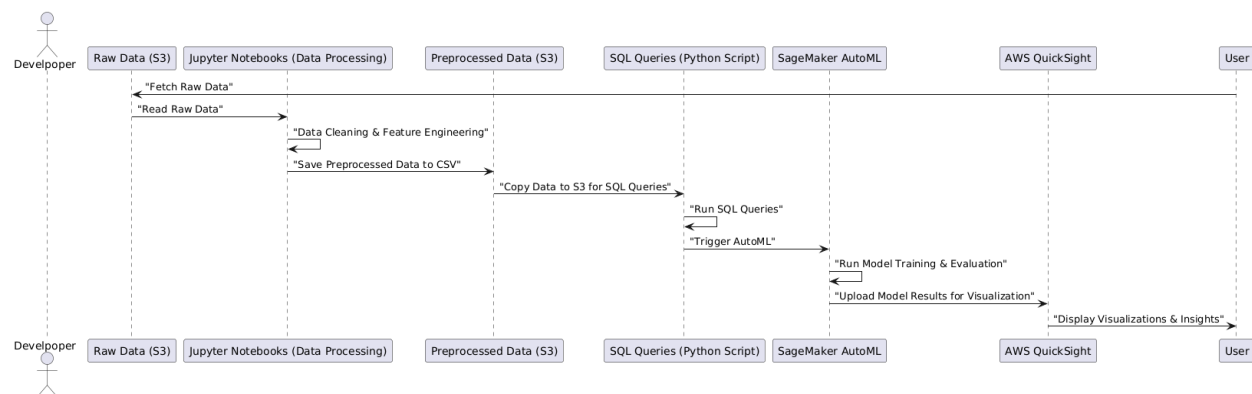


Fig 1- Architecture Sequence diagram of the Data pipeline

Environment Setup:

I have used the boto3 library of python provided by AWS to interact with the AWS services programmatically.

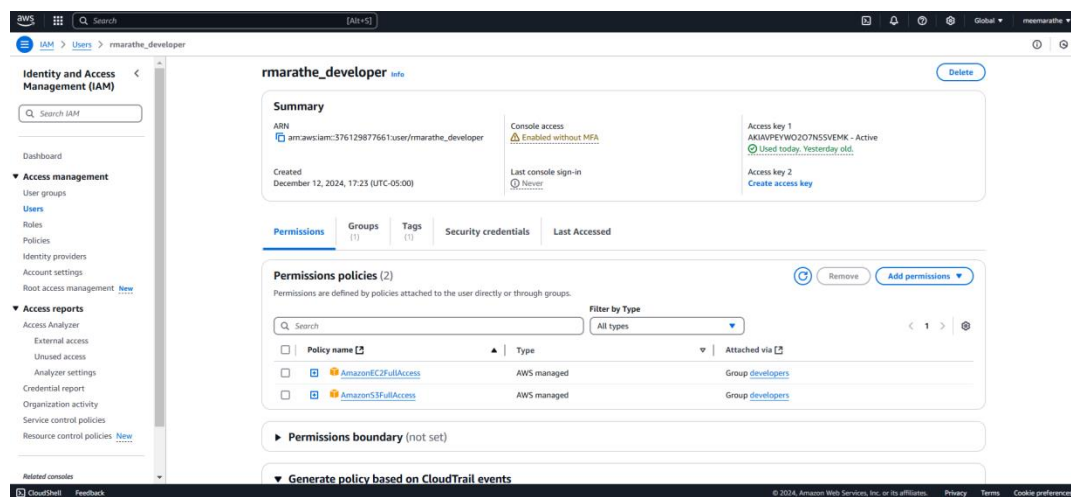


Fig 2 – Setting up of permission for user to access S3 and EC2 instance.

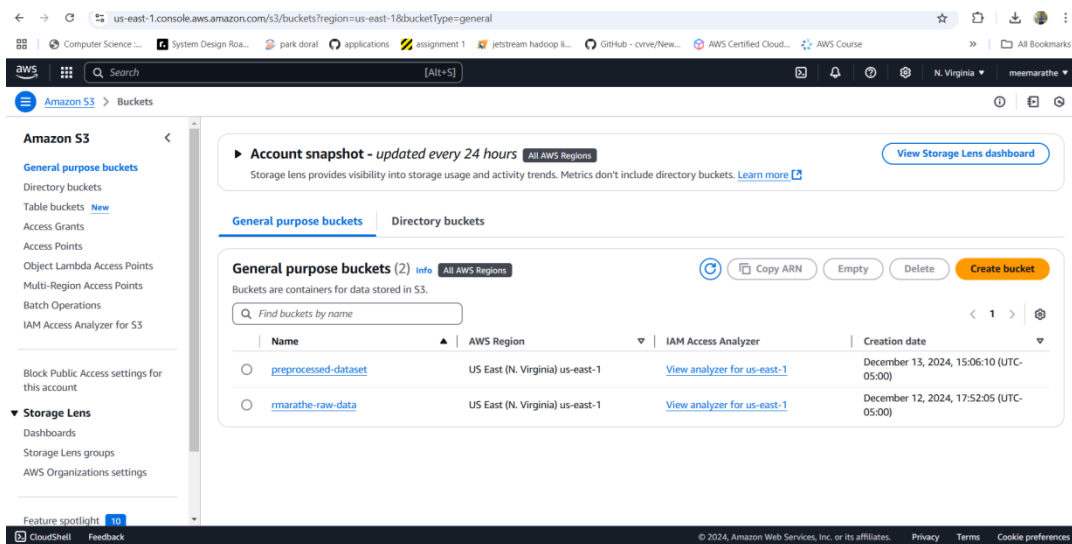


Fig 3 – The uploaded csv files are present in the S3 buckets seen above.

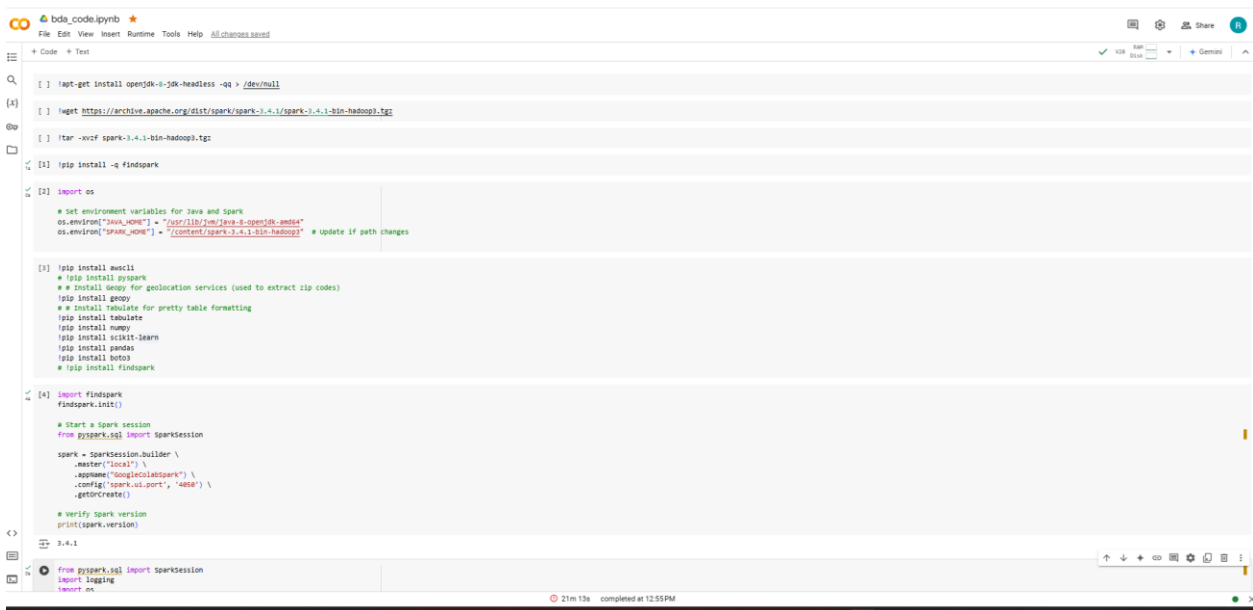


Fig 4 – Installing the libs and downloading the python, pyspark, hadoop packages to match the right version.

Data Ingestion and preprocessing

I have used Google collab to run the data ingestion from AWS S3 using boto3 package and preprocessing using pyspark.

[**Challenge** faced when tried with EC2 due to resource limitation for free tier as the dataset was large]

```
bda_13_12_2024.ipynb
File Edit View Insert Runtime Tools Help All changes saved

[4]: Verify Spark version
print(spark.version)

3.4.1

[5]: from pyspark.sql import SparkSession
import logging
from tabulate import tabulate # Import tabulate for better table formatting

# Set up the logger
log_file = "output.log"

# If the log file already exists, remove it to create a fresh file
if os.path.exists(log_file):
    os.remove(log_file)

logger = logging.getLogger()
logger.setLevel(logging.INFO)

# Create a file handler to write log messages to a file (overwrite the log file each time)
# Specify utf-8 encoding to handle special characters
file_handler = logging.FileHandler(log_file, mode="w", encoding="utf-8") # Use 'w' mode to overwrite the log file
file_handler.setLevel(logging.INFO)

# Create a log formatter
formatter = logging.Formatter("%(message)s")
file_handler.setFormatter(formatter)

# Add the file handler to the logger
logger.addHandler(file_handler)

# Function to log and print information messages (not DataFrame content)
def log_message(message, title=None):
    if title:
        logger.info(f"=== {title} ===")
    logger.info(message)
    # Optionally print to the console for local execution
    print(message)

# Function to log and print the output of show() from DataFrame in a table format
def log_show_output(spark_df=None, title=None):
    if spark_df and title:
        logger.info(f"=== {title} ===")
        # Get the column names from the DataFrame
        columns = spark_df.columns
        # Get the top 5 rows
        rows = spark_df.take(5)
        # Format the rows into a list of lists, ready for tabulation
        formatted_rows = [list(row) for row in rows]
        # Log the table as a string using tabulate
        table = tabulate(formatted_rows, headers=columns, tablefmt="grid")
        # Log and print the table
        logger.info(table)
        print(table) # Optionally print to the console for local execution
```

Fig 5- Logging enabled for debugging purposes – logs are dumped to a file

```
bda_13_12_2024.ipynb
File Edit View Insert Runtime Tools Help All changes saved

[6]: Import as pd
import pandas as pd
import io # Import the io module to use StringIO
from pyspark.sql import SparkSession

# Set up AWS credentials as environment variables (this can be done securely)
os.environ["AWS_ACCESS_KEY_ID"] = "AKIAVPEK027H5E9EH"
os.environ["AWS_SECRET_ACCESS_KEY"] = "7p9n0e0r7Spw6cJ7v9u7f5w6c0t7v9p0q0t"
os.environ["AWS_DEFAULT_REGION"] = "us-east-1" # Replace with your AWS region

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("S3 Data Project") \
    .config("spark.executor.memory", "1g") \
    .config("spark.driver.memory", "1g") \
    .config("spark.kryoserializer.buffer", "32m") \
    .config("spark.kryoserializer.buffer.max", "1g") \
    .config("spark.sql.shuffle.partitions", "100") \
    .config("spark.default.parallelism", "100") \
    .getOrCreate()

# Use boto3 to interact with S3 (Initialize a session)
s3_client = boto3.client('s3')

# Define the S3 bucket and file
bucket_name = "marathe-raw-data"
file_key = "car_dataset.csv"

# Generate a signed URL to access the S3 file
s3_url = f"s3://{bucket_name}/{file_key}"

# Read the file into a pandas DataFrame (using boto3 to fetch it)
# Using boto3's s3 client to get the file as a CSV
obj = s3_client.get_object(Bucket=bucket_name, Key=file_key)
data = obj['Body'].read().decode('utf-8') # Read the file as a string

# Use io.StringIO to read the string data as if it were a file
df_pandas = pd.read_csv(io.StringIO(data)) # Convert to pandas DataFrame

# Convert pandas DataFrame to PySpark DataFrame
df = spark.createDataFrame(df_pandas)

# Get the shape (row count and column count)
row_count = df.count() # Number of rows
column_count = len(df.columns) # Number of columns

# Log the shape of the DataFrame
log_message(f"Shape of the DataFrame: ({row_count}, {column_count})")

# Log and print the top 5 rows of the DataFrame as a table
log_show_output(spark_df=df, title="Top 5 Rows of the DataFrame")

INFO:boto3.credentials:Found credentials in environment variables.
INFO:root:Shape of the DataFrame: (30781, 26)
INFO:root:Top 5 Rows of the DataFrame ==
|         id | url | region | region_url | price | year | manufacturer | model | condition | cylinders | fuel | odometer | title_status | transmission | vin |
```

Fig 6 – Setting up the spark session to interact with dataset in AWS S3 bucket and reading raw file

```
bda_13_12_2024.ipynb
File Edit View Insert Runtime Tools Help Last saved at 5:56 PM

+ Code + Text

bucket_name = "marathe-raw-data"
file_key = "car_dataset.csv"

# Generate a signed URL to access the S3 file
s3_url = f"s3://{bucket_name}/{file_key}"

# Read the file into a pandas DataFrame (using boto3 to fetch it)
# using boto3's s3 client to get the file as a CSV
obj = s3_client.get_object(Bucket=bucket_name, Key=file_key)
data = obj['Body'].read().decode('utf-8') # Read the file as a string

# Use io.StringIO to read the string data as if it were a file
df_pandas = pd.read_csv(io.StringIO(data)) # Convert to pandas DataFrame

# Convert pandas DataFrame to PySpark DataFrame
df = spark.createDataFrame(df_pandas)
# Get the shape (row count and column count)
row_count = df.count() # Number of rows
column_count = len(df.columns) # Number of columns
# Log the shape of the DataFrame
log_message(message=f"Shape of the DataFrame: ({row_count}, {column_count})")
# Log and print the top 5 rows of the DataFrame as a table
log_show_output(spark_df=df, title="Top 5 Rows of the DataFrame")

INFO:boto3.credentials:Found credentials in environment variables.
INFO:root:Shape of the DataFrame: (38781, 26)
INFO:root:--- Top 5 Rows of the DataFrame ---
INFO:root:-----
| id | url | region | region_url | price | year | manufacturer | model | condition | cylinders | fu
| 7313747540 | https://baltimore.craigslist.org/cfo/d/baltimore-dodge-caravan/7313747540.html | baltimore | https://baltimore.craigslist.org | 48000 | 2014 | dodge | grand caravan sport | like new | 6 cylinders | ga
| 7311507956 | https://boulder.craigslist.org/cfo/d/boulder-2011-ford-mustang-manual/7311507956.html | boulder | https://boulder.craigslist.org | 10000 | 2011 | ford | mustang | good | 6 cylinders | ga
| 7302151520 | https://madison.craigslist.org/cfo/d/scomomoc-2012-ford-econoline-cargo/7302151520.html | madison | https://madison.craigslist.org | 11785 | 2013 | ford | econoline cargo van | good | 8 cylinders | ga
| 731400012 | https://seattle.craigslist.org/cfo/d/tempe-2012-nissan-juke-sl-mech-finance/731400012.html | seattle | https://seattle.craigslist.org | 7900 | 2012 | nissan | juke | excellent | 4 cylinders | ga
| 7318297143 | https://hickory.craigslist.org/cfo/d/hickory-2018-kia-forte-lx-sedan-4d/7318297143.html | hickory / lenoir | https://hickory.craigslist.org | 15900 | 2018 | kia | forte lx sedan 4d | good | n/a | ga
```

Fig 7 – Data ingested successfully and displayed in the terminal.

```
bda_13_12_2024.ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

# Replace non-standard null values with None
df = df.replace(["", "NA", "None", "null", "nan"], None)

# Check for null and nan counts for each column
null_counts = df.select(
    _sumwhen(col(c).isNull() | isnan(col(c)), 1).otherwise(0)).alias(c)
    for c in df.columns
)
null_counts_row = null_counts.collect()[0].asDict()
log_show_output(spark_df=df, title="Top 5 Rows of the DataFrame")
log_message(message=null_counts_row)

INFO:root:--- Top 5 Rows of the DataFrame ---
INFO:root:-----
| price | year | manufacturer | model | condition | cylinders | odometer | transmission | size | type | paint_color | county | state | lat | long | posting_date |
| 7990 | 2012 | nissan | juke | excellent | 4 cylinders | 12085 | automatic | mid-size | SUV | white | nan | fl | 28.827 | -82.4999 | 2021-04-23T14:45:12-0400 |
| 18000 | 2013 | subaru | impreza wrx | 4 cylinders | 100724 | manual | wagon | silver | nan | wa | 47.0929 | -127.412 | 2021-05-04T18:00:56-0700 |
| 28000 | 2012 | ford | f-150 | 8 cylinders | 93781 | automatic | truck | black | nan | wa | 47.0929 | -127.412 | 2021-04-19T19:11:18-0700 |
| 6900 | 2009 | gmc | sierra | fair | 8 cylinders | 250000 | automatic | full-size | truck | red | nan | oh | 39.2553 | -84.4619 | 2021-04-20T19:07:51-0400 |
| 14000 | 2008 | ram | 1500 big horn | excellent | 8 cylinders | 128290 | automatic | full-size | truck | black | nan | ri | 41.8221 | -71.3539 | 2021-04-23T11:15:12-0400 |
INFO:root:('price': 0, 'year': 31, 'manufacturer': 546, 'models': 136, 'condition': 5047, 'cylinders': 9181, 'odometer': 131, 'transmission': 70, 'size': 8079, 'type': 2496, 'paint_color': 3604, 'county': 12349, 'state': 0, 'lat': 139, 'long': 139, 'posting_date': 0)

[10] from pyspark.sql.functions import col, sum, is_nan, when, isnan
# Get the total number of rows in the dataframe
total_rows = df.count()
# Calculate the percentage of nulls and nans in each column
null_percentage = (
    df.select(
        _sumwhen(col(c).isNull() | isnan(col(c)), 1).otherwise(0)).alias(c)
        for c in df.columns
    )
)
# Collect the percentages into a dictionary for evaluation
null_percentage_dict = null_percentage.collect()[0].asDict()
# Identify columns to drop (null percentage > 40%)
columns_to_drop = [col for col, perc in null_percentage_dict.items() if perc > 4.0]
# Drop the identified columns
df_cleaned = df.drop(*columns_to_drop)
# Log the dropped columns and cleaned dataframe schema
log_message(message=f"Columns dropped: {columns_to_drop}")
log_message(message=df_cleaned.printSchema()) # Log the schema directly without embedding in log_message

INFO:root:Columns dropped: ('condition', 'cylinders', 'size', 'county')
root
|-- price: long (nullable = true)
|-- year: double (nullable = true)
|-- manufacturer: string (nullable = true)
|-- model: string (nullable = true)
|-- odometer: double (nullable = true)
|-- transmission: string (nullable = true)
|-- type: string (nullable = true)
```

Fig 8 – code to check the nulls in all the columns.

```

bda_13_12_2024.ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text
[11] # set the shape of the dataframe after dropping the columns
num_rows = df.count()
num_columns = len(df.columns)
log_message(message="Shape of the dataset after dropping columns: ({num_rows}, {num_columns})")

INFO:root:Shape of the dataset after dropping columns: (12349, 14)

[12] df_cleaned = df_cleaned.dropna()
# set the shape of the dataframe after dropping the columns
num_rows = df_cleaned.count()
num_columns = len(df_cleaned.columns)
log_message(message="Shape of the dataset after dropping columns: ({num_rows}, {num_columns})")

INFO:root:Shape of the dataset after dropping columns: (7244, 12)

[13] from pyspark.sql.functions import col
# cast 'price' and 'odometer' to float
columns_to_cast_float = ["price", "odometer"]
for column in columns_to_cast_float:
    df_cleaned = df_cleaned.withColumn(column, col(column).cast("float"))
# extract numeric part from 'cylinders' and cast 'year' and 'cylinders' to integer
df_cleaned = df_cleaned.withColumn("year", col("year").cast("int"))
log_message(message="df_cleaned.printSchema()")
log_message(message="df_cleaned.printSchema()")

INFO:root:Top 5 Rows of the Dataframe after type casting ==
root
|-- price: float (nullable = true)
|-- year: integer (nullable = true)
|-- manufacturer: string (nullable = true)
|-- model: string (nullable = true)
|-- odometer: float (nullable = true)
|-- transmission: string (nullable = true)
|-- type: string (nullable = true)
|-- paint_color: string (nullable = true)
|-- state: string (nullable = true)
|-- lat: double (nullable = true)
|-- long: double (nullable = true)
|-- posting_date: string (nullable = true)

INFO:root:
price | year | manufacturer | model | odometer | transmission | type | paint_color | state | lat | long | posting_date |
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
7990 | 2012 | nissan | juke | 120956 | automatic | SUV | white | FL | 28.827 | -82.4999 | 2021-04-23T14:05:13-0400 |
15900 | 2013 | subaru | impreza wrx | 100724 | manual | wagon | silver | WA | 47.6929 | -117.432 | 2021-04-04T18:00:56-0700 |
28900 | 2012 | ford | f-150 | 93761 | automatic | truck | black | WA | 47.6929 | -117.432 | 2021-04-15T19:11:18-0700 |
4900 | 2009 | gmc | sierra | 256000 | automatic | truck | red | OH | 39.2353 | -84.4619 | 2021-04-20T19:07:51-0400 |
15400 | 2010 | ram | 1500 big horn | 110290 | automatic | truck | black | FL | 41.8221 | -71.3539 | 2021-04-23T11:15:12-0400 |

[14] from pyspark.sql.functions import col, udf
from pyspark.sql.types import StringType
from geopy.geocoders import Nominatim

# Define the udf to get ZIP code
@udf(StringType())
def get_zip_code(lat, long):
    if lat is None or long is None: # Check for NoneType values
        return None
    try:
        geolocator = Nominatim(user_agent="geopy")
        location = geolocator.reverse(f"{lat},{long}", timeout=10)
        if location and "postcode" in location.raw["address"]:
            return location.raw["address"].get("postcode")
        return None
    except Exception:
        return None

# Filter rows where lat or long is null (optional but recommended)
df_cleaned = df_cleaned.filter((col("lat").isNotNull()) & (col("long").isNotNull()))
# Apply the udf to add the zip_code column
df_cleaned = df_cleaned.withColumn("zip_code", get_zip_code(col("lat"), col("long")))
df_cleaned = df_cleaned.withColumn("zip_code", col("zip_code").cast("int"))
# Drop the lat and long columns
df_cleaned = df_cleaned.drop("lat", "long")
# Show the resulting dataframe
log_message(message="df_cleaned.select('zip_code').", title="Top 5 Rows extracting the zipcode from lat long - FEATURE ENGINEERING")

INFO:root:Top 5 Rows extracting the zipcode from lat long - FEATURE ENGINEERING ==
INFO:root:
zip_code |
-----|
5206 |
90307 |
90307 |
45215 |
2014 |

[15] from pyspark.sql import functions as f
# concatenate 'manufacturer' and 'model' into a new column 'car_name'
df_cleaned = df_cleaned.withColumn(
    "car_name", f.concat(f.col("manufacturer"), f.lit(" "), f.col("model"))
)
df_cleaned = df_cleaned.drop("model")
# Show the resulting dataframe with the new 'car_name' column
log_message(message="df_cleaned.select('car_name').", title="Top 5 Rows with car_name column")

INFO:root:Top 5 Rows with car_name column ==
INFO:root:
car_name |
-----|

```

Fig 9 – rows with nulls dropped and data type casting performed

```

bda_13_12_2024.ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text
[14] from pyspark.sql.functions import col, udf
from pyspark.sql.types import StringType
from geopy.geocoders import Nominatim

# Define the udf to get ZIP code
@udf(StringType())
def get_zip_code(lat, long):
    if lat is None or long is None: # Check for NoneType values
        return None
    try:
        geolocator = Nominatim(user_agent="geopy")
        location = geolocator.reverse(f"{lat},{long}", timeout=10)
        if location and "postcode" in location.raw["address"]:
            return location.raw["address"].get("postcode")
        return None
    except Exception:
        return None

# Filter rows where lat or long is null (optional but recommended)
df_cleaned = df_cleaned.filter((col("lat").isNotNull()) & (col("long").isNotNull()))
# Apply the udf to add the zip_code column
df_cleaned = df_cleaned.withColumn("zip_code", get_zip_code(col("lat"), col("long")))
df_cleaned = df_cleaned.withColumn("zip_code", col("zip_code").cast("int"))
# Drop the lat and long columns
df_cleaned = df_cleaned.drop("lat", "long")
# Show the resulting dataframe
log_message(message="df_cleaned.select('zip_code').", title="Top 5 Rows extracting the zipcode from lat long - FEATURE ENGINEERING")

INFO:root:Top 5 Rows extracting the zipcode from lat long - FEATURE ENGINEERING ==
INFO:root:
zip_code |
-----|
5206 |
90307 |
90307 |
45215 |
2014 |

[15] from pyspark.sql import functions as f
# concatenate 'manufacturer' and 'model' into a new column 'car_name'
df_cleaned = df_cleaned.withColumn(
    "car_name", f.concat(f.col("manufacturer"), f.lit(" "), f.col("model"))
)
df_cleaned = df_cleaned.drop("model")
# Show the resulting dataframe with the new 'car_name' column
log_message(message="df_cleaned.select('car_name').", title="Top 5 Rows with car_name column")

INFO:root:Top 5 Rows with car_name column ==
INFO:root:
car_name |
-----|

```

Fig 10 – Feature engineering – using lat and long along with geopy, extracted the zip codes.

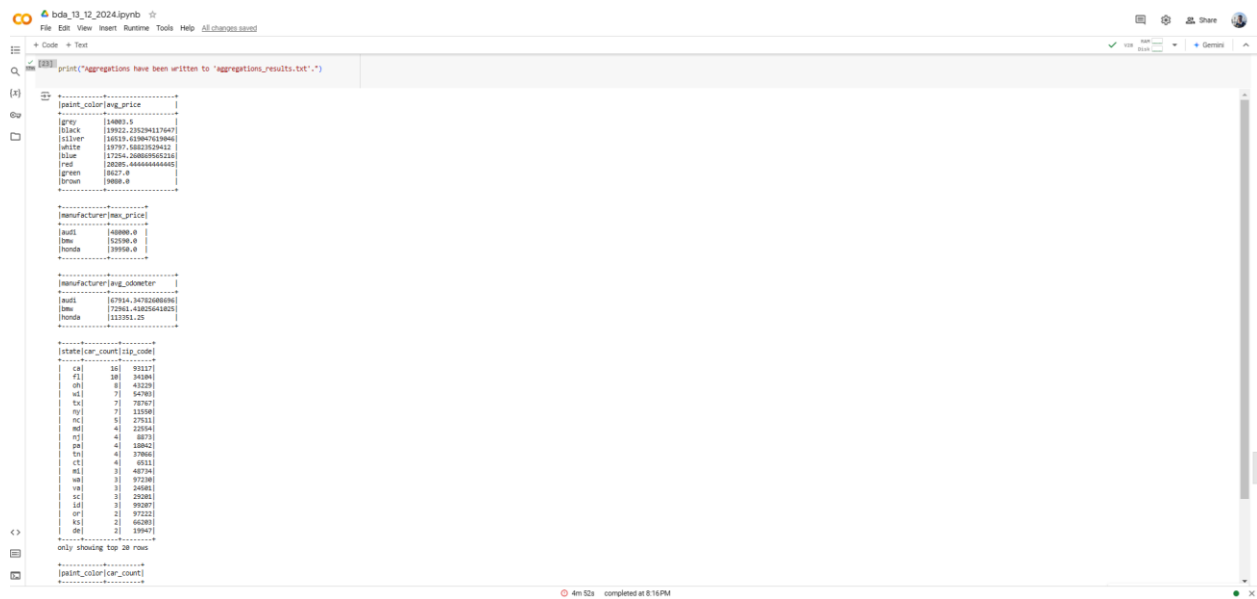


Fig 15 – Output of the data aggregation.

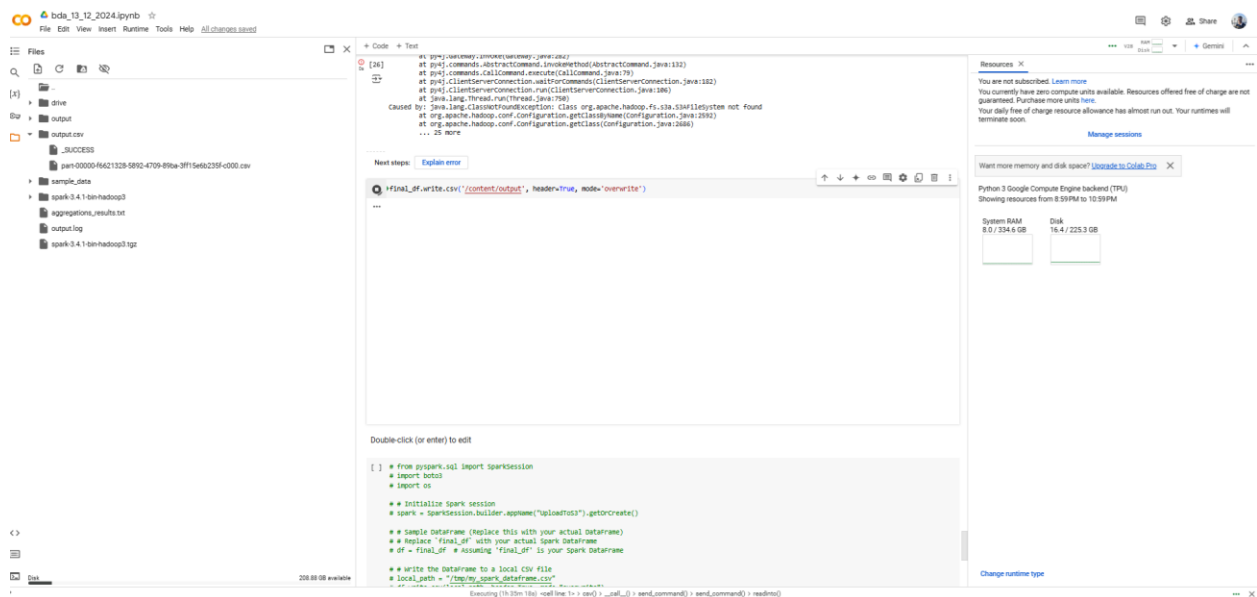


Fig 16 – Dumping the cleaned dataset to a csv file as seen on the right side [part file]

```
# prompt: df.show(), instead write to a new csv file, make sure no data is lost
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, log, exp, when
from pyspark.sql.functions import skewness

# Initialize SparkSession
spark = SparkSession.builder.appName("SkewnessHandling").getOrCreate()

# Load the CSV file into a Spark DataFrame
df = spark.read.csv("content/part-00000-f6a21328-5892-4709-899a-3ff15e6235f-c000.csv", header=True, inferSchema=True)

# Specify the numeric columns that need skewness and outlier handling
numeric_inputs = ["odometer", "car_name_sum"]

# Create a dictionary to store the quantiles (1st and 99th percentiles)
d = {}

# Calculate the 1st and 99th percentiles for each column
for col_name in numeric_inputs:
    d[col_name] = df.approxQuantile(col_name, [0.01, 0.99], 0.25)

# Handle skewness and outliers
for col_name in numeric_inputs:
    # Calculate skewness for each numeric column
    skew = df.agg(skewness(col_name)).collect()[0][0]

    # Clip values to handle outliers (below 1st quantile and above 99th quantile)
    clipped_col = when(df[col_name] < d[col_name][0], d[col_name][0]) \
        .when(df[col_name] > d[col_name][1], d[col_name][1]) \
        .otherwise(df[col_name])

    # If skewness is positive (right skew), apply log transformation
    if skew > 1:
        df = df.withColumn(col_name, log(clipped_col + 1).alias(col_name)) # Apply log transformation
        print(f"{col_name} has been treated for positive (right) skewness. (skew = {skew})")

    # If skewness is negative (left skew), apply exponential transformation
    elif skew < -1:
        df = df.withColumn(col_name, exp(clipped_col).alias(col_name)) # Apply exponential transformation
        print(f"{col_name} has been treated for negative (left) skewness. (skew = {skew})")

df.show()

# Write the transformed DataFrame to a new CSV file
df.write.csv("content/transformed_data.csv", header=True, mode="overwrite")
print("transformed data saved to transformed_data.csv")

# Stop the SparkSession
spark.stop()

odometer has been treated for positive (right) skewness. (skew = 41.8593084132445)
car_name_sum has been treated for positive (right) skewness. (skew = 3.882396236388465)
| price|year|manufacturer|odometer|transmission|type|paint_color|state|posting_date|zip_code|transmission_indexed|paint_color_indexed|state_indexed|manufacturer_indexed|car_name_sum|
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 7996.0|2012|Hyundai|11.695723089794|automatic|SUV|white|TX|2012-04-23 12:45:23|33064|0.0|0.0|1.0|4.0|1.895622286623861
```

Fig 16 – Reused the preprocessed csv file to further process and remove skewness by applying log and exp transform

Pyspark SQL queries

```
# Step 2: Read CSV file
file_path = "content/part-00000-f6a21328-5892-4709-899a-3ff15e6235f-c000.csv"
df = spark.read.csv(file_path, header=True, inferSchema=True)

# Step 3: Create a temp table
df.createOrReplaceTempTable("car_data")

# Step 4: Define SQL queries (excluding rows where zip_code is NULL)
queries = [
    "Query 1: Top 5 Regions with the Highest Average Price:",
    "SELECT zip_code, ROUND(AVG(price), 2) AS avg_price",
    "FROM car_data",
    "WHERE zip_code IS NOT NULL",
    "GROUP BY zip_code",
    "ORDER BY avg_price DESC",
    "LIMIT 5",
    "",
    "Query 2: Total Car Sales Revenue by Year:",
    "SELECT year, SUM(price) AS total_revenue",
    "FROM car_data",
    "WHERE zip_code IS NOT NULL",
    "GROUP BY year",
    "ORDER BY year ASC",
    "",
    "Query 3: Count of Cars by Transmission Type per Region:",
    "SELECT zip_code, transmission, COUNT(*) AS car_count",
    "FROM car_data",
    "WHERE zip_code IS NOT NULL",
    "GROUP BY zip_code, transmission",
    "ORDER BY zip_code",
    "",
    "Query 4: Monthly trend in car listings:",
    "SELECT SUBSTRING(posting_date, 1, 7) AS month, COUNT(*) AS car_listings",
    "FROM car_data",
    "WHERE zip_code IS NOT NULL",
    "GROUP BY SUBSTRING(posting_date, 1, 7)",
    "ORDER BY month ASC",
    "",
    "Query 5: Top 3 Car Types with the Highest Total Sales in each region:",
    "SELECT zip_code, type, SUM(price) AS total_sales",
    "FROM car_data",
    "WHERE zip_code IS NOT NULL",
    "GROUP BY zip_code, type",
    "ORDER BY zip_code, total_sales DESC",
    "LIMIT 3"
]

# Replace with the correct file path
```

Fig 1 – Ingested the clean dataset csv file into spark dataframe and performed data analysis using spark SQL.

AWS SageMaker Autopilot

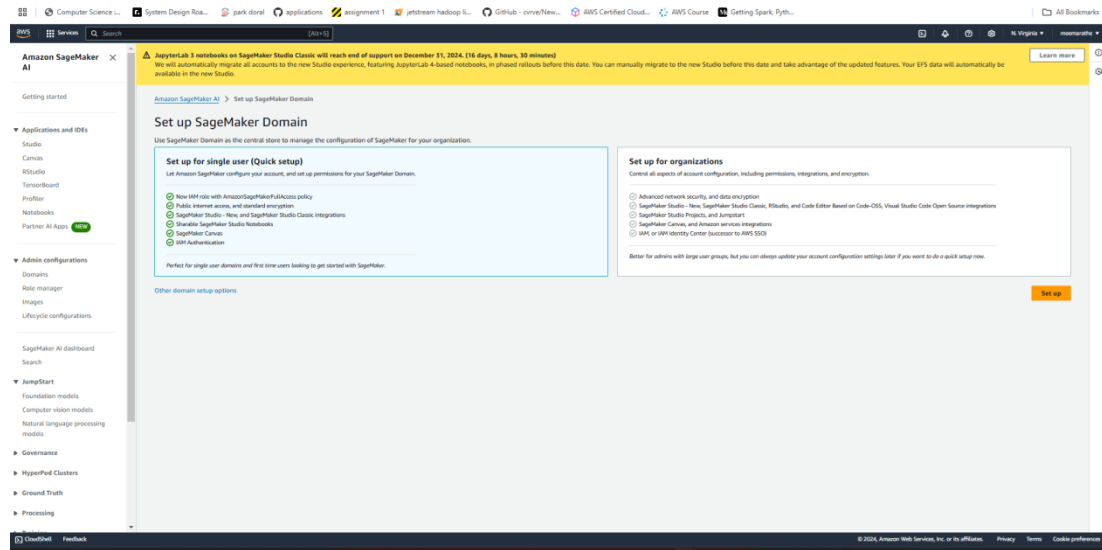


Fig 1 – Domain creation for Sage maker – up and running

We use this domain to launch the Studio where we can run the experiment Autopilot – AutoML for modeling and evaluation.

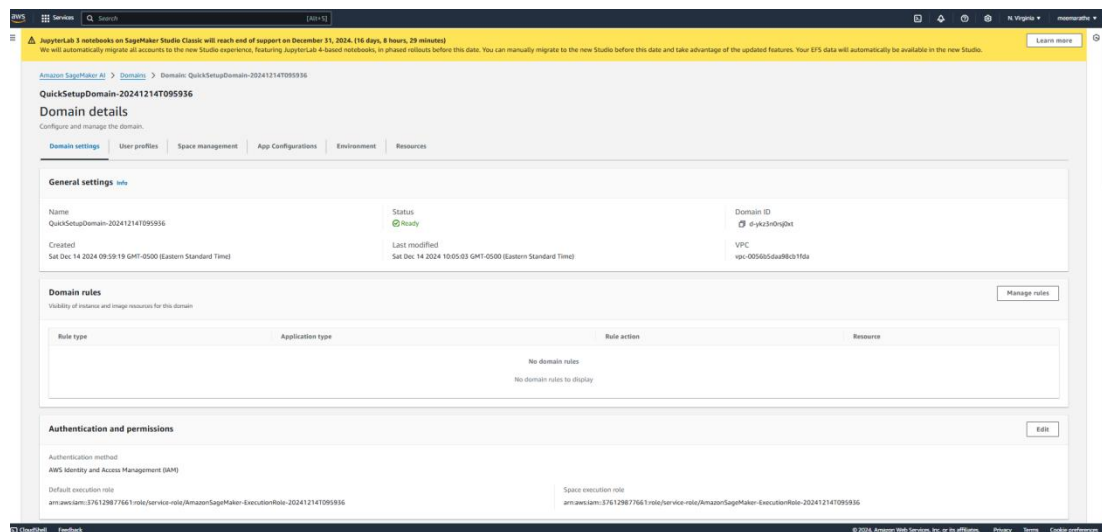


Fig 2 – Sagemaker domain setup success.

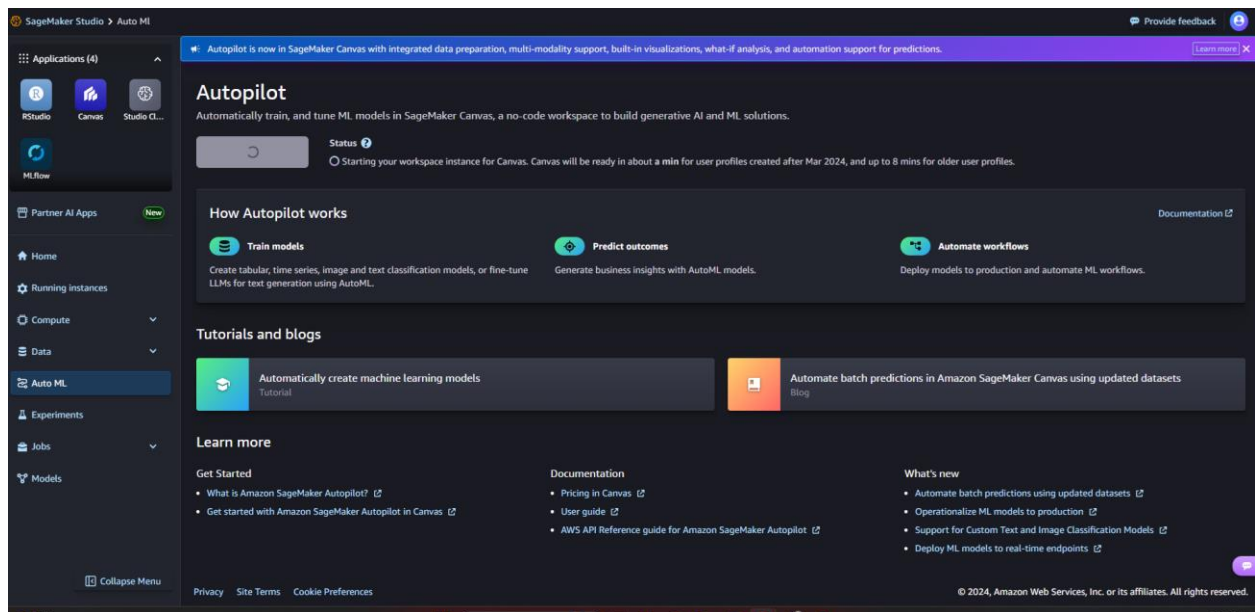


Fig 3 – AWS Sage maker Autopilot running to launch the canvas.

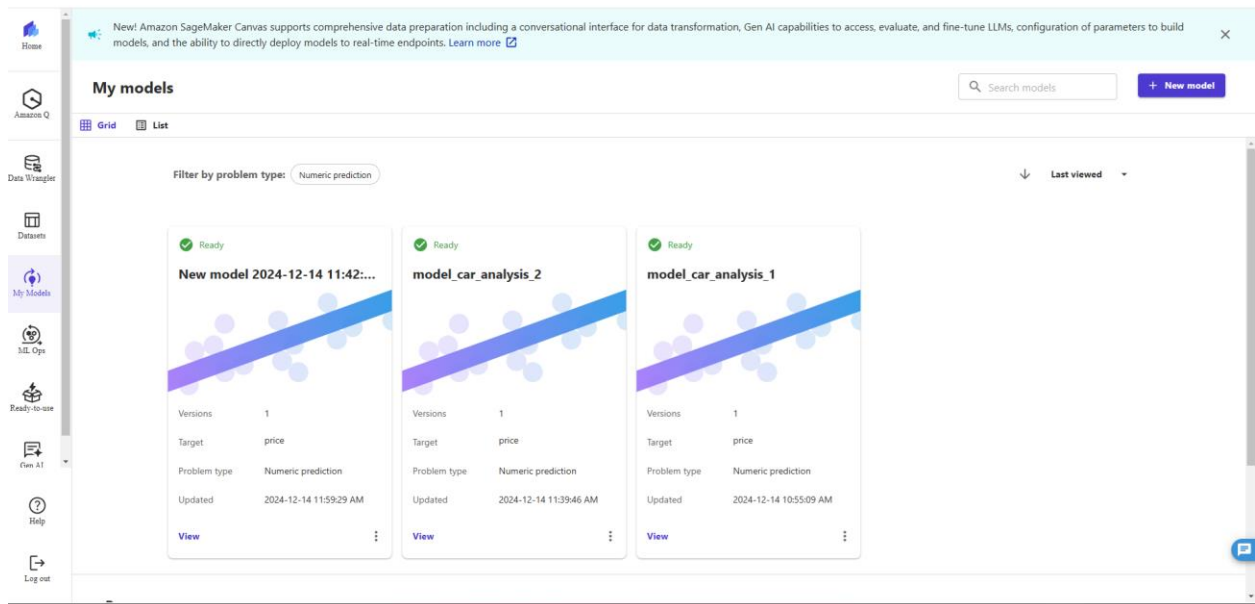
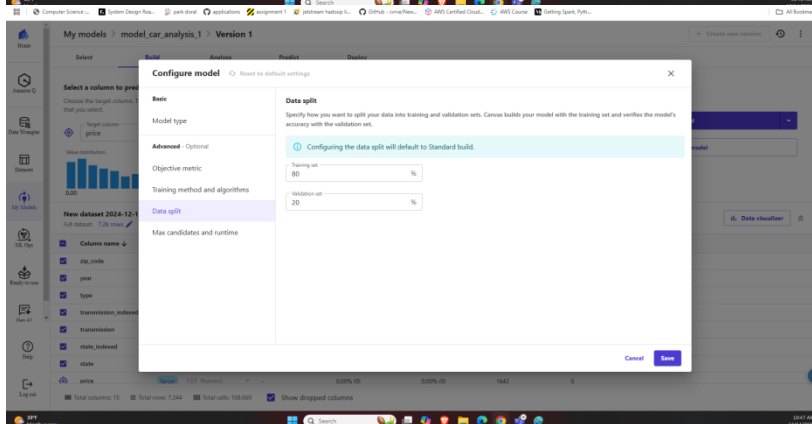
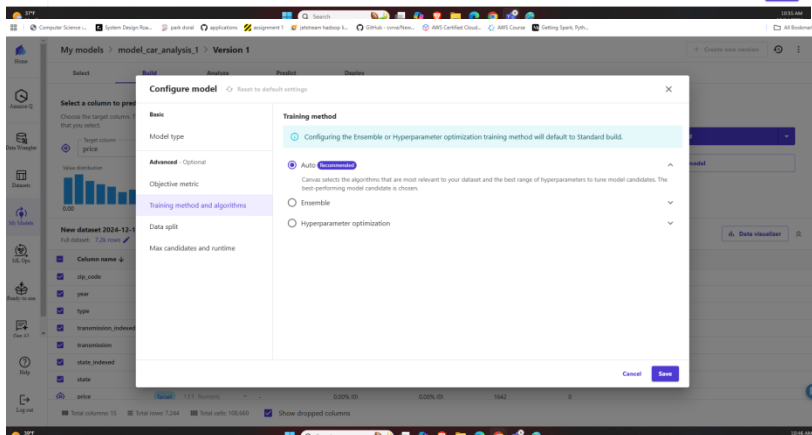
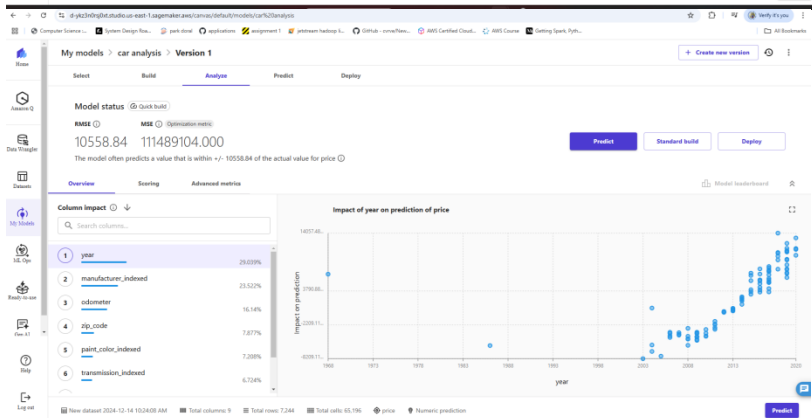


Fig 4 – Created and experimented 3 models [1. With only numeric cols, 2. Raw set with selecting transform operation in AutoML 3. Tree based models on cleaned Dataset]



- The prediction accuracy appears to decrease as prices increase, shown by the widening purple band
- The model shows better performance in the lower price range (0-30,000) with tighter clustering around the diagonal line

The width of the RMSE band indicates moderate model performance.

AWS QuickSight



Fig 1- creation of the dashboard and data plots.

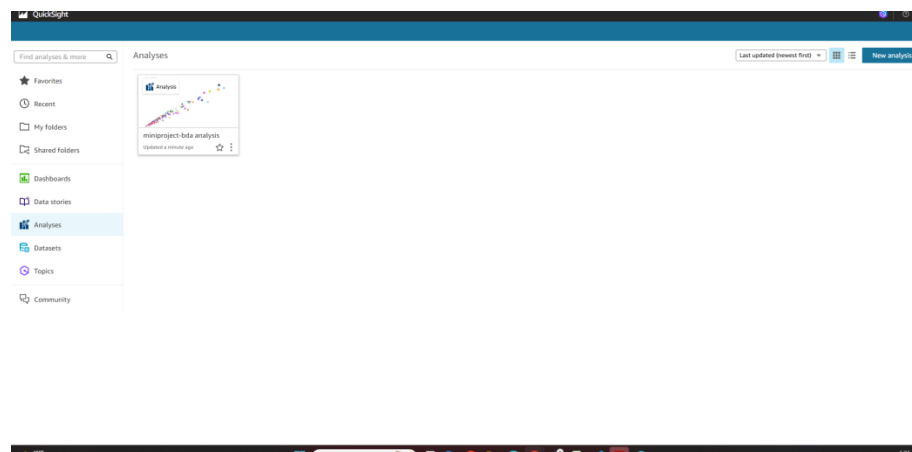


Fig 2 – Landing page of AWS QuickSight, Dashboard for analysis is displayed.

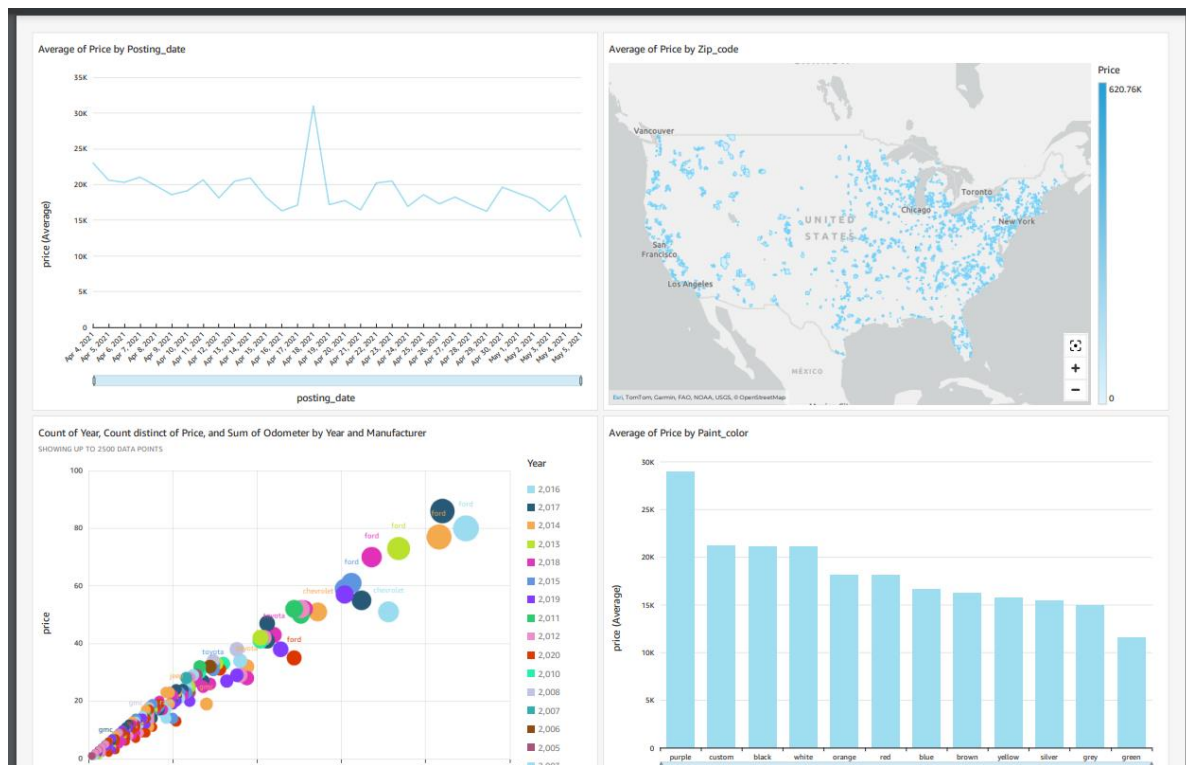


Fig 3 – The dashboard displaying trends with respect to the used car prices.

1. **Price Trend Over Time:** The line graph shows average prices fluctuating between \$15,000-\$20,000 throughout the posting period, with a notable spike reaching approximately \$30,000 at one point..
2. **Geographic Distribution:** The map visualization reveals a higher concentration of car listings in the eastern United States, particularly along the coast. The pricing intensity (shown in blue) varies across regions, with some areas showing higher average prices than others.
3. **Manufacturer Analysis:** The bubble chart shows a positive correlation between year and price across different manufacturers. More recent years (2015-2017) tend to have larger bubbles, indicating higher prices. Ford appears to be one of the dominant manufacturers.
4. **Paint Color Impact:** The bar chart indicates that purple cars command the highest average price (around \$28,000), followed by custom colors. Green vehicles show the lowest average price (approximately \$12,000). Most common colors like black, white, and silver fall in the middle range between \$15,000-\$20,000

Summary of Issues and Resolutions for S3 setup and reading data

1. S3 Bucket Access (400 Error)

- **Issue:** Incorrect AWS credentials caused a Bad Request error.
- **Resolution:** Updated and verified AWS credentials, ensuring compatibility with PySpark.

2. Bucket Permissions

- **Issue:** Public access was blocked, and the EC2 instance lacked sufficient permissions.
- **Resolution:** Updated the S3 bucket policy and IAM role, granting the required permissions.

3. PySpark Configuration on EC2 (t2.micro)

- **Issue:** Spark configurations exceeded the resource limitations of the t2.micro instance.
- **Resolution:** Adjusted Spark memory and partition settings to fit the instance's capacity.

4. Large Dataset Processing

- **Issue:** EC2's limited resources caused hangs during Spark transformations on a large dataset.
- **Resolution:** Migrated preprocessing to Google Colab, which handled the dataset effectively.

5. Downloading Preprocessed Data

- **Issue:** Version conflicts with Hadoop, Python, and PySpark hindered CSV file downloads.
- **Resolution:** Installed compatible versions of dependencies and successfully saved data to S3.

6. Accessing S3 Dataset: Encountered issues with AWS credentials and permissions while accessing the S3 bucket. Resolved by ensuring IAM roles had s3:GetObject permission and setting credentials correctly in Spark configuration.

7. Configuring Spark Session: Faced errors with Spark session initialization due to syntax mistakes and missing dependencies. Corrected the code structure and added required JAR packages for S3 access.

8. Feature Engineering with Geo-Data: Needed to map latitude and longitude to postal codes for analysis. Addressed this by integrating geocoding libraries like Geopy for Python to convert coordinates to ZIP codes.

Conclusion

The project implements a comprehensive data pipeline for analyzing and predicting used car prices using AWS services. The pipeline begins with raw data stored in S3, which is then processed using Jupyter Notebooks for cleaning and feature engineering. The preprocessed data is saved back to S3 and used for SQL queries via Python scripts and machine learning model training through SageMaker AutoML. Finally, AWS QuickSight is utilized to create visualizations and deliver insights to end users. The resulting analysis reveals several key insights about the used car market. The average price of cars fluctuates over time, with a notable spike observed in the time series. Geographic distribution of listings shows a concentration in the eastern United States. The data also indicates a correlation between car age and price across different manufacturers, with newer models generally commanding higher prices. Additionally, the analysis reveals that car color impacts pricing, with purple and custom-colored vehicles fetching higher average prices compared to more common colors. The machine learning model developed for price prediction achieves an RMSE of 10558.84, indicating that predictions are typically within this range of the actual prices.

References:

1. AWS Services Documentation

- Amazon S3: <https://docs.aws.amazon.com/s3/>
- Amazon SageMaker AutoML:
 - <https://docs.aws.amazon.com/sagemaker/latest/dg/autopilot-automate-model-development.html>
 - <https://www.youtube.com/watch?v=Atak2tU1iHY>
- AWS QuickSight:
 - <https://docs.aws.amazon.com/quicksight/>
 - <https://www.youtube.com/watch?v=rxyLC247h6E>
- Amazon Athena: <https://docs.aws.amazon.com/athena/>
- Jupyter Notebooks on EC2: <https://docs.aws.amazon.com/dlami/latest/devguide/setup-jupyter.html>

2. GitHub Repository

- AWS Sample Notebooks: <https://github.com/aws/amazon-sagemaker-examples>

3. AWS Case Studies

- AWS Machine Learning Case Studies: <https://aws.amazon.com/machine-learning/customers/>