PROJECT PHASE -3
A Predictive Assessment System for Diabetes Risk Factors

Mounika Pasupuleti
Engineering Science Data Science
University at Buffalo
Buffalo, New York
mpasupul@buffalo.edu

Sahithya Arveti Nagaraju
Engineering Science Data Science
University at Buffalo
Buffalo, New York
sarvetin@buffalo.edu

Sushmitha Manjunatha
Engineering Science Data Science
University at Buffalo
Buffalo, New York
smanjuna@buffalo.edu

***Abstract*—Diabetes has rapidly transformed from a relatively uncommon disease among a small number of people to a serious disease of global proportions affecting every age group. Diabetes Pathophysiology may be defined as a disease where the sugar levels in the blood are not well regulated leading to substantially deteriorated health and shortened life expectancy. In many cases treatment of diabetes comes with high costs which is an additional burden on the affected individual.**

The goal of this project is to investigate the most common correlation with diabetes, factors which help in understanding possible causative agents and their potential risk. This analysis is expected to aid in prevention of diabetes cases and promote healthy behavior.

In this regard, we expect to evaluate how these basic attributes such as work performance, nutrition, and health history could be used to estimate the probability of an individual being at risk of diabetes by employing complex machine learning models and analyzing data from health activities. This contribution is crucial as it addresses the growing health challenges like diabetes by providing valuable insights that enhances overall community health.

Index Terms: Diabetes, sugar levels, lifestyle, machine learning models.

I. DATA SOURCES

The dataset is taken from the Kaggle website. Below is the link:

https://www.kaggle.com/datasets/prosperchuks/health-dataset?select=diabetes_data.csv

The dataset consists of 70,707 rows and 19 columns in which there are 10 numeric features and 9 categorical features. Numeric features in our dataset are:

- Age
- HighChol
- CholCheck
- BMI
- HeartDiseaseorAttack
- PhysActivity
- HvyAlcoholConsump
- MentHlth
- PhyHlth
- HighBP

Categorical features in our dataset are:

- Sex
- Smoker
- Fruits
- Veggies
- GenHlth
- DiffWalk
- Stroke
- SugarConsumption
- Diabetes

Observe the data in the dataset:

```
[2] import pandas as pd
    import numpy as np
    data = pd.read_csv("impure_dataset_final.csv");
    data.head(5)
```

	Age	Sex	HighChol	CholCheck	BMI	Smoker	HeartDiseaseorAttack	PhysActivity	Fruits	Ve
0	4	Male	0	1	26	Non Smoker	0	1.0	Does not Eat	
1	12	Male	1	1	26	Smoker	0	0.0	Eat	
2	13	Male	0	1	26	Non Smoker	0	1.0	Eat	
3	11	Male	1	1	28	Smoker	0	1.0	Eat	
4	8	Female	0	1	29	Smoker	0	1.0	Eat	

```
[ ] data.shape
```

```
(70707, 19)
```

```
[ ] dataset_info = {
        "Number of rows in the dataset are": data.shape[0],
        "Number of columns in the dataset are ": data.shape[1],
        "Names of the columns": data.columns.tolist()
    }

    dataset_info
```

```
{'Number of rows in the dataset are': 70707,
 'Number of columns in the dataset are ': 19,
 'Names of the columns': ['Age',
  'Sex',
  'HighChol',
  'CholCheck',
  'BMI',
  'Smoker',
  'HeartDiseaseorAttack',
  'PhysActivity',
```

```
 'HvyAlcoholConsump',
 'GenHlth',
 'MentHlth',
 'PhysHlth',
 'DiffWalk',
 'Stroke',
 'HighBP',
 'SugarConsumption',
 'Diabetes']}
```

```
[ ] data.describe()
```

	Age	HighChol	CholCheck	HeartDiseaseorAttack	PhysActivity	HvyAlcol
count	70707.000000	70707.000000	70707.000000	70707.000000	70511.000000	69
mean	8.581116	0.525747	0.975264	0.147878	0.703124	
std	2.864696	0.499340	0.155320	0.354981	0.456885	
min	-31.000000	0.000000	0.000000	0.000000	0.000000	
25%	7.000000	0.000000	1.000000	0.000000	0.000000	
50%	9.000000	1.000000	1.000000	0.000000	1.000000	
75%	11.000000	1.000000	1.000000	0.000000	1.000000	
max	13.000000	1.000000	1.000000	1.000000	1.000000	

II. DATA CLEANING/PROCESSING

Uncleaned data contains duplicate entries or non-unique records, missing values and inconsistencies in the data. This data needs to be thoroughly cleaned to remove such issues and make the dataset fit for analysis.

A. Removing Duplicate Rows:

```
# Step 1: Counting duplicate rows
# Counting the number of rows before removing duplicates
num_rows_before = data.count()

# Counting the number of unique rows (after dropping duplicates)
num_unique_rows = data.dropDuplicates().count()

# Calculating the number of duplicate rows
num_duplicates = num_rows_before - num_unique_rows

# Removing duplicate rows
data = data.dropDuplicates()

# Counting the number of rows after removing duplicates
num_rows_after = data.count()

# Printing the results
print(f"Number of duplicated rows in the dataset: {num_duplicates}")
print(f"Number of rows before removing duplicates: {num_rows_before}")
print(f"Number of rows after removing duplicates: {num_rows_after}")
```

```
Number of duplicated rows in the dataset: 5554
Number of rows before removing duplicates: 70707
Number of rows after removing duplicates: 65153
```

Using data.duplicated() method, we found 5,554 records having duplicated data. These duplicates were deleted and reduced the dataset from 70,707 rows to 65,153 rows.

B. Removing Missing Values:

Initially, the dataset contained missing values across several columns, including “BMI”, “Smoker”, “PhysActivity”, “Fruits”, “Veggies,” and others. Subsequently, columns with fewer than 300 missing values were identified and handled. Rows with missing data in these columns were removed, reducing the dataset to 64,286 rows. The remaining missing values in categorical and numerical columns were replaced using mode and mean, respectively.

```
# Step 2: Remove missing values
#Calculate the number of non-null values in each column
non_null_counts = data.select([F.sum(F.when(F.col(column).isNotNull(), 1).otherwise(0)).alias(column)
                               for column in data.columns])

# Showing the non-null counts per column
non_null_counts.show()

# Calculating the number of null values in each column
null_counts = data.select([F.sum(F.when(F.col(column).isNull(), 1).otherwise(0)).alias(column)
                           for column in data.columns])

# Showing the null counts per column
null_counts.show()
```

```
+-----+-----+--------+---------+-----+------+--------------------+------------+------+-------+-----------------+-------+---
-----+--------+--------+------+------+----------------+--------+
|  Age|  Sex|HighChol|CholCheck|  BMI|Smoker|HeartDiseaseorAttack|PhysActivity|Fruits|Veggies|HvyAlcoholConsump|GenHlth|Men
tHlth|PhysHlth|DiffWalk|Stroke|HighBP|SugarConsumption|Diabetes|
+-----+-----+--------+---------+-----+------+--------------------+------------+------+-------+-----------------+-------+---
-----+--------+--------+------+------+----------------+--------+
|65153|65153|   65153|    65153|64166| 64140|               65153|       64957| 64921|  64938|            64221|  64164|
64206|   64189|   64926| 65153| 65153|           65153|   65153|
+-----+-----+--------+---------+-----+------+--------------------+------------+------+-------+-----------------+-------+---
-----+--------+--------+------+------+----------------+--------+

+---+---+--------+---------+---+------+--------------------+------------+------+-------+-----------------+-------+--------+
--------+--------+------+------+----------------+--------+
|Age|Sex|HighChol|CholCheck|BMI|Smoker|HeartDiseaseorAttack|PhysActivity|Fruits|Veggies|HvyAlcoholConsump|GenHlth|MentHlth|
PhysHlth|DiffWalk|Stroke|HighBP|SugarConsumption|Diabetes|
+---+---+--------+---------+---+------+--------------------+------------+------+-------+-----------------+-------+--------+
--------+--------+------+------+----------------+--------+
|  0|  0|       0|        0|987|  1013|                   0|         196|   232|    215|              932|    989|     947|
964|     227|     0|     0|               0|       0|
+---+---+--------+---------+---+------+--------------------+------------+------+-------+-----------------+-------+--------+
--------+--------+------+------+----------------+--------+
```

```
# Calculateing the number of null values for each column
null_counts = data.select([F.sum(F.when(F.col(column).isNull(), 1).otherwise(0)).alias(column)
                           for column in data.columns])

# Collecting the null counts and find columns with fewer than 300 null values
columns_with_less_than_300_nulls = [column for column in null_counts.columns if null_counts.select(column).first()[0] < 300]

# Dropping rows that have missing values in those columns
data_cleaned = data.dropna(subset=columns_with_less_than_300_nulls)

# Printing the number of rows after removing
print(f"Number of rows after removing rows with missing values in columns with fewer than 300 missing values: {data_cleaned.
```

```
Number of rows after removing rows with missing values in columns with fewer than 300 missing values: 64286
```

C. Remove inconsistencies:

```
# Step 3: Remove inconsistencies
for column in data.columns:
    # Filtering out 'null' values and then get the distinct unique values
    unique_items = data.filter(F.col(column).isNotNull()).select(column).distinct().collect()

    # Converting the result to a list of unique values
    unique_items_list = [row[0] for row in unique_items]
    print(f"Unique items in '{column}':", unique_items_list)
```

```
Unique items in 'Age': [-1, -21, 12, -15, 1, 13, 6, 3, -9, -7, -31, 5, -19, 9, 4, 8, 7, 10, -2, -12, 11, 2, -3]
Unique items in 'Sex': ['F', 'Female', 'M', 'Male']
Unique items in 'HighChol': [1, 0]
Unique items in 'CholCheck': [1, 0]
Unique items in 'BMI': ['51', '24.0 kg/m2', '15', '54', '29', '69', '42', '73', '87', '64', '30', '34', '59', '28', '22',
'85', '35', '52', '16', '71', '98', '47', '43', '31', '18', '70', '27', '61', '75', '31.0 kg/m2', '17', '26', '46', '77',
'89', '68', '60', '19', '41', '23', '55', '95', '38', '40', '38.0 kg/m2', '25', '44', '82', '53', '92', '86', '58', '33.0 k
g/m2', '81', '33', '48', '67', '84', '79', '24', '32', '29.0 kg/m2', '20', '56', '36', '37', '49', '63', '65', '39', '25.0
kg/m2', '62', '12', '83', '13', '21', '14', '66', '72', '28.0 kg/m2', '76', '80', '50', '45', '57', '34.0 kg/m2', '21.0 kg/
m2', '78', '26.0 kg/m2', '74']
Unique items in 'Smoker': ['Smoker', 'Non Smoker']
Unique items in 'HeartDiseaseorAttack': [1, 0]
Unique items in 'PhysActivity': [1, 0]
Unique items in 'Fruits': ['Eat', 'Does not Eat', 'No', 'Yes']
Unique items in 'Veggies': ['Eat', 'Does not Eat', 'No', 'Yes']
Unique items in 'HvyAlcoholConsump': [1, 0]
Unique items in 'GenHlth': ['very good', 'poor', 'excellent', 'good', 'fair']
Unique items in 'MentHlth': [28, 26, 27, 12, 22, 1, 13, 6, 16, 3, 20, 5, 19, 15, 9, 17, 4, 8, 23, 7, 10, 25, 24, 29, 21, 1
1, 14, 2, 30, 0, 18]
Unique items in 'PhysHlth': [28, 26, 27, 12, 22, 1, 13, 6, 16, 3, 20, 5, 19, 15, 17, 9, 4, 8, 23, 7, 10, 25, 24, 29, 21, 1
1, 14, 2, 30, 0, 18]
Unique items in 'DiffWalk': ['Y', 'N', 'No', 'Yes']
Unique items in 'Stroke': ['Y', 'N', 'No', 'Yes']
Unique items in 'HighBP': [1, 0]
Unique items in 'SugarConsumption': ['High', 'Low']
Unique items in 'Diabetes': ['No', 'Yes']
```

```
# Replacing values in columns using PySpark
data = data.withColumn('Sex', F.when(F.col('Sex') == 'Male', 'M')
                       .when(F.col('Sex') == 'Female', 'F')
                       .otherwise(F.col('Sex')))

data = data.withColumn('Fruits', F.when(F.col('Fruits') == 'Yes', 'Eat')
                        .when(F.col('Fruits') == 'No', 'Does not Eat')
                        .otherwise(F.col('Fruits')))

data = data.withColumn('Veggies', F.when(F.col('Veggies') == 'Yes', 'Eat')
                        .when(F.col('Veggies') == 'No', 'Does not Eat')
                        .otherwise(F.col('Veggies')))

data = data.withColumn('DiffWalk', F.when(F.col('DiffWalk') == 'Yes', 'Y')
                       .when(F.col('DiffWalk') == 'No', 'N')
                       .otherwise(F.col('DiffWalk')))

data = data.withColumn('Stroke', F.when(F.col('Stroke') == 'Yes', 'Y')
                      .when(F.col('Stroke') == 'No', 'N')
                      .otherwise(F.col('Stroke')))

data = data.withColumn('Diabetes', F.when(F.col('Diabetes') == 'Yes', 'Y')
                       .when(F.col('Diabetes') == 'No', 'N')
                       .otherwise(F.col('Diabetes')))

# Converting 'BMI' column to float after removing ' kg/m2'
data = data.withColumn('BMI', F.regexp_replace('BMI', ' kg/m2', '').cast('float'))


    # Looping through columns to get unique values in each column using distinct
for column in data.columns:
    # Replacing 'None' or null values with 'Not Specified' before getting distinct values
    unique_items = data.filter(F.col(column).isNotNull()).select(column).distinct().collect()

    # Collecting and convert to list
    unique_items_list = [row[0] if row[0] is not None else 'Not Specified' for row in unique_items]

    print(f"Unique items in '{column}':", unique_items_list)
```

```
Unique items in 'Age': [-1, -21, 12, -15, 1, 13, 6, 3, -9, -7, -31, 5, -19, 9, 4, 8, 7, 10, -2, -12, 11, 2, -3]
Unique items in 'Sex': ['F', 'M']
Unique items in 'HighChol': [1, 0]
Unique items in 'CholCheck': [1, 0]
Unique items in 'BMI': [18.0, 82.0, 64.0, 47.0, 58.0, 39.0, 30.0, 17.0, 26.0, 41.0, 86.0, 29.0, 23.0, 89.0, 67.0, 49.0, 56.
0, 14.0, 54.0, 21.0, 55.0, 38.0, 51.0, 48.0, 72.0, 66.0, 70.0, 65.0, 43.0, 27.0, 85.0, 53.0, 87.0, 68.0, 40.0, 12.0, 77.0,
20.0, 57.0, 35.0, 83.0, 34.0, 71.0, 50.0, 76.0, 80.0, 24.0, 69.0, 44.0, 42.0, 16.0, 45.0, 62.0, 46.0, 52.0, 22.0, 81.0, 95.
0, 98.0, 59.0, 32.0, 19.0, 25.0, 84.0, 36.0, 33.0, 79.0, 15.0, 92.0, 73.0, 31.0, 63.0, 28.0, 60.0, 61.0, 37.0, 75.0, 13.0,
78.0, 74.0]
Unique items in 'Smoker': ['Smoker', 'Non Smoker']
Unique items in 'HeartDiseaseorAttack': [1, 0]
Unique items in 'PhysActivity': [1, 0]
Unique items in 'Fruits': ['Eat', 'Does not Eat']
Unique items in 'Veggies': ['Eat', 'Does not Eat']
Unique items in 'HvyAlcoholConsump': [1, 0]
Unique items in 'GenHlth': ['very good', 'poor', 'excellent', 'good', 'fair']
Unique items in 'MentHlth': [28, 26, 27, 12, 22, 1, 13, 6, 16, 3, 20, 5, 19, 15, 9, 17, 4, 8, 23, 7, 10, 25, 24, 29, 21, 1
1, 14, 2, 30, 0, 18]
Unique items in 'PhysHlth': [28, 26, 27, 12, 22, 1, 13, 6, 16, 3, 20, 5, 19, 15, 17, 9, 4, 8, 23, 7, 10, 25, 24, 29, 21, 1
1, 14, 2, 30, 0, 18]
Unique items in 'DiffWalk': ['Y', 'N']
Unique items in 'Stroke': ['Y', 'N']
Unique items in 'HighBP': [1, 0]
Unique items in 'SugarConsumption': ['High', 'Low']
Unique items in 'Diabetes': ['Y', 'N']
```

There were inconsistencies in the data, such as in the 'Sex' column with values like 'Male', 'M', 'Female', 'F'; in the 'DiffWalk' and 'Stroke' columns with 'Yes', 'Y', 'No', 'N'; and in the 'BMI' column, where values were recorded with the unit kg/m². These inconsistencies were addressed to ensure uniformity across the dataset.

D. Filling Null Values with Mean & Mode:

```python
# Step 4: Filling null values  with mode and mean
# Defining a function to impute missing values
def impute_nulls(data):
    for column in data.columns:
        # Checking if column has null values
        if data.filter(data[column].isNull()).count() > 0:

            # Handling categorical columns (strings) by filling with mode
            if dict(data.dtypes)[column] == 'string':
                # Calculate the mode (most frequent value) for categorical columns
                mode_value = data.groupby(column).count().orderBy(F.desc('count')).first()[0]
                data = data.fillna({column: mode_value})

            # Handling numeric columns by filling with mean
            else:
                # Calculating the mean for numeric columns
                mean_value = data.agg({column: 'avg'}).collect()[0][0]
                data = data.fillna({column: mean_value})

    return data

# Applying the function to impute missing values in the dataset
data = impute_nulls(data)

# Verifying if there are any remaining null values
remaining_nulls = data.filter(F.coalesce(*[F.col(col).isNull() for col in data.columns])).count()
print(f"After imputing the null values, the number of nulls in the dataset are: {remaining_nulls}")
```

```
After imputing the null values, the number of nulls in the dataset are: 0
```

After filling the null values in the dataset, there should be no missing values remaining, as the null values in categorical columns were replaced with the mode and in numerical columns with the mean.

E. Removing Negative Values:

```python
# Step 5: Removing negative values in age column
# Checking for negative values in the 'Age' column
negative_age_count = data.filter(data['Age'] < 0).count()
print(f"Number of negative values in the 'Age' column: {negative_age_count}")

# Removing rows with negative values in the 'Age' column
data_cleaned = data.filter(data['Age'] >= 0)

# Printing the number of rows after removing negative values
remaining_rows = data_cleaned.count()
print(f"Number of rows after removing negative values: {remaining_rows}")
```

```
Number of negative values in the 'Age' column: 11
Number of rows after removing negative values: 65142
```

There are a total of 11 negative values in the 'Age' column, which need to be removed. Therefore, after their removal, the total number of remaining rows in the dataset is 65,142

F. Creating New Feature:

```python
# Step 6: Creating new attribute AgeGroup to classify as kids and teens
# Defining the age bins and labels
age_bins = [0, 10, 100]
age_labels = ['Kid', 'Teen']

# Creating a new column 'Age_Group' based on the age classification
# Using when() to classify 'Age' into 'Kid' or 'Teen'
data = data_cleaned.withColumn('Age_Group',
                       F.when((data['Age'] >= age_bins[0]) & (data['Age'] < age_bins[1]), age_labels[0])
                       .when((data['Age'] >= age_bins[1]) & (data['Age'] < age_bins[2]), age_labels[1])
                       .otherwise(None))

# Counting the number of 'Kid' and 'Teen' entries in the 'Age_Group' column
age_group_counts = data.groupBy('Age_Group').count().collect()

# Printing the result
for row in age_group_counts:
    print(f"Age Group: {row['Age_Group']}, Count: {row['count']}")
```

```
Age Group: Teen, Count: 27510
Age Group: Kid, Count: 37632
```

Classifying individuals based on their age and creating a new column, 'Age Group', with the following categories:

- Kid: Age < 10
- Teen: Age ≥ 10

G. Data Conversion:

```python
# Step 7: Data type conversion for BMI column
data = data.withColumn("BMI", F.col("BMI").cast("int"))
data = data.withColumn("Age", F.col("Age").cast("int"))

data.show(5)
```

```
+---+---+--------+---------+---+----------+--------------------+------------+------------+------------+-----------------+--
-------+--------+--------+--------+------+------+----------------+--------+---------+
|Age|Sex|HighChol|CholCheck|BMI|    Smoker|HeartDiseaseorAttack|PhysActivity|      Fruits|     Veggies|HvyAlcoholConsump|
GenHlth|MentHlth|PhysHlth|DiffWalk|Stroke|HighBP|SugarConsumption|Diabetes|Age_Group|
+---+---+--------+---------+---+----------+--------------------+------------+------------+------------+-----------------+--
-------+--------+--------+--------+------+------+----------------+--------+---------+
|  3|  F|       0|        1| 39|Non Smoker|                   0|           1|Does not Eat|Does not Eat|                0|
good|       2|       0|       N|     N|     0|            High|       N|      Kid|
|  9|  M|       1|        1| 29|Non Smoker|                   0|           0|         Eat|         Eat|                0|
good|       0|       0|       N|     N|     0|            High|       N|      Kid|
|  8|  F|       0|        1| 28|Non Smoker|                   0|           1|         Eat|         Eat|                0|ex
cellent|       0|       0|       N|     N|     0|            High|       N|      Kid|
|  8|  M|       1|        1| 26|Non Smoker|                   0|           0|         Eat|         Eat|                0|ve
ry good|       0|       0|       N|     N|     0|            High|       N|      Kid|
|  9|  M|       0|        1| 27|    Smoker|                   0|           1|         Eat|         Eat|                0|
good|       0|       0|       N|     N|     0|            High|       N|      Kid|
+---+---+--------+---------+---+----------+--------------------+------------+------------+------------+-----------------+--
-------+--------+--------+--------+------+------+----------------+--------+---------+
only showing top 5 rows
```

The 'BMI' and 'Age' columns are converted to integers to simplify analysis using mathematical techniques that require numerical precision.

H. BMI Group Classification:

```python
# Step 8: Creating new attribute "BMI_Group"
bmi_bins = [12, 25, 30, 40, 100]
bmi_labels = ['UnderWeight', 'HealthyWeight', 'OverWeight', 'Obese']

# Using when() to categorize BMI
data = data.withColumn(
    "BMI_Group",
    F.when(F.col("BMI") <= bmi_bins[1], bmi_labels[0])
     .when((F.col("BMI") > bmi_bins[1]) & (F.col("BMI") <= bmi_bins[2]), bmi_labels[1])
     .when((F.col("BMI") > bmi_bins[2]) & (F.col("BMI") <= bmi_bins[3]), bmi_labels[2])
     .otherwise(bmi_labels[3])
)


# Counting the number of records in each BMI group
bmi_group_counts = data.groupBy("BMI_Group").count()
bmi_group_counts.show()
```

```
+-------------+-----+
|    BMI_Group|count|
+-------------+-----+
|        Obese| 4966|
|   OverWeight|20834|
|HealthyWeight|22677|
|  UnderWeight|16665|
+-------------+-----+
```

Classifying individuals according to their BMI and creating a new feature called 'BMI Group', which includes the categories: 'Underweight', 'Healthy Weight', 'Overweight', and 'Obese'.

I. Label Encoding:

```python
# Step 9: Label Encoding to convert categorical values into numerical labels
# Identifying categorical columns by checking the data type
categorical_columns = [col_name for col_name, dtype in data.dtypes if dtype == "string"]

# Applying StringIndexer for each categorical column
indexers = [StringIndexer(inputCol=col, outputCol=f"{col}_Index").fit(data) for col in categorical_columns]

# Transforming the data using each indexer
for indexer in indexers:
    data = indexer.transform(data)
```

```python
# Dropping the original string columns
data = data.drop(*categorical_columns)

# Verifying the columns in the DataFrame
print("Columns in the DataFrame after dropping original string columns:", data.columns)

data.show(5)
```

```
Columns in the DataFrame after dropping original string columns: ['Age', 'HighChol', 'CholCheck', 'BMI', 'HeartDiseaseorAtt
ack', 'PhysActivity', 'HvyAlcoholConsump', 'MentHlth', 'PhysHlth', 'HighBP', 'Sex_Index', 'Smoker_Index', 'Fruits_Index',
'Veggies_Index', 'GenHlth_Index', 'DiffWalk_Index', 'Stroke_Index', 'SugarConsumption_Index', 'Diabetes_Index', 'Age_Group_
Index', 'BMI_Group_Index']
```

Age	HighChol	CholCheck	BMI	HeartDiseaseorAttack	PhysActivity	HvyAlcoholConsump	MentHlth	PhysHlth	HighBP	Sex_Index	Smoker_Index	Fruits_Index	Veggies_Index	GenHlth_Index	DiffWalk_Index	Stroke_Index	SugarConsumption_Index	Diabetes_Index	Age_Group_Index	BMI_Group_Index
3	0	1	39	0	1	0	2	0	0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
9	1	1	29	0	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
8	0	1	28	0	1	0	0	0	0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0	1.0	0.0	0.0
8	1	1	26	0	0	0	0	0	0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0
9	0	1	27	0	1	0	0	0	0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0

only showing top 5 rows

Label Encoding transforms a categorical variable into a numerical format, making it more suitable for use in machine learning models.

J. Dropping Features:

```python
# Step 10: Dropping Features
# Count the occurrences of each value in 'SugarConsumption_Index'
sugar_consumption_counts = data.groupBy('SugarConsumption_Index').count().orderBy('SugarConsumption_Index')

# Showing the counts for each label in 'SugarConsumption_Index'
sugar_consumption_counts.show()

# Dropping the 'SugarConsumption_Index' column
data = data.drop('SugarConsumption_Index')
```

SugarConsumption_Index	count
0.0	65138
1.0	4

By using feature extraction, we dropped the Sugar Consumption Index feature as it contains only 4 entries marked as 'High' making it low variance.

K. Removing Outliers:

```python
# Step 11: Removing outliers
# Specifying the columns to clean (BMI and Age)
columns_to_clean = ['BMI', 'Age']

# Calculateing Q1, Q3, and IQR for BMI and Age columns, and filter out outliers
for col_name in columns_to_clean:
    # Calculate Q1 (25th percentile), Q3 (75th percentile), and IQR (Interquartile Range)
    q1 = data.approxQuantile(col_name, [0.25], 0.01)[0]
    q3 = data.approxQuantile(col_name, [0.75], 0.01)[0]
    iqr = q3 - q1

    # Defining the lower and upper bounds for outliers
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr

    # Printing the bounds for the column
    print(f"Lower bound for '{col_name}': {lower_bound}")
    print(f"Upper bound for '{col_name}': {upper_bound}")

    # Filtering out the rows that are outliers for this column
    data = data.filter((F.col(col_name) >= lower_bound) & (F.col(col_name) <= upper_bound))

# Printing the number of rows after removing outliers
print(f"Number of rows after removing outliers: {data.count()}")
```

```
Lower bound for 'BMI': 13.0
Upper bound for 'BMI': 45.0
Lower bound for 'Age': 1.0
Upper bound for 'Age': 17.0
Number of rows after removing outliers: 63003
```

Using the IQR method, we identified outliers based on the upper and lower bounds. After removing these outliers, the dataset now contains 63,003 rows.

L. Rolling average of BMI:

```python
# Step 12: Rolling average of BMI
# Defining a rolling window grouped by BMI_Group_Index and ordered by Age
rolling_window = Window.partitionBy("BMI_Group_Index").orderBy("Age").rowsBetween(-2, 2)

# Adding a column for rolling average of BMI
data = data.withColumn("Rolling_BMI_Avg", F.avg("BMI").over(rolling_window))

# Showing the result with the Rolling_BMI_Avg column
data.select("BMI_Group_Index", "Age", "BMI", "Rolling_BMI_Avg").show(10)
```

BMI_Group_Index	Age	BMI	Rolling_BMI_Avg
0.0	1	26	26.666666666666668
0.0	1	26	26.75
0.0	1	28	26.6
0.0	1	27	26.6
0.0	1	26	26.8
0.0	1	26	26.8
0.0	1	27	27.0
0.0	1	28	27.4
0.0	1	28	27.8
0.0	1	28	27.8

only showing top 10 rows

The rolling average of BMI is calculated using a window grouped by 'BMI Group Index' and ordered by 'Age'. This allows for smoother BMI values by averaging over a 5-year age range.

M. Grouping by BMI_Group$_{I}ndexandCalculatingtheCount$:

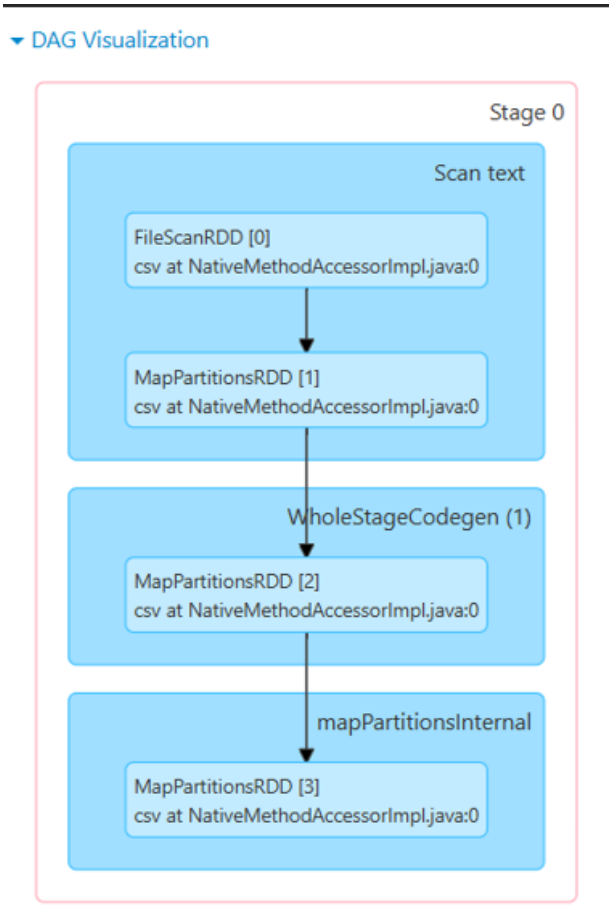
```python
# Step 13:Grouping by BMI_Group_Index and Calculating the Count
data_grouped = data.groupBy("BMI_Group_Index").agg(F.count("*").alias("count"))

# Defining a window partitioned by BMI_Group_Index and ordered by count descending
ranking_window = Window.orderBy(F.desc("count"))

# Adding the rank column
data_with_rank = data_grouped.withColumn("Rank", F.rank().over(ranking_window))

data_with_rank.show(10)
```

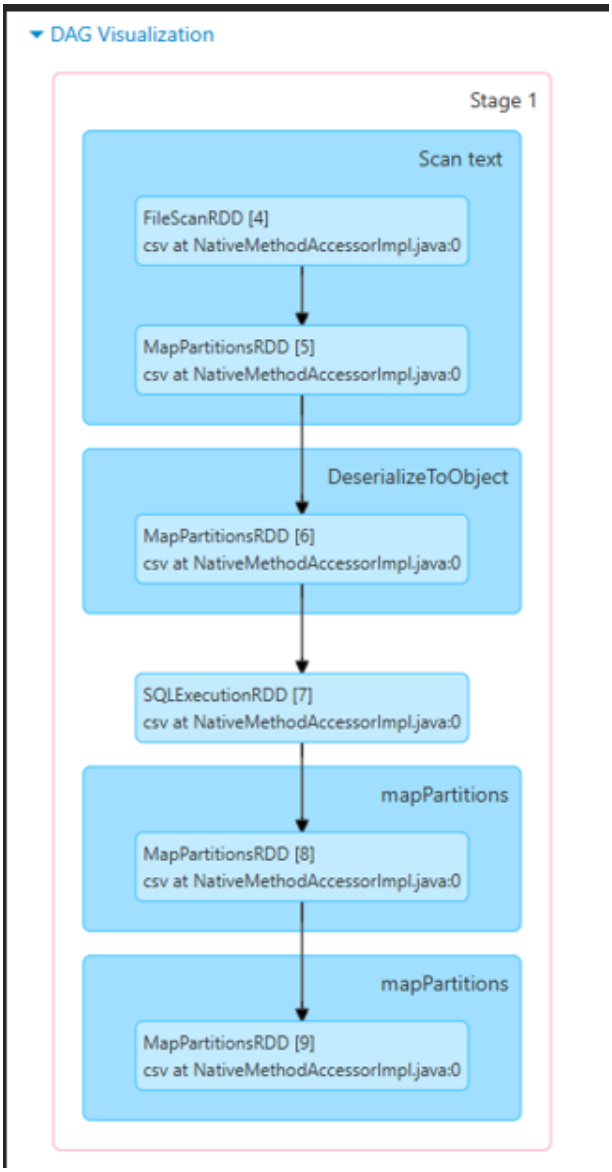
BMI_Group_Index	count	Rank
0.0	22677	1
1.0	20834	2
2.0	16664	3
3.0	2828	4

The dataset is grouped by 'BMI Group Index' to calculate the count of entries in each group. Then, ranks are assigned to each BMI group based on the count in descending order.

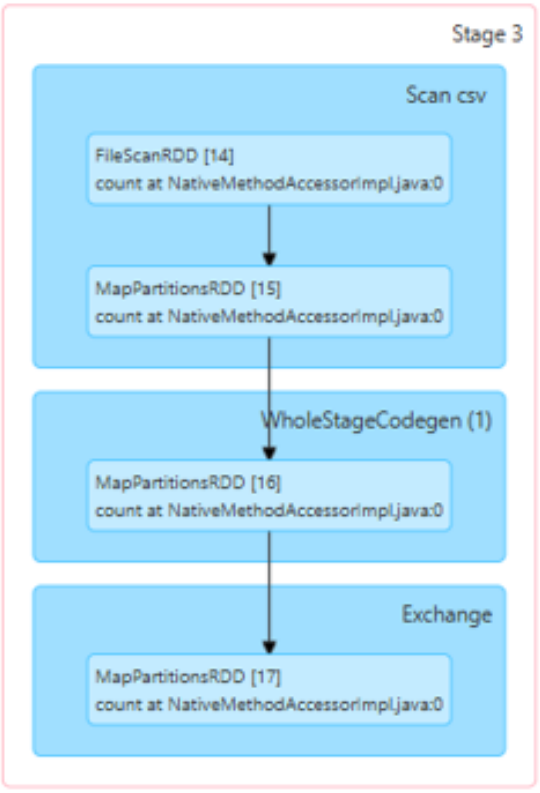
III. DAG VISUALISATIONS

A. *Stage 0 - 135:*

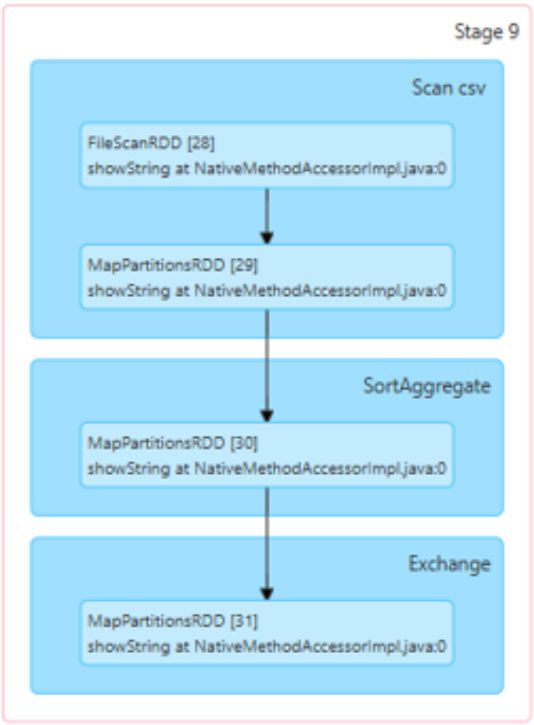




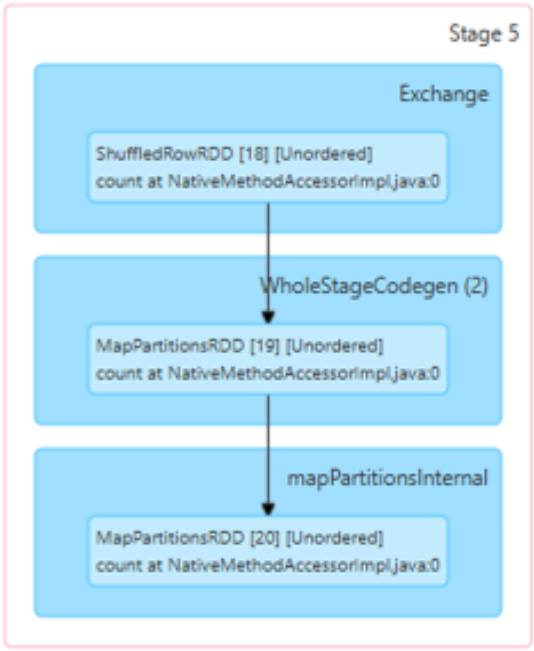
DAG Visualization



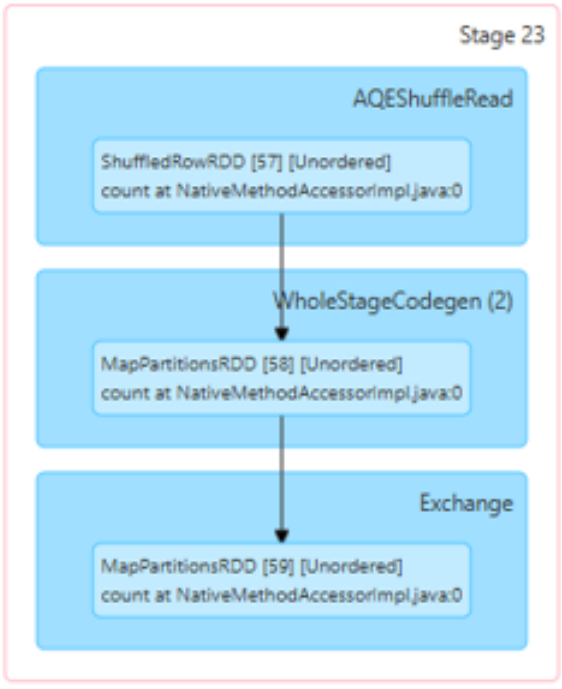
DAG Visualization

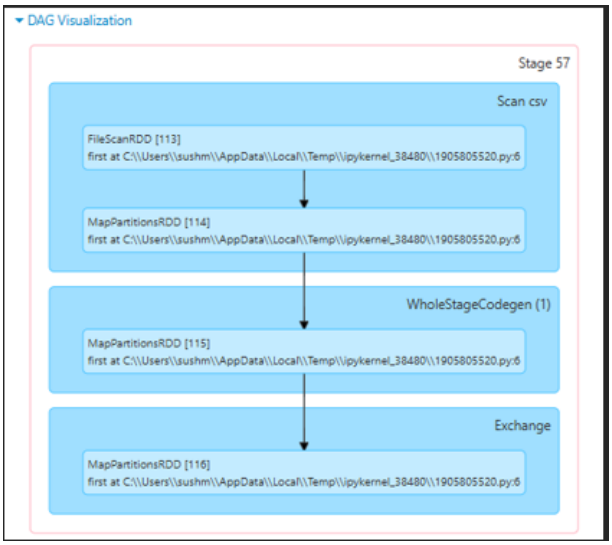


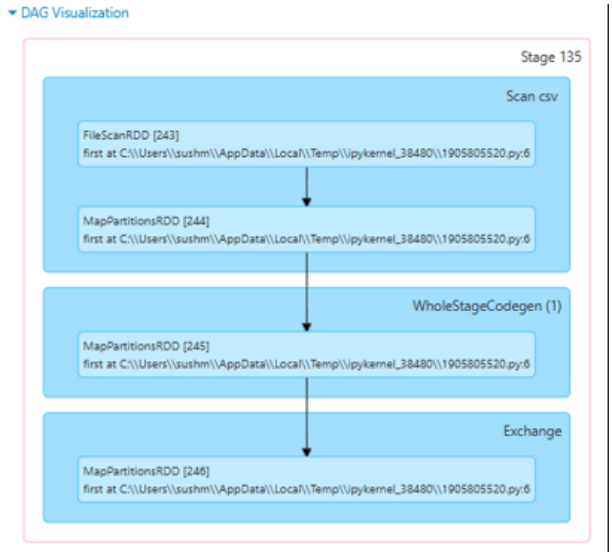
DAG Visualization

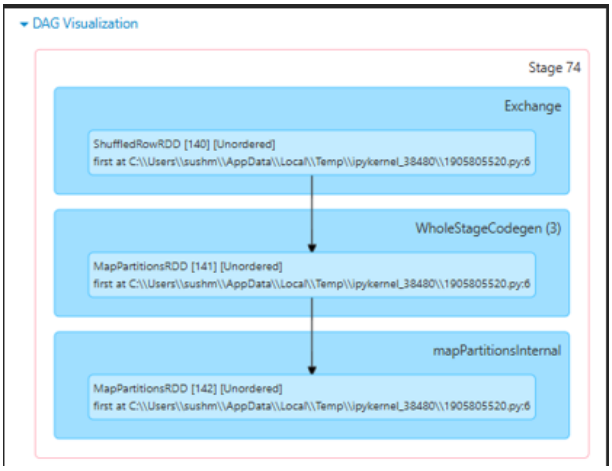


DAG Visualization





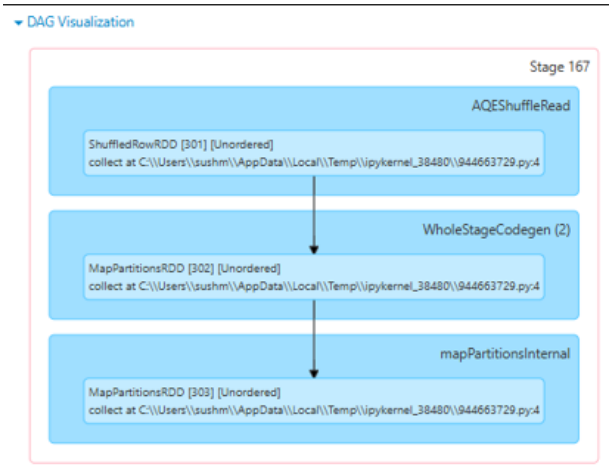


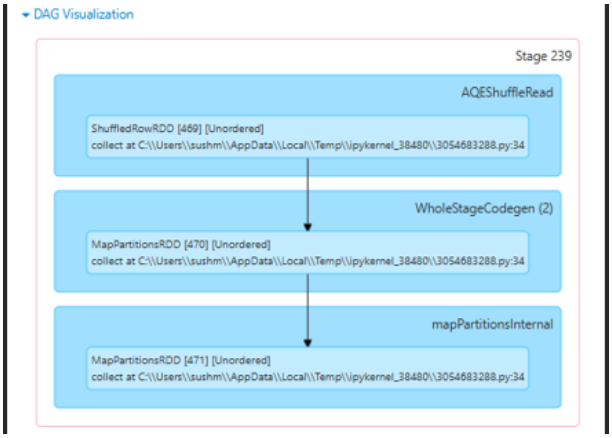


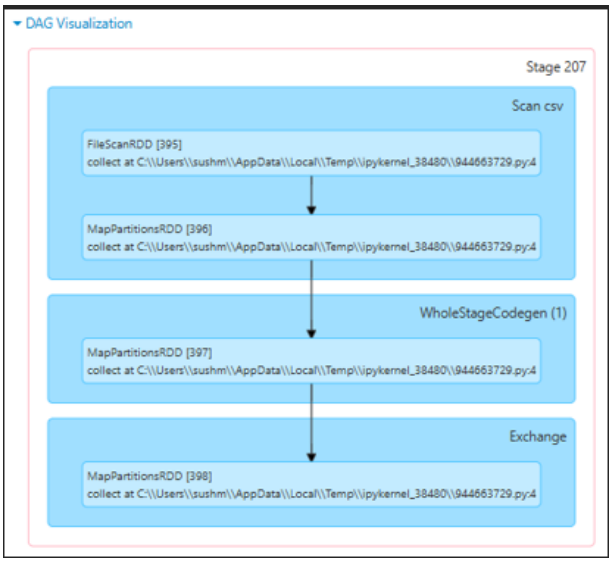
Observations:

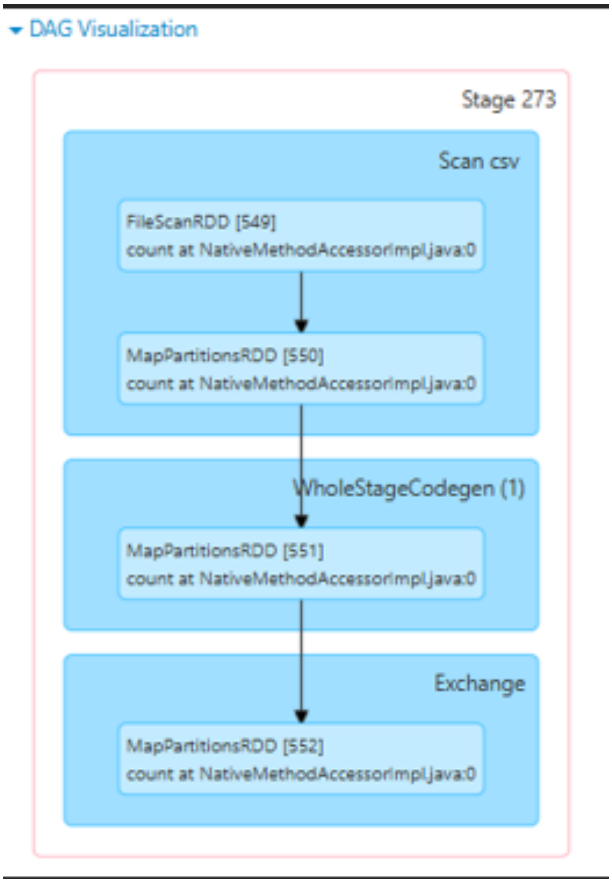
- Stage 0-9: These stages depicts an initial stage setup, including reading input data and performing foundational transformations.
- Stage 23-57: Represents intermediate transformations with fewer shuffle operations, indicating a relatively linear computation.
- Stage 74-135: Shows complex operations with multiple shuffle and partitioning steps, for aggregations or joins.

B. Stage 167 - 273:

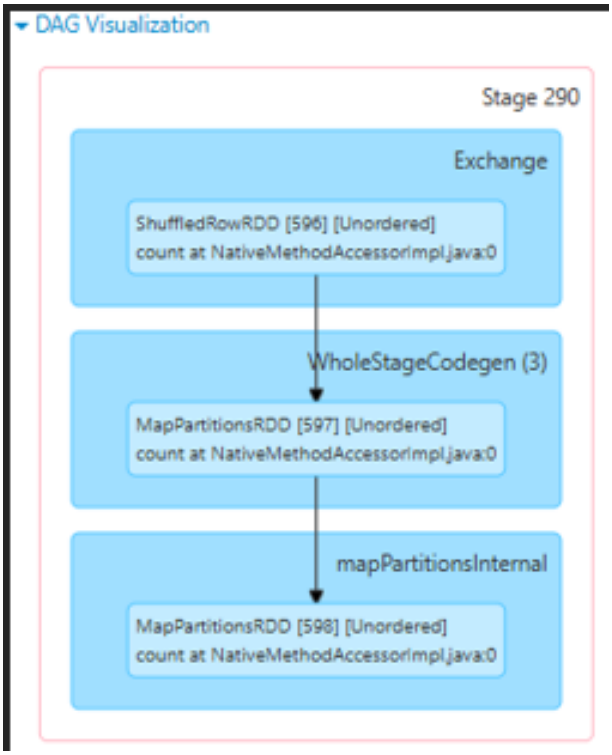






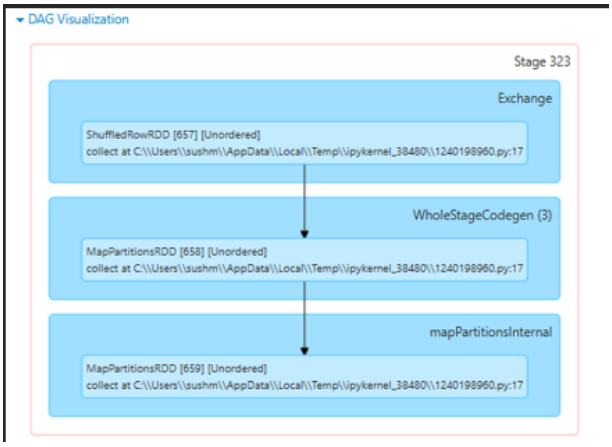


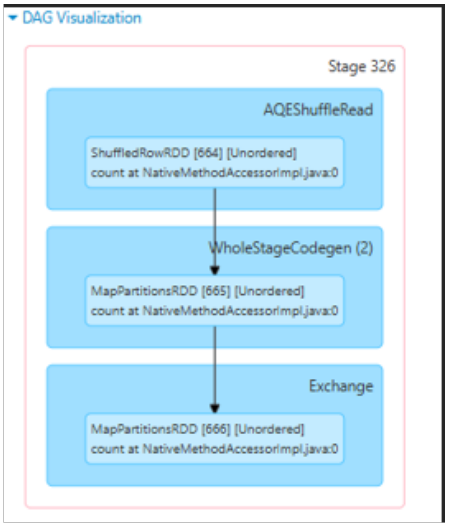
C. Stage 290 - 348:



Observations:

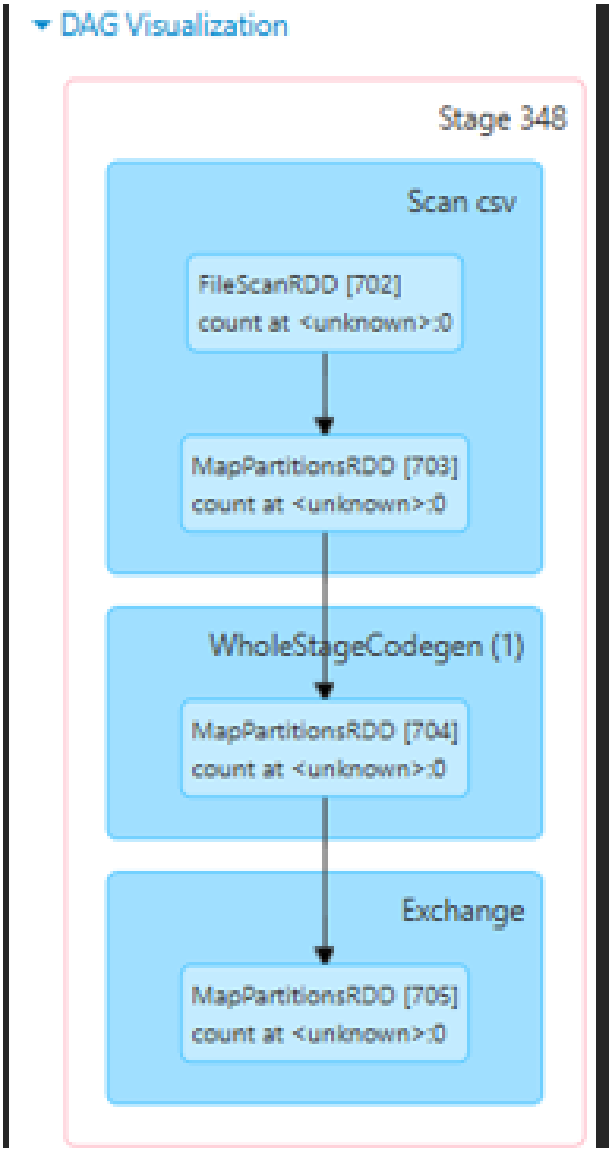
- Stage 167-207: This stage represent an initial phase of progression, potentially focusing on early interactions or fundamental processes in the overall sequence. It also involves foundational steps or smaller sub-tasks that contribute to building the next stages.
- Stage 239-261: Depicts stages with shuffle and partitioning steps, likely handling data redistribution or sorting operations. These stages indicate moderately complex transformations like WholeStageCodegen and Exchange.
- Stage 273: Represents a terminal stage with relatively simple operations such as scanning input (Scan csv) and mapping over partitions. This is indicative of preparing the final dataset for output or evaluation.



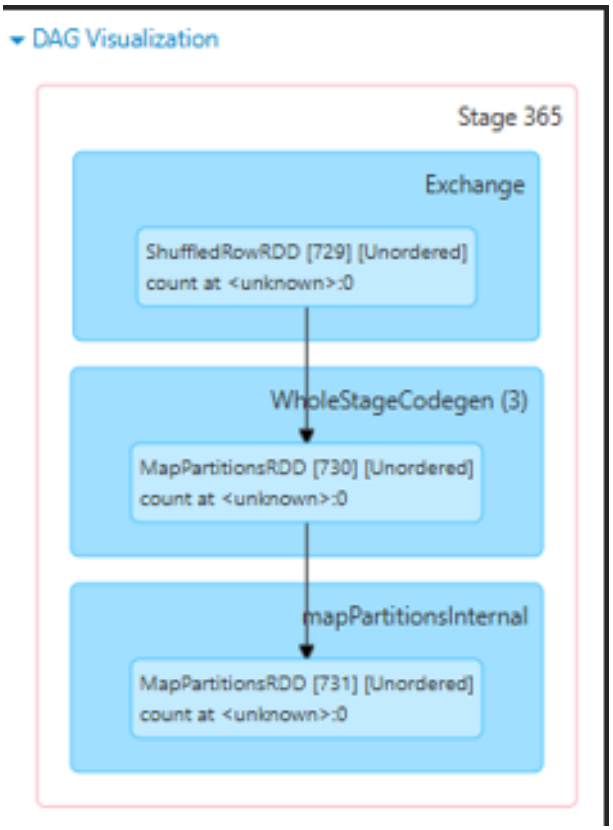


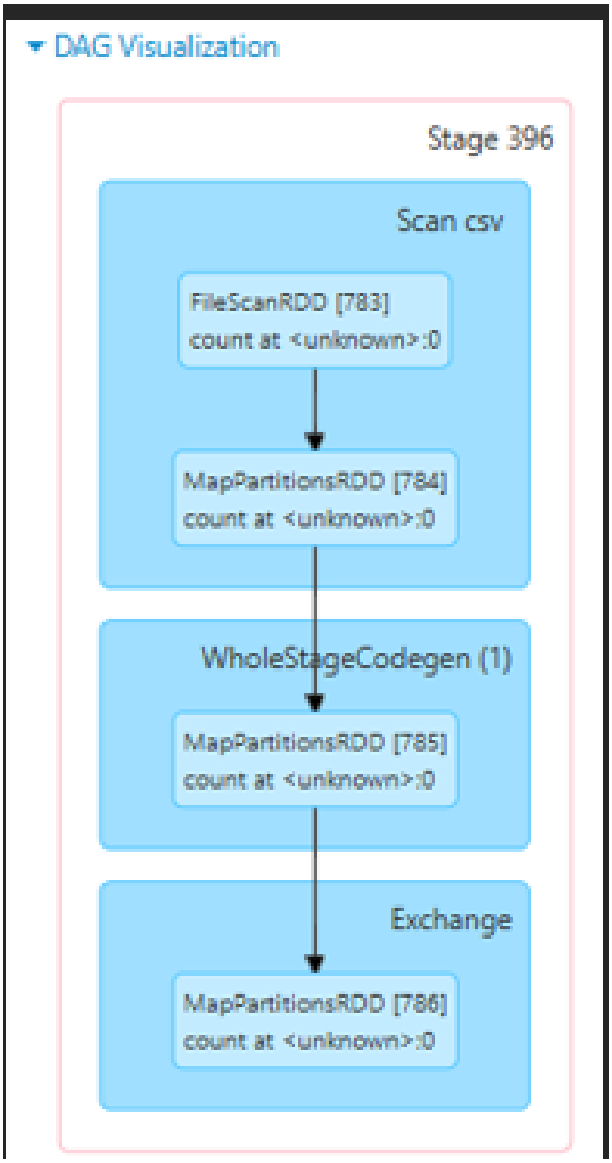
Observations:

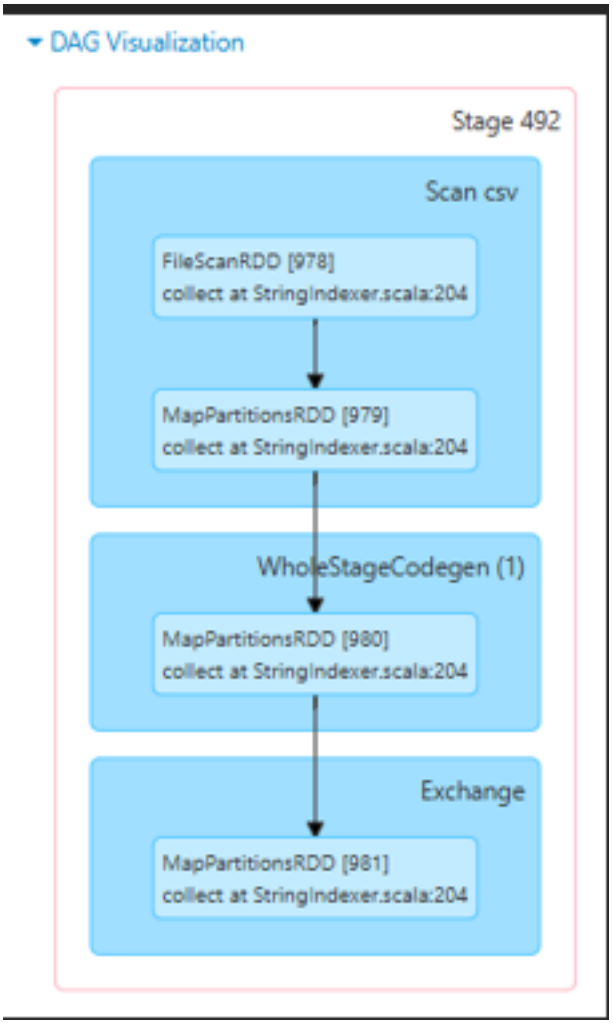
- Stage 290-323: Involves shuffle and partitioning operations, likely part of a join or groupBy operation, as indicated by the Exchange and ShuffleRDD. The WholeStageCodegen indicates optimized execution of map functions within these stages.
- Stage 326: Represents an AGGShuffleRead, indicating aggregation-like operations post-shuffle. This stage is responsible for processing results from the previous shuffle operations.
- Stage 348: This stage shows the final stage for scanning input data and performing lightweight transformations (e.g., Scan csv and WholeStageCodegen) before an action such as writing or collecting results.

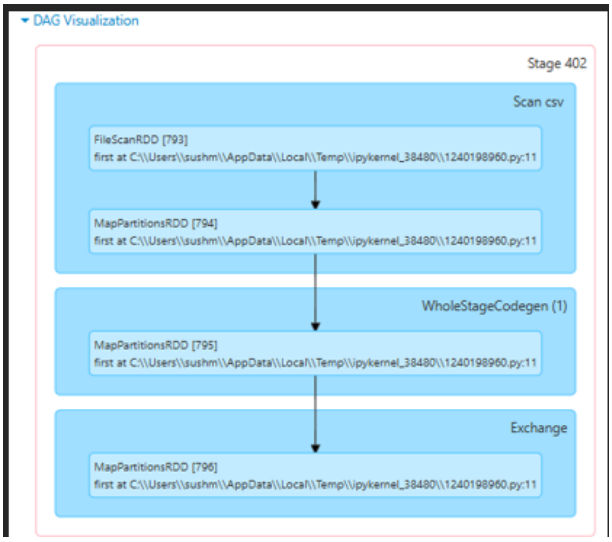


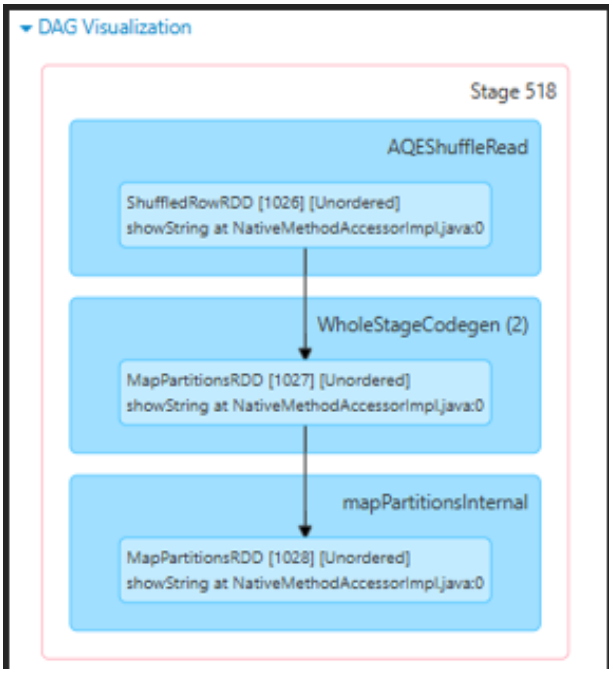
D. Stage 365 - 554:

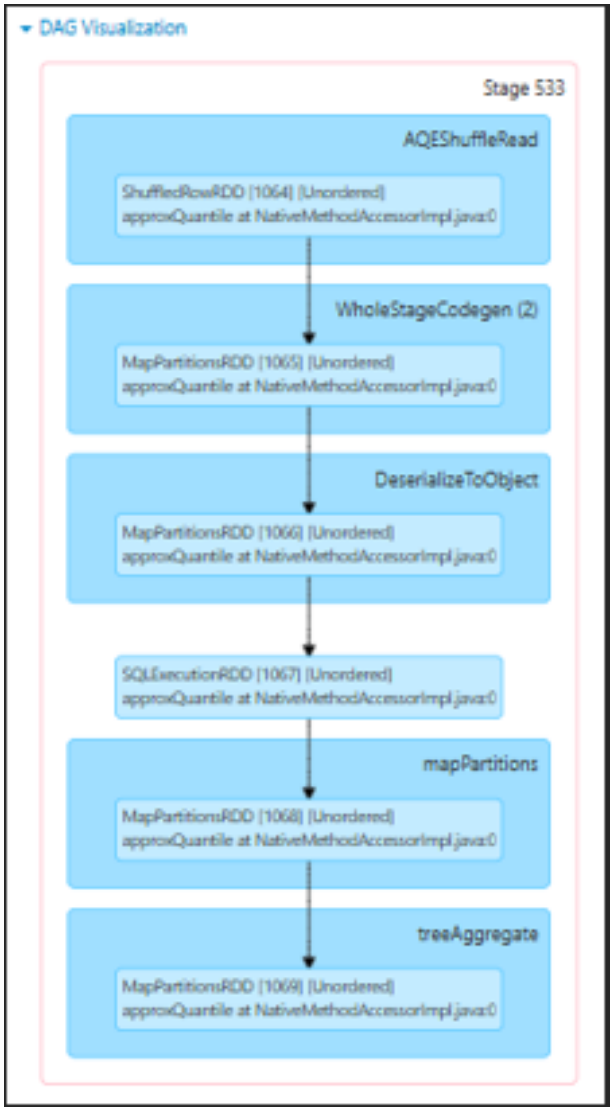



DAG Visualization
Stage 396
Scan csv
FileScanRDD [783]
count at <unknown>:0
MapPartitionsRDD [784]
count at <unknown>:0
WholeStageCodegen (1)
MapPartitionsRDD [785]
count at <unknown>:0
Exchange
MapPartitionsRDD [786]
count at <unknown>:0


DAG Visualization
Stage 492
Scan csv
FileScanRDD [978]
collect at StringIndexer.scala:204
MapPartitionsRDD [979]
collect at StringIndexer.scala:204
WholeStageCodegen (1)
MapPartitionsRDD [980]
collect at StringIndexer.scala:204
Exchange
MapPartitionsRDD [981]
collect at StringIndexer.scala:204


DAG Visualization
Stage 402
Scan csv
FileScanRDD [793]
first at C:\\Users\\sushm\\AppData\\Local\\Temp\\ipykernel_38480\\1240198960.py:11
MapPartitionsRDD [794]
first at C:\\Users\\sushm\\AppData\\Local\\Temp\\ipykernel_38480\\1240198960.py:11
WholeStageCodegen (1)
MapPartitionsRDD [795]
first at C:\\Users\\sushm\\AppData\\Local\\Temp\\ipykernel_38480\\1240198960.py:11
Exchange
MapPartitionsRDD [796]
first at C:\\Users\\sushm\\AppData\\Local\\Temp\\ipykernel_38480\\1240198960.py:11


DAG Visualization
Stage 518
AQEShuffleRead
ShuffledRowRDD [1026] [Unordered]
showString at NativeMethodAccessorImpl.java:0
WholeStageCodegen (2)
MapPartitionsRDD [1027] [Unordered]
showString at NativeMethodAccessorImpl.java:0
mapPartitionsInternal
MapPartitionsRDD [1028] [Unordered]
showString at NativeMethodAccessorImpl.java:0


DAG Visualization
Stage 533
AQEShuffleRead
ShuffledRowRDD [1064] [Unordered]
approxQuantile at NativeMethodAccessorImpl.java:0
WholeStageCodegen (2)
MapPartitionsRDD [1065] [Unordered]
approxQuantile at NativeMethodAccessorImpl.java:0
DeserializeToObject
MapPartitionsRDD [1066] [Unordered]
approxQuantile at NativeMethodAccessorImpl.java:0
SQLExecutionRDD [1067] [Unordered]
approxQuantile at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [1068] [Unordered]
approxQuantile at NativeMethodAccessorImpl.java:0
treeAggregate
MapPartitionsRDD [1069] [Unordered]
approxQuantile at NativeMethodAccessorImpl.java:0


DAG Visualization
Stage 543
Scan csv
FileScanRDD [1090]
showString at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [1091]
showString at NativeMethodAccessorImpl.java:0
WholeStageCodegen (1)
MapPartitionsRDD [1092]
showString at NativeMethodAccessorImpl.java:0
Exchange
MapPartitionsRDD [1093]
showString at NativeMethodAccessorImpl.java:0

Observations:

- Stage 365-492: This stage could represent a transitional phase where intermediate processes or tasks occur. It also involves more complex interrelations or operations, suggesting the progression from basic to more refined systems or workflows.
- Stage 518-554: This is a culmination or final phase, characterized by the convergence of prior inputs or processes into a comprehensive output. It also represent the achievement of objectives or a final synthesis in the context of the system.

IV. ALGORITHMS

In this project, we applied six significant algorithms to predict diabetes risk using PySpark's MLlib. The selected algorithms include both commonly discussed methods and those not covered in class.

Algorithms Applied:

- Logistic Regression
- Naïve Bayes
- Support Vector Machines (SVM)
- Random Forest
- Gradient Boosting (XGBoost)
- Decision Tree
- Multi Layer Perceptron

A. Logistic Regression

Logistic regression is a widely used supervised machine learning technique designed to estimate the likelihood of binary outcomes, such as whether a patient is diagnosed with diabetes.

The main strength of logistic regression is its straightforwardness. The model provides clear insights into how different variables contribute to predicted outcomes, making it easier for users to understand the relationship between predictors and diabetes probability Coefficients from the logistic regression equation represent the odds associated with each predictor.

Furthermore, logistic regression is known for its complexity. In general, it performs better when applied to new data, as it is less likely to overfit the training data. This characteristic is particularly important in real-world applications, where the training data may not fully reflect the state of the data used to predict. By balancing accuracy, logistic regression is a reliable tool in the healthcare context for diabetes prediction.

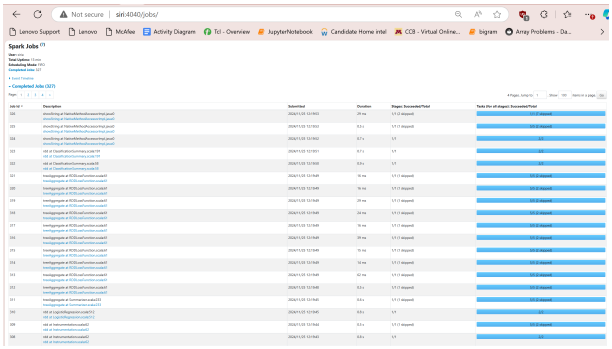
We trained the logistic regression model using a pipeline that includes the necessary data preprocessing (through the VectorAssembler) and model fitting stages. The model achieved an accuracy of 79.0%, demonstrating its strong performance in predicting whether individuals have diabetes based on the given features. The total execution time for training the model and making predictions was 11.43 seconds, demonstrating the efficiency of the logistic regression implementation in PySpark, even with large datasets.

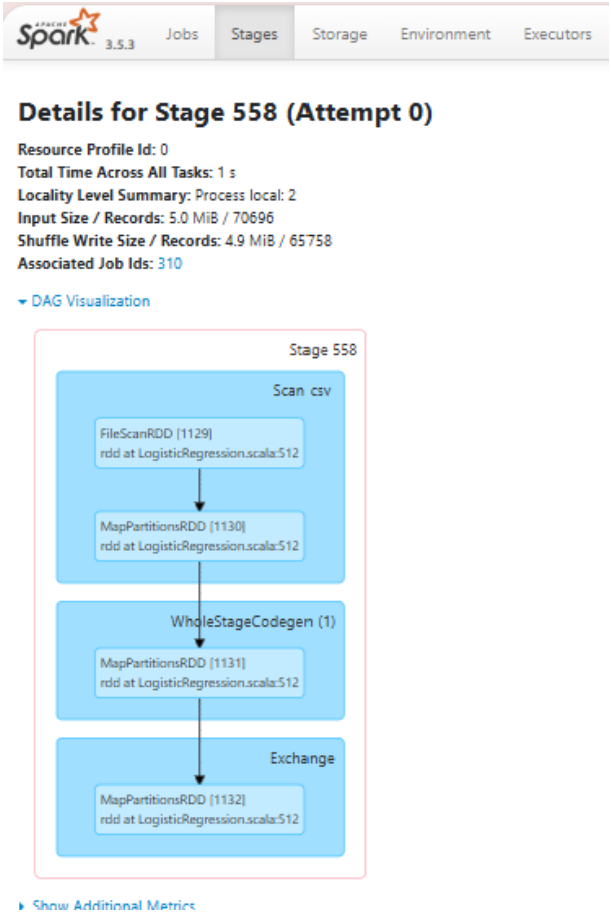
Logistic Regression Confusion matrix & Execution time:

```
Confusion Matrix:
+--------------+----------+-----+
|Diabetes_Index|prediction|count|
+--------------+----------+-----+
|           0.0|       0.0| 4882|
|           0.0|       1.0| 1581|
|           1.0|       0.0| 1944|
|           1.0|       1.0| 4177|
+--------------+----------+-----+

Execution Time: 11.43158483505249 seconds
```

DAG Visualization:

Below are the jobs created for the implementation of Logistic Regression model (Job Id 308 to 326)





This is the main stage where the logistic regression model

is trained and predictions are made. The steps involved are:

- AQEShuffleRead: Reads data from other executors.
- WholeStageCodegen: Optimizes and generates code for efficient execution.
- DeserializeToObject: Deserializes network data.
- MapPartitions: Applies a map function to each data partition for feature extraction, transformation, and label assignment.
- map: Further cleans or preprocesses each RDD element.
- mapPartitions: Prepares data for training.
- LogisticRegression.trainBlocks: Trains the logistic regression model.
- treeAggregate: Aggregates results to compute final model parameters.

B. Naive Bayes

Naive Bayes is one of the most popular supervised machine learning algorithms, based on Bayes' Theorem. It works exceptionally well for classification problems in which the features are conditionally independent given the class label. In diabetes prediction, Naive Bayes estimates the probability of an individual being diagnosed with diabetes based on various health-related features.

The major strengths of Naive Bayes include simplicity and speed. Therefore, it bodes well on big, high-dimensional datasets and it is to be recommended in distributed computing environments such as PySpark. Although independence between features is an unrealistic assumption, in many classification tasks, the results turned out to be good, and it generalizes well to unseen data because it suffers less from overfitting.

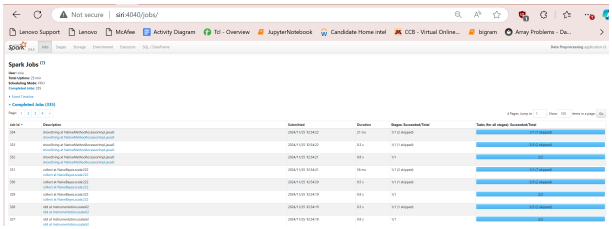
We implemented the training of the Naive Bayes model using a pipeline that involved preprocessing the data with VectorAssembler and fitting the model. The total execution time taken to train and make predictions was 3.20 seconds, which is pretty decent. While delivering an accuracy of 68.3

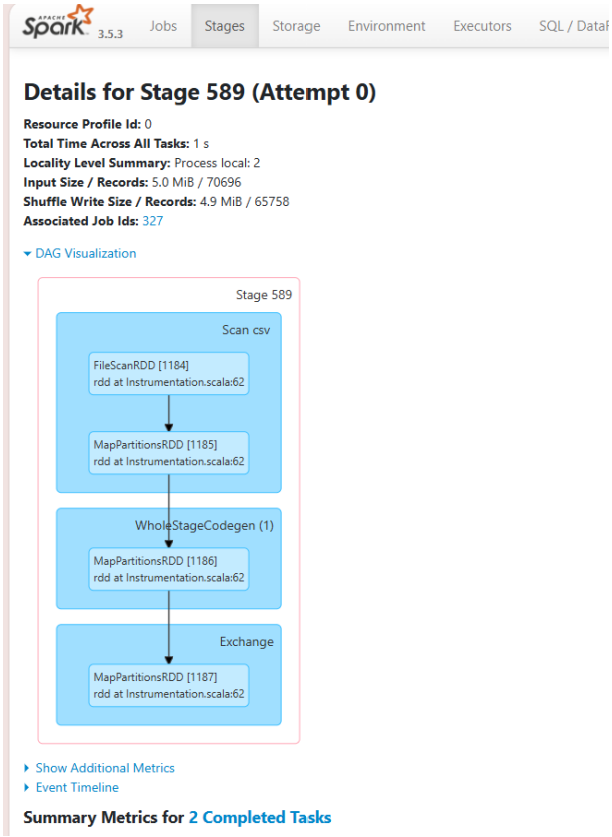
Naive Bayes Confusion matrix & Execution time:

```
Confusion Matrix:
+--------------+----------+-----+
|Diabetes_Index|prediction|count|
+--------------+----------+-----+
|           0.0|       0.0| 2380|
|           0.0|       1.0| 4083|
|           1.0|       0.0| 1254|
|           1.0|       1.0| 4867|
+--------------+----------+-----+

Execution Time: 3.202589750289917 seconds
```

DAG Visualization:

Below are the jobs created for the implementation of Naive Bayes model (Job Id 327 to 334)





The steps involved in above stage are:

- AQEShuffleRead: Reads the data from other executors.
- WholeStageCodegen (2): Performs optimizations and code generation for efficient execution.

- ObjectHashAggregate: Aggregates data to compute statistics needed for Naive Bayes model training.
- WholeStageCodegen (3): Performs further optimizations and code generation.
- mapPartitionsInternal: Applies the Naive Bayes algorithm to the aggregated data to train the model and make predictions and then Collects the results of the prediction stage and brings them back to the driver program.

C. Support Vector Machine

Linear SVM is a supervised classification algorithm that finds the optimal hyperplane to separate data into distinct classes by maximizing the margin between them. In diabetes prediction, SVM identifies the boundary between diabetic and non-diabetic individuals based on health features. The linear version of SVM is effective when data is nearly linearly separable, making it suitable for high-dimensional problems. SVMs excel in handling complex datasets, making them a powerful tool in machine learning.

The PySpark implementation of LinearSVC (Linear Support Vector Classification) in MLlib provides an efficient, distributed way to apply SVM to large datasets. By leveraging Spark's distributed computing, the model can handle vast amounts of data, making it scalable for real-world applications.

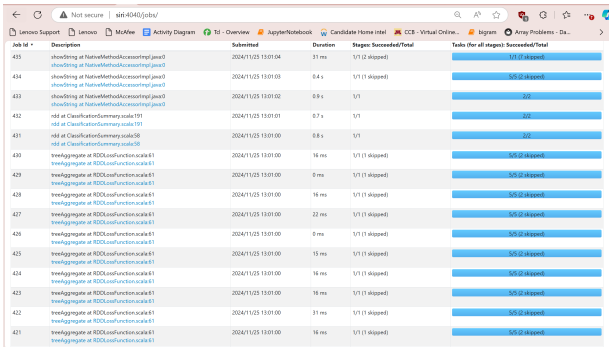
The model achieved an accuracy of 78.5%, demonstrating its effectiveness in classifying diabetic and non-diabetic individuals based on health data. The confusion matrix highlights the model's ability to distinguish between the two classes. With an execution time of just 8.63 seconds for training and prediction in PySpark, the use of distributed computing proves the model's efficiency in handling large-scale data. Despite being computationally intensive, SVM's performance and speed make it suitable for real-world healthcare applications, such as diabetes prediction.

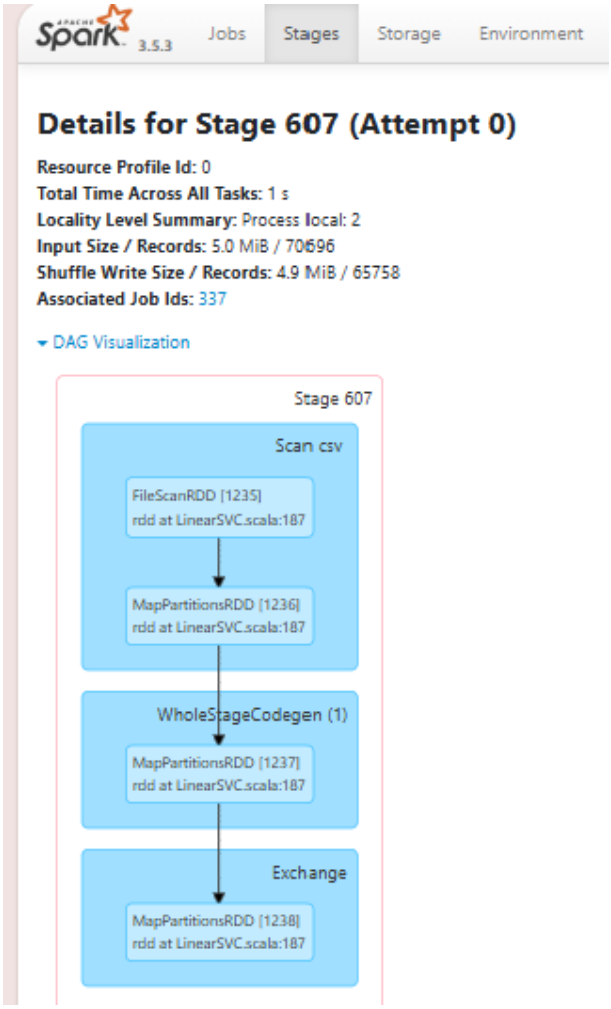
Support Vector Machine Confusion matrix & Execution time:

```
Confusion Matrix:
+--------------+----------+-----+
|Diabetes_Index|prediction|count|
+--------------+----------+-----+
|           0.0|       0.0| 4974|
|           0.0|       1.0| 1489|
|           1.0|       0.0| 2120|
|           1.0|       1.0| 4001|
+--------------+----------+-----+

Execution Time: 8.627492427825928 seconds
```

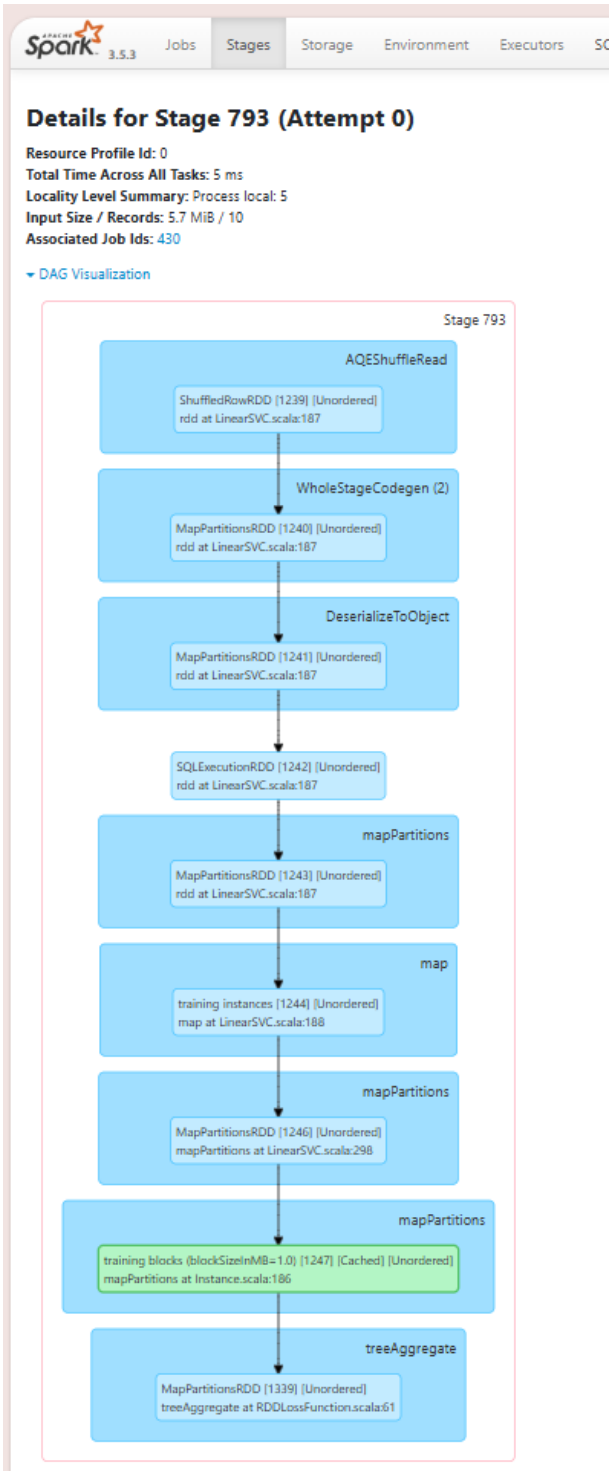
DAG Visualization:

Below are the jobs created for the implementation of Support Vector Machine model (Job Id 335 to 435)





Below DAG depicts a Spark job starting to read a CSV file; this is probably a training dataset. The data gets converted into an RDD and goes through a number of transformations, partitioning, and mapping. WholeStageCodegen optimizes the whole process by combining these transformations into one stage. This is followed by shuffling and partitioning of the data for efficient distribution across the cluster. At the last stage, train the data prepared above with the Linear SVM model.



D. Random Forest Classifier

One of the most useful ensemble learning techniques is the Random Forest technique, where several decision trees are combined to obtain a more accurate result. In the process, each tree is trained on random subsets of data and features for better generalization and lower risk of overfitting, making Random Forest more powerful in classification problems like diabetes prediction. The resultant algorithm is far more accurate and less prone to errors, especially for a big complex dataset, compared to a single decision tree.

This model was the result of a pipeline that involved data preprocessing through VectorAssembler and the fitting of the Random Forest model. Indeed, it had classified diabetic versus non-diabetic subjects based on their health features with an accuracy of 78.1%. This is further confirmed by the confusion matrix, which shows a detailed breakdown of true positives, true negatives, false positives, and false negatives regarding the capabilities of the model in differentiating between the two classes.

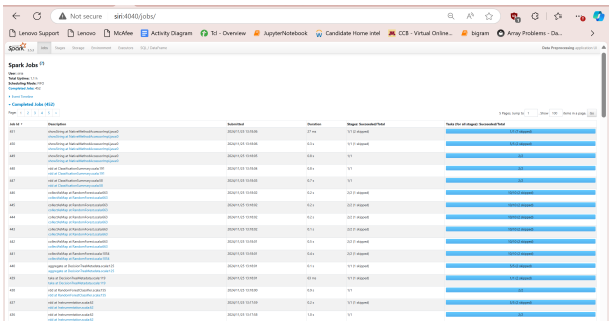
The execution time for training the model and making predictions was 8.22 seconds, which showed the efficiency of the PySpark environment in handling large-scaled data. Despite the high load of computation when training several decision trees, Random Forest remains efficient and appropriate for a distributed environment like PySpark. This is appropriate for practical applications in real time, such as in the health sector, that requires speed and accuracy in the prediction of diabetes.

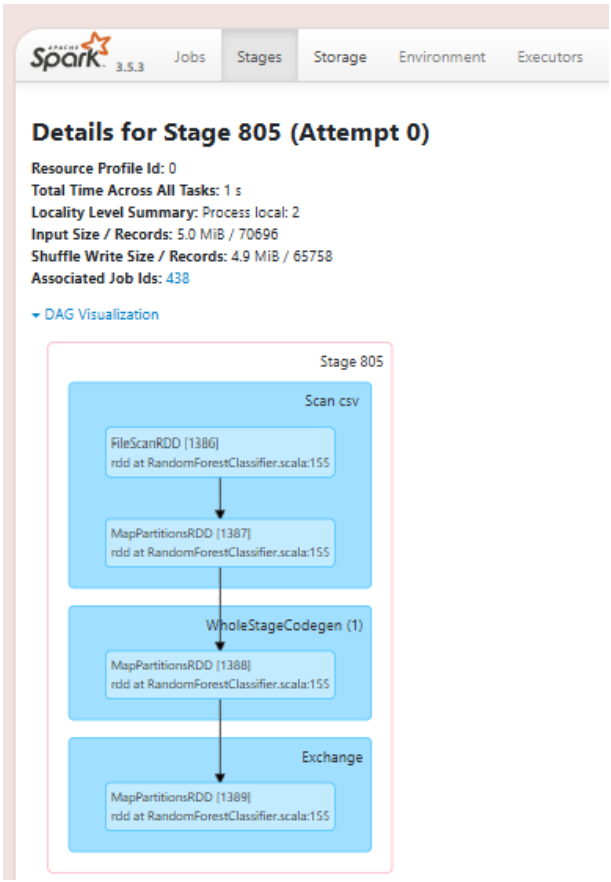
Random Forest Confusion matrix & Execution time:

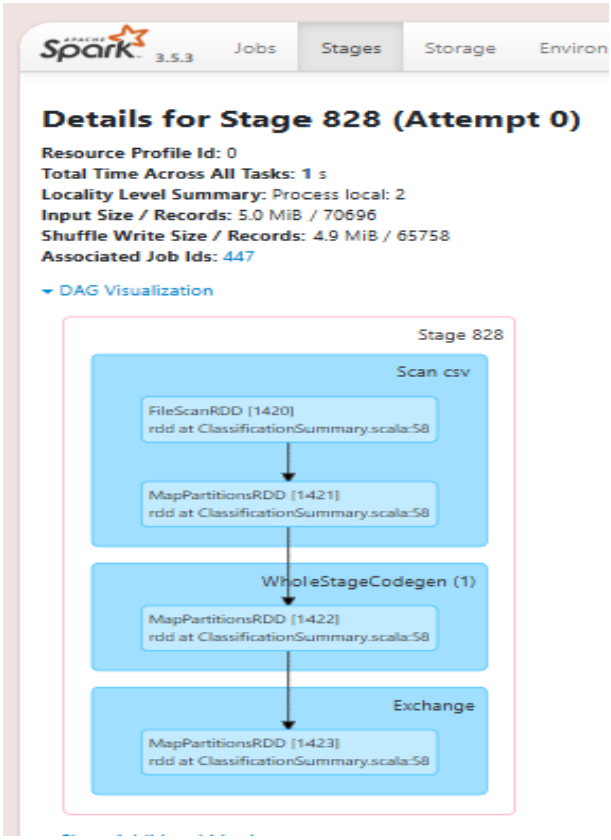
```
Confusion Matrix:
+--------------+----------+-----+
|Diabetes_Index|prediction|count|
+--------------+----------+-----+
|           0.0|       0.0| 4949|
|           0.0|       1.0| 1514|
|           1.0|       0.0| 2118|
|           1.0|       1.0| 4003|
+--------------+----------+-----+

Execution Time: 8.224338293075562 seconds
```

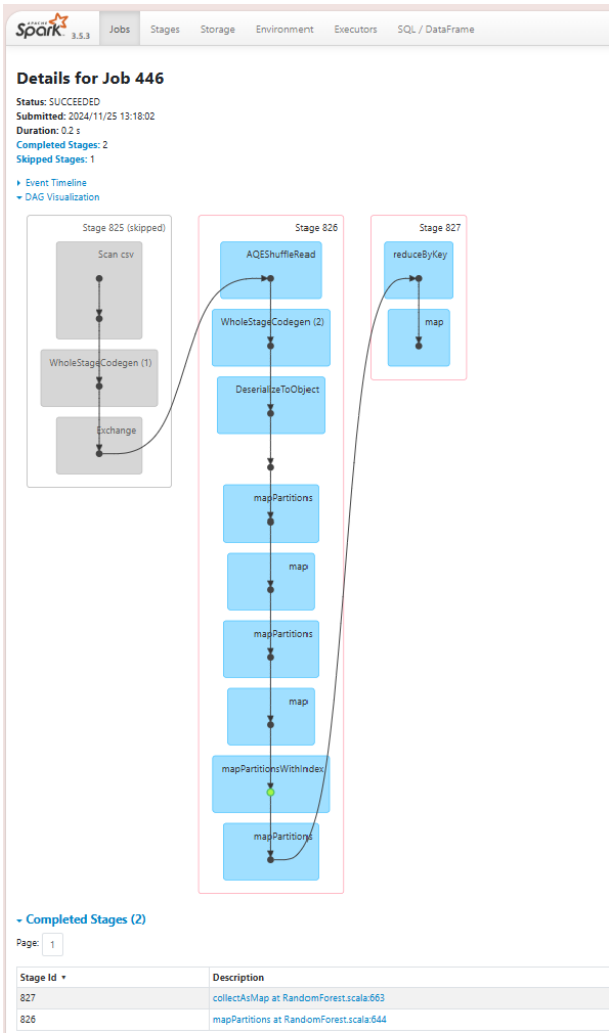
DAG Visualization:

Below are the jobs created for the implementation of Random Forest model (Job Id 436 to 451)









The above DAG shows a Spark job that transforms the data into an RDD, and applies multiple transformations. The WholeStageCodegen operator optimizes the process. The data is then shuffled, partitioned, and used to train a Random Forest model. The final stages involve building multiple decision trees and making predictions.

E. Gradient Boost Classifier

Gradient-Boosted Tree(GBT) represents one of the ensemble learning techniques in which decision trees are built sequentially, correcting the errors of the previously built tree. Due to this iteration process, it enhances the model's accuracy; hence, GBT is effective for complex tasks such as diabetes prediction. GBT improves predictive performance by combining multiple weak learners into one strong model and reduces bias and variance, which leads to more accurate and robust classification.

This implementation involved the use of a GBTClassifier within a pipeline for predicting diabetes by health features. The model performed well, yielding an accuracy of 79.4%, thus proving to be very efficient in classifying a person as diabetic or otherwise. As shown in the confusion matrix of

this model, it classified both classes properly, and was thereby well-balanced between true positives and true negatives.

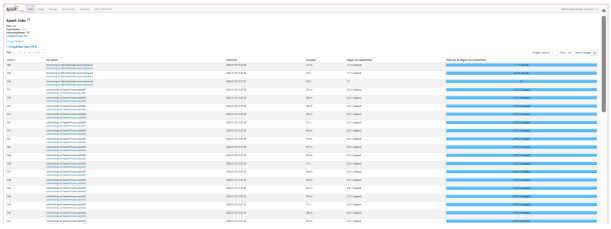
The execution time to train and predict was 11.88 seconds, somewhat longer when compared with other models; however, efficiency in the PySpark environment can still be at its best. Considering that GBT is iterative-a number of trees are built within the model-the relatively fast execution time makes this model suitable for very large-scale applications in healthcare. The GBTClassifier is a very robust classifier in terms of accuracy and efficiency. Therefore, this model would be highly suitable for diabetes prediction in real time, among other critical healthcare use cases.

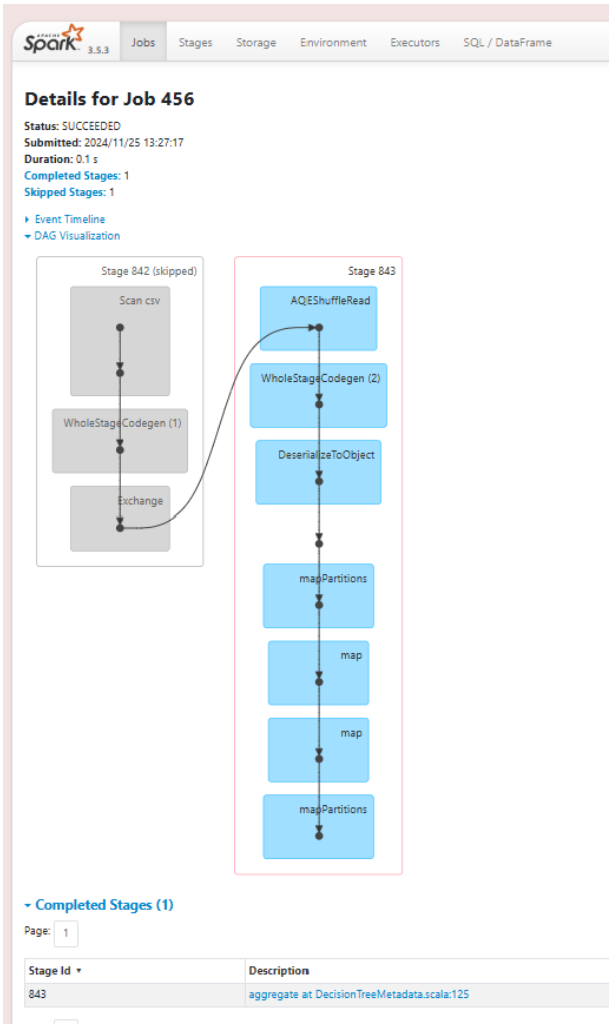
Gradient Boost Confusion matrix & Execution time:

```
Confusion Matrix:
+--------------+----------+-----+
|Diabetes_Index|prediction|count|
+--------------+----------+-----+
|           0.0|       0.0| 4944|
|           0.0|       1.0| 1519|
|           1.0|       0.0| 2045|
|           1.0|       1.0| 4076|
+--------------+----------+-----+

Execution Time: 11.884103298187256 seconds
```

DAG Visualization:

Below are the jobs created for the implementation of Gradient Boost Classifier model (Job Id 452 to 560)

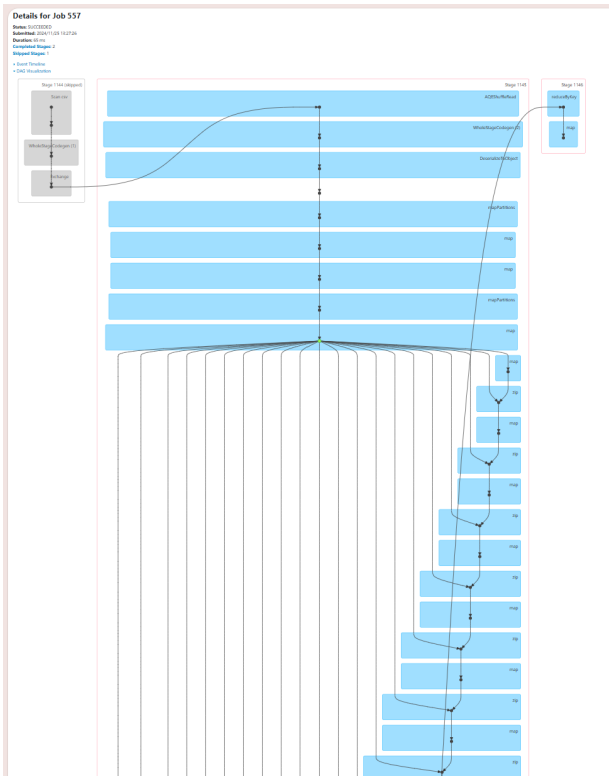




The above DAG represents the execution flow of a Spark job performing a Gradient Boosting Classifier. It first reads from other executors at the beginning of the job and then deserializes the data. Further, multiple transformations are performed on the data, partitioning, and mapping operations. WholeStageCodegen operator optimizes the execution by combining the multiple transformations into one stage.

The core of the Gradient Boosting algorithm is an iterative training of decision trees. Multiple stages are visible in the DAG for training individual trees, which involve data preparation, feature selection, and building a tree. The last stage, with high probability, combines the predictions of all trees to yield the final prediction.

This DAG provides insights about how complex and iterative the Gradient Boosting algorithm is, including how efficiently the Spark job is executed.



The above DAG shows a Spark job that transforms the data into an RDD, and applies multiple transformations. The WholeStageCodegen operator optimizes the process. The data is then shuffled, partitioned, and used to train a Random Forest model. The final stages involve building multiple decision trees and making predictions.

F. Decision Tree

The Decision Tree Classifier is a type of supervised learning algorithm that recurrently splits the dataset into subsets, based on feature values, into a node representing decision rules. This process goes until the leaves represent the final data division, therefore predictions. In diabetes prediction problems, a Decision Tree would classify someone as a diabetic or not based on health-related features. Due to simplicity and interpretability, Decision Trees are the best to go with on classification tasks.

The model was then trained using a DecisionTreeClassifier in a pipeline. From the confusion matrix, one can observe that the model is able to predict both diabetic and non-diabetic classes correctly. The model, out of simplicity, achieved an accuracy of 66.7%, offering a reasonable balance in correctly predicting both classes and avoiding overfitting.

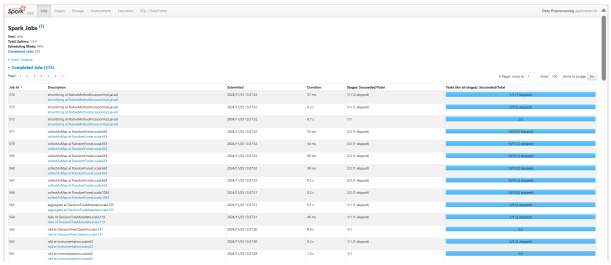
The training and prediction time was 3.94 seconds, proving its efficiency. Generally, decision trees are faster to train and predict than more complex models, such as Gradient Boosted Trees. This makes them suitable to apply in any application requiring quick decisions. Though the accuracy is somewhat lower than that of some other models, the fast execution time and simplicity of the Decision Tree Classifier make it effective in real-time diabetes prediction in resource-constrained environments.

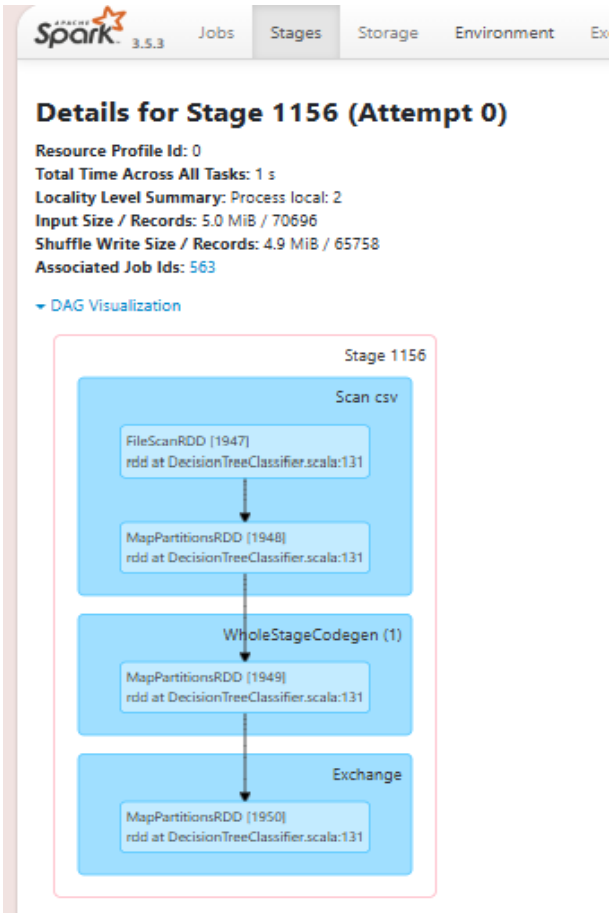
Decision Tree Confusion matrix & Execution time:

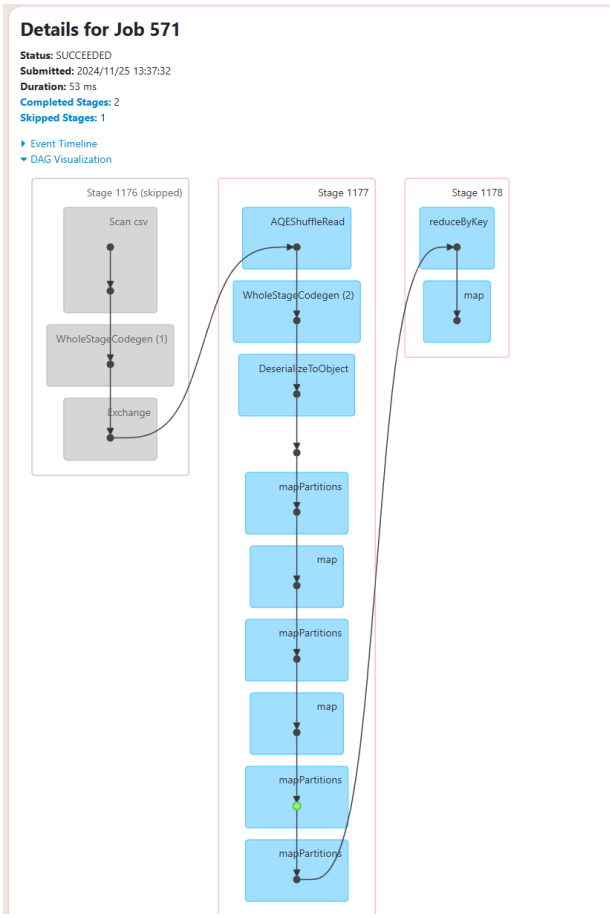
```
Confusion Matrix:
+--------------+----------+-----+
|Diabetes_Index|prediction|count|
+--------------+----------+-----+
|           0.0|       0.0| 4880|
|           0.0|       1.0| 1583|
|           1.0|       0.0| 2117|
|           1.0|       1.0| 4004|
+--------------+----------+-----+

Execution Time: 3.9417908191680911 seconds
```

DAG Visualization:

Below are the jobs created for the implementation of Decision tree model (Job Id 561 to 574)







The key in the Decision Tree algorithm lies in the process of iterative splitting on feature values of data. The DAG shows multiple steps in splitting the data to build the decision tree, and the last step most likely aggregates predictions from the decision tree to make the final prediction.

G. Multilayer Perceptron Classifier

The Multilayer Perceptron Classifier (MLP) consists of multiple layers of interconnected nodes (neurons) that learn

complex patterns in the data. Each layer transforms the input features progressively to make predictions. In this case, MLP classifies individuals based on health-related features (e.g., BMI, age, and physical activity) to predict diabetes. MLPs are especially good at capturing nonlinear relationships and are therefore ideal for complex classification tasks such as diabetes prediction.

Then, the model was trained, using a Multilayer Perceptron Classifier in a pipeline that first assembled input features into a vector and then trained the model. The confusion matrix below gives an indication of how well the model has classified both diabetic and non-diabetic individuals, and thus is a very effective prediction performance. The accuracy achieved from this model is 78.6%, hence reliable to make a difference between the classes.

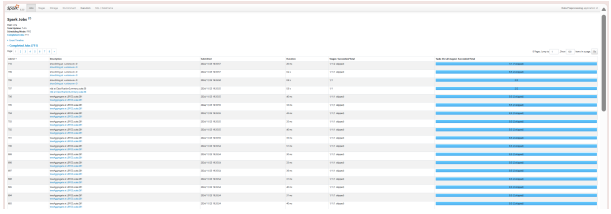
It took 16.45 seconds to execute both training and prediction, showing the efficiency of this complex model. While MLP models do take greater computational resources than simpler algorithms, they have much higher predictive power for complex datasets, and therefore are suitable in real-world applications such as diabetes prediction.

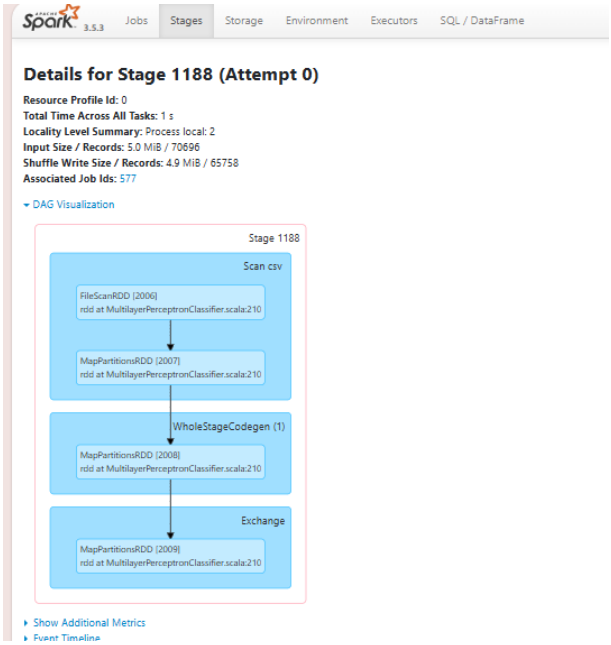
Multilayer Perceptron Classifier Confusion matrix & Execution time:

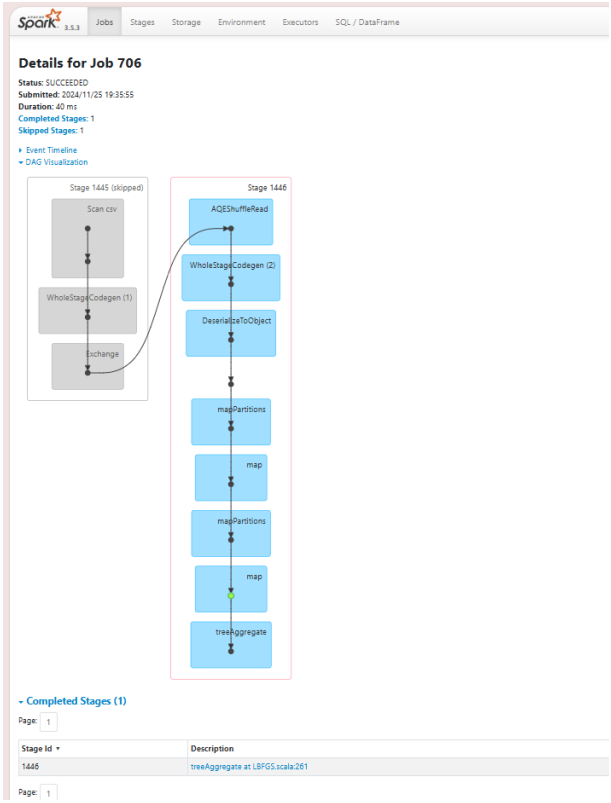
```
Confusion Matrix:
+--------------+----------+-----+
|Diabetes_Index|prediction|count|
+--------------+----------+-----+
|           0.0|       0.0| 5206|
|           0.0|       1.0| 1257|
|           1.0|       0.0| 2304|
|           1.0|       1.0| 3817|
+--------------+----------+-----+

Execution Time: 16.44677186012268 seconds
```

DAG Visualization:

Below are the jobs created for the implementation of Multilayer Perceptron model (Job Id 575 to 710)







The job reads data from the other executor using the AQEShuffleRead operator. Following that, the data is deserialized into objects and goes through many transformations, partitioning, and mapping operations. The WholeStageCodegen operator optimizes the execution of many transformations in a single stage.

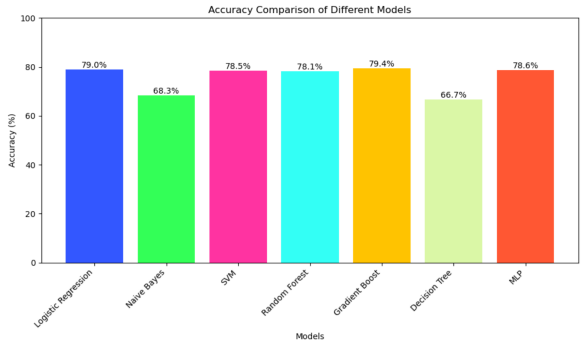
The central point of the MLP algorithm is the several layers of neurons, and each layer does some amount of computation.

The DAG shows multiple steps to process data through these layers, including activation and weight updates. The final stage likely aggregates the predictions from the output layer to make the final prediction.

V. EVALUATION

A. Accuracy

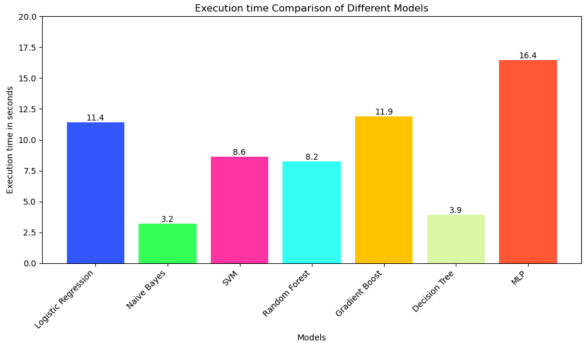
The bar graph compares the accuracy of different classification algorithms (Logistic Regression, Naive Bayes, Support Vector Machine, Random Forest, Gradient Boost Classifier, Decision Tree and Multilayer Perceptron) on a diabetes prediction problem using PySpark's MLlib.



- **Top Performers:** Logistic Regression, Support Vector Machine, Random Forest, Gradient Boosting and Multilayer Perceptron consistently achieve the highest accuracy scores of around 78% to 79%, making them strong contenders for various machine learning tasks.
- **Lower Performers:** Naive Bayes and Decision Tree appear to have lower accuracy, suggesting that they may not be the best choices for complex datasets or tasks requiring high precision.

B. Execution Time

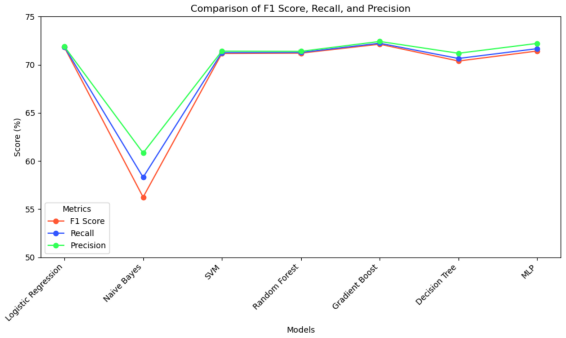
The below graph visually represents the time taken by each model in seconds to train on a diabetes dataset.



- Naive Bayes and Decision Tree are the fastest models, with execution times below 5 seconds but they didn't perform well(since accuracies are less).
- Logistic Regression and Gradient Boosting Classifiers have moderate training times, around 11 seconds.
- MLP is significantly slower, taking around 16.4 seconds or more.
- Random Forest and SVM falls in the middle, with a training time of about 8 seconds.

C. F1 Score, Precision and Recall

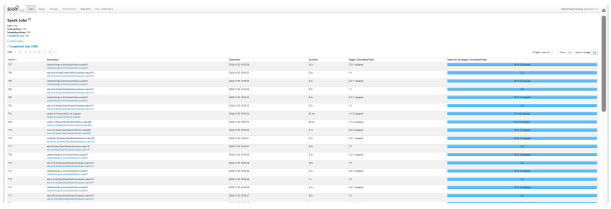
The below graph illustrates the F1 Score, recall and precision taken by each model to train on a diabetes dataset.



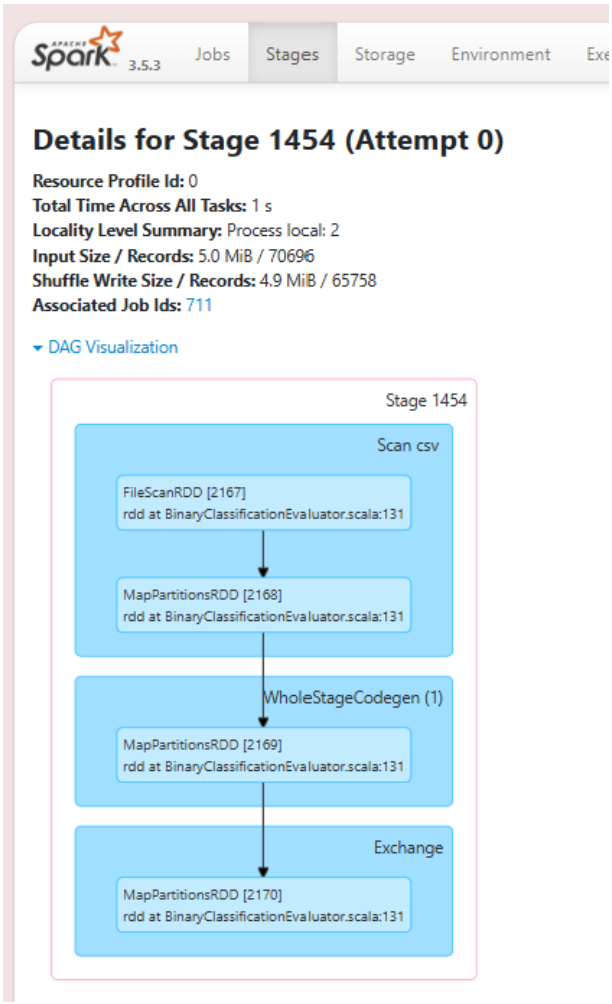
- **Random Forest**, **MLP** and **Gradient Boosting** generally exhibit strong performance across all three metrics, suggesting their ability to achieve a good balance between precision and recall.
- **Naive Bayes** and **Decision Tree** tend to have lower performance compared to other models, particularly in terms of precision.

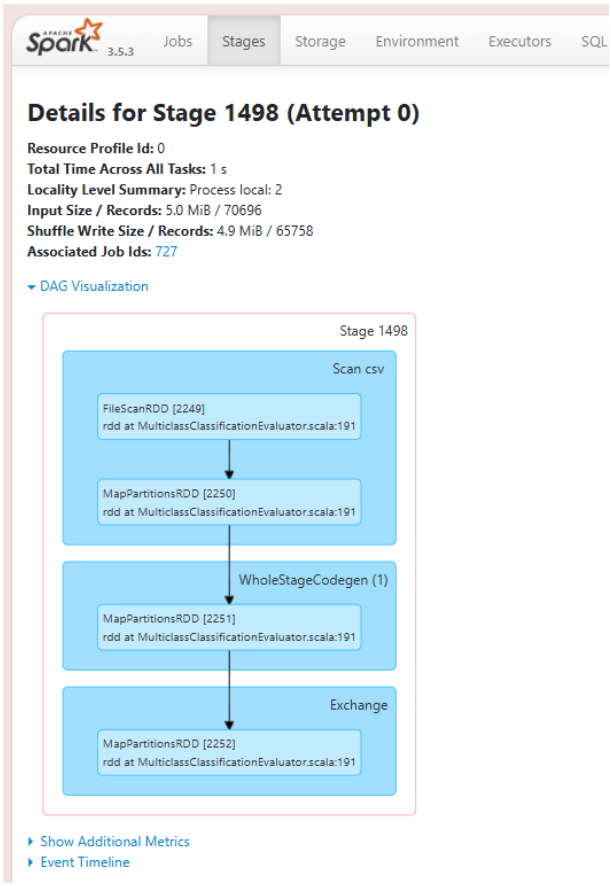
D. DAG Visualization

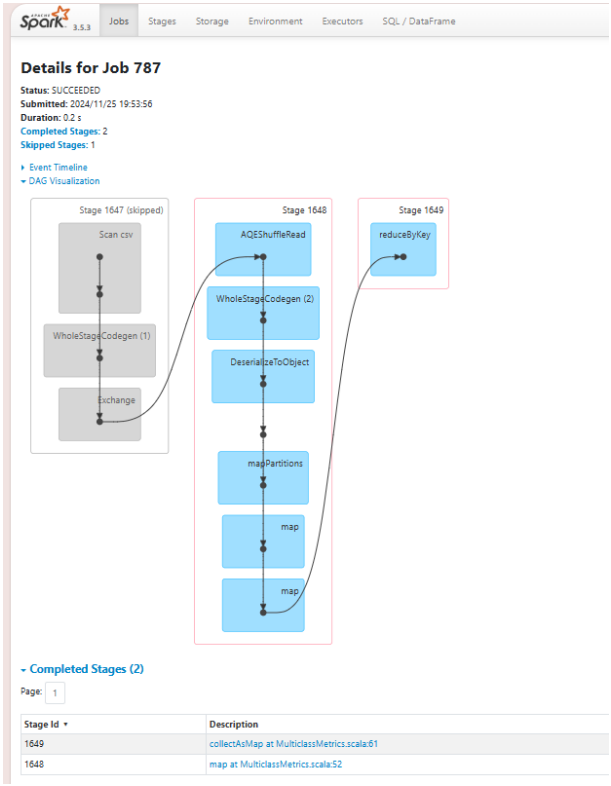
Below are the jobs created for the evaluation of models (Job Id 711 to 787),



The below stage involves the BinaryClassificationEvaluator, which calculates the accuracy metric based on the predicted labels and true labels. This DAG provides insights into the evaluation process, highlighting the data shuffling, transformation, and metric calculation steps involved in assessing the model's accuracy.







The final stage involves the MulticlassClassificationEvaluator, which calculates the weighted precision metric based on the predicted labels and true labels. This stage involves computing confusion matrices, calculating true positives, false positives, and true negatives, and then deriving the weighted precision metric. Similarly, these steps were performed for f1_score and recall.

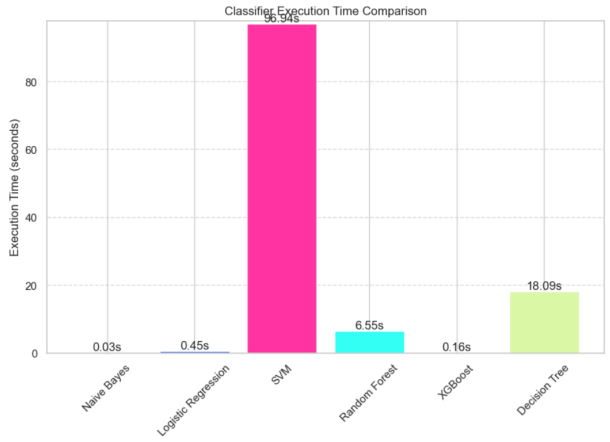
The core of the evaluation would involve aggregation across

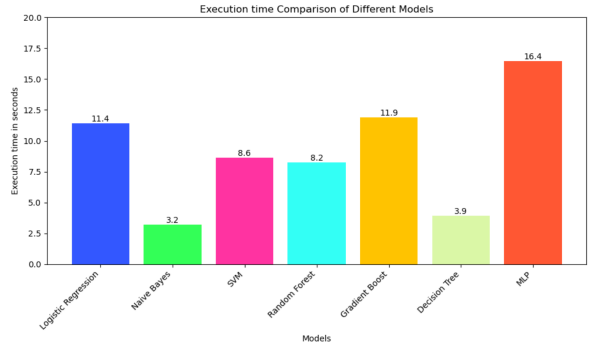
several partitions. Use the reduceByKey operator to combine results from each partition. The final stage most probably calculates overall performance metrics, either accuracy, precision, recall, or an F1-score.

This DAG thus provides insights on how this evaluation would be done, considering data shuffling, transformation, and aggregations in calculating performance.

VI. PERFORMANCE COMPARISION

A. Execution Time Comparision:





1.Naive Bayes

- phase 2 (1st image):Executes in 0.03 seconds, showcasing its efficiency for simpler models with minimal computations. Pandas performs better here as there is no overhead associated with distributed processing.
- PySpark (2nd image): Executes in 3.2 seconds, slightly slower due to the inherent setup and communication overhead in a distributed environment for such a lightweight task.
- Implication:Pandas is better suited for simple, lightweight models like Naive Bayes, where the computation is minimal, and the overhead of distributed systems becomes a disadvantage.

2.Logistic Regression

- phase 2 (1st image): Executes in 0.45 seconds, demonstrating its ability to handle medium-complexity models efficiently on small datasets.
- PySpark (2nd image): Executes in 11.4 seconds, which is slower due to the initial overhead of setting up a distributed system.
- Implication: Logistic Regression benefits more from Pandas for smaller datasets, where distributed processing adds unnecessary overhead.

3.Support Vector Machine (SVM)

- phase 2 (1st image): Executes in 96.94 seconds, demonstrating how computationally expensive this model can be when run on a single-threaded system for large datasets.
- PySpark (2nd image):Executes in 8.6 seconds, significantly faster due to its ability to distribute and parallelize the workload.
- Implication:SVM heavily benefits from PySpark's distributed architecture, as its iterative nature and high computational demands make it a bottleneck in single-threaded systems.

4.Random Forest

- phase 2 (1st image): Executes in 6.55 seconds, which is relatively efficient due to its parallelizable structure.
- PySpark (2nd image):Executes in 8.2 seconds, showing comparable performance to Pandas, as Random Forest inherently benefits less from distributed systems for smaller datasets.
- Implication: For Random Forest, the execution times between Pandas and PySpark are similar, but PySpark would likely scale better for larger datasets due to its distributed nature.

5.Gradient Boost (XGBoost)

- phase 2 (1st image): Executes in 0.16 seconds, showing the efficiency of pandas for relatively smaller datasets and models like Random Forest.
- PySpark (2nd image):Executes in 11.9 seconds, slower compared to pandas due to the overhead of distributed processing in a setup that doesn't significantly benefit from parallelization for small models.
- Implication: For small datasets or simpler models, pandas performs better. However, PySpark's performance would scale better with larger datasets, where its distributed nature would significantly improve execution times.

6.Decision Tree

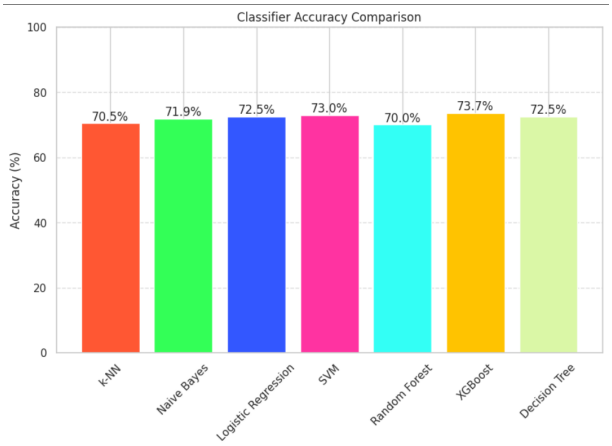
- phase 2 (1st image):runs in 18.09 seconds, showing the computational complexity of Decision Trees on large datasets in a single-threaded environment.
- PySpark (2nd image):executes in 3.9 seconds, significantly faster due to its distributed processing and parallelization of tasks like splitting data and building subtrees.
- Implication: While pandas works for smaller datasets, PySpark greatly enhances performance on larger datasets due to its distributed nature, making it a better choice for scalability.

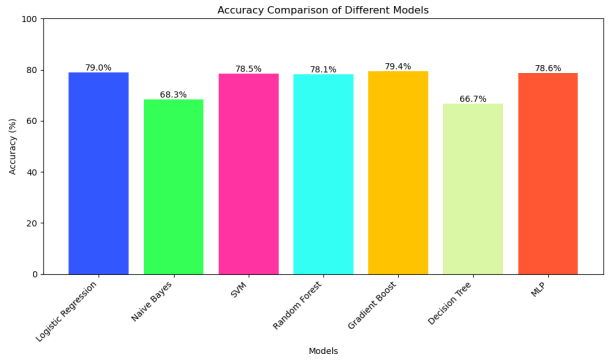
Observation

- Pandas is better suited for small-scale, lightweight models where the overhead of distributed processing is not justified.

- PySpark excels in scaling computationally intensive models on larger datasets, offering significant execution time improvements due to its parallelism and distributed nature.

B. Accuracy Comparision:





1.Naive Bayes

- phase 2 (1st image):Accuracy is 71.9%, reflecting good efficiency for a lightweight algorithm.
- PySpark (2nd image):Accuracy is 68.3%, slightly lower, likely due to distributed processing overhead for such a simple model.
- Implication: Pandas performs better for lightweight algorithms like Naive Bayes where computation is minimal, and distributed systems add unnecessary complexity.

2.Logistic Regression

- phase 2 (1st image): Accuracy is 72.5%, showing its effectiveness for medium-complexity models.
- PySpark (2nd image): Accuracy is 79.0%, significantly higher, possibly due to better optimization in PySpark for larger or more complex datasets.
- Implication: PySpark's distributed processing significantly benefits Logistic Regression on larger datasets, whereas Pandas works well for smaller ones.

3.Support Vector Machine (SVM)

- phase 2 (1st image): Accuracy is 73.0%, indicating good performance for smaller datasets despite SVM's computational cost.
- PySpark (2nd image):Accuracy is 78.5%, higher than Pandas, leveraging PySpark's parallel processing capabilities.
- Implication:SVM benefits significantly from PySpark's distributed architecture due to its high computational demands.

4.Random Forest

- phase 2 (1st image):Accuracy is 70.0%, reflecting reasonable performance on a smaller dataset.
- PySpark (2nd image): Accuracy is 78.1%, demonstrating better performance due to PySpark's ability to handle larger datasets efficiently.
- Implication: PySpark outperforms Pandas for Random Forest on larger datasets, even though Pandas handles small datasets adequately.

5.Gradient Boost (XGBoost)

- phase 2 (1st image): Accuracy is 73.7%, showing excellent efficiency for smaller datasets.
- PySpark (2nd image):Accuracy is 79.4%, higher due to better scaling and optimization for larger datasets.
- Implication:PySpark scales better with XGBoost, especially for larger datasets, while Pandas is sufficient for smaller tasks.

6.Decision Tree

- phase 2 (1st image):Accuracy is 72.5%, reflecting decent performance without parallelism.
- PySpark (2nd image)Accuracy is 66.7%, lower, possibly due to less optimization in PySpark for this specific algorithm.
- Implication: Pandas may be better suited for Decision Trees in smaller datasets, as PySpark does not offer significant advantages here.

Observation

- For lightweight algorithms like Naive Bayes and Decision Tree, Pandas is more efficient due to minimal computation and no setup overhead.
- For computationally intensive models like SVM, Random Forest, and XGBoost, PySpark excels, leveraging its distributed architecture to handle large datasets more effectively.

COMPARISON OF METRICS (F1 SCORE, PRECISION, RECALL)

1. Naive Bayes

- **Pandas (Phase 2):** Scores are close to each other ($\sim$ 71%), reflecting consistency and good performance.
- **PySpark:** F1 Score, Precision, and Recall dip significantly ($\sim$ 68%), indicating performance degradation in PySpark due to overheads for such a lightweight algorithm.
- **Implication:** Pandas outperforms PySpark in terms of all three metrics for Naive Bayes on smaller datasets.

2. Logistic Regression

- **Pandas (Phase 2):** All three metrics cluster around $\sim 72.5\%$, indicating balanced performance.
- **PySpark:** The metrics are much higher ($\sim 79\%$), showcasing improved performance due to PySpark's distributed optimization.
- **Implication:** Logistic Regression benefits greatly from PySpark's distributed architecture for larger datasets, showing clear improvements across F1 Score, Precision, and Recall.

3. Support Vector Machine (SVM)

- **Pandas (Phase 2):** Scores are consistent ($\sim 73\%$), showing effective handling for smaller datasets despite computational demands.
- **PySpark:** Metrics are significantly higher ($\sim 78\%$), demonstrating PySpark's ability to optimize resource-intensive algorithms like SVM.
- **Implication:** PySpark's parallel processing gives it a distinct advantage for SVM, leading to higher F1 Score, Precision, and Recall.

4. Random Forest

- **Pandas (Phase 2):** Metrics cluster around 70%, reflecting stable but unimpressive performance on smaller datasets.
- **PySpark:** Scores are higher ($\sim 78.5\%$), indicating better optimization and resource handling.
- **Implication:** Random Forest clearly benefits from PySpark's distributed system for larger datasets, resulting in superior F1 Score, Precision, and Recall.

5. XGBoost (Gradient Boost)

- **Pandas (Phase 2):** Metrics are high ($\sim 73.7\%$), showcasing strong performance on smaller datasets.
- **PySpark:** Scores are slightly higher ($\sim 79.4\%$), leveraging distributed optimization.
- **Implication:** Both frameworks handle XGBoost well, but PySpark has a distinct edge with larger datasets, providing consistently higher metrics.

6. Decision Tree

- **Pandas (Phase 2):** Scores hover around $\sim 72.5\%$, indicating strong performance without requiring distributed resources.
- **PySpark:** Metrics drop ($\sim 66.7\%$), suggesting reduced optimization for Decision Tree in PySpark.
- **Implication:** Pandas is preferable for Decision Tree algorithms, particularly for smaller datasets, due to PySpark's inefficiencies.

Effectiveness and advantages of using PySpark for distributed processing on large datasets

- **Scalability and Parallelism:** PySpark efficiently distributes data and computations, ensuring consistent performance as dataset size grows. Models like SVM and Decision Tree show significant reductions in execution time due to parallel processing.
- **Improved Execution Time for Large Datasets:** While PySpark may introduce overhead for small datasets, it excels in processing large and complex models, as seen in Random Forest and Gradient Boost.
- **Enhanced Model Performance:** PySpark often achieves higher accuracy on large datasets due to its optimization capabilities. Models like Logistic Regression and XGBoost show notable improvements in metrics like accuracy, precision, and recall.
- **Resource Efficiency:** By utilizing distributed memory and processing, PySpark handles high-dimensional and resource-intensive data better than single-threaded frameworks like Pandas.
- **Suitability for Complex Algorithms:** PySpark is particularly effective for computation-heavy models, reducing time while maintaining or improving accuracy, as shown in SVM and Random Forest comparisons.
- **Conclusion:** PySpark's distributed processing capabilities make it an ideal choice for large datasets and complex algorithms, providing faster execution, better scalability, and improved accuracy. It outperforms Pandas in scenarios where dataset size and computational demands are high.

VII. BONUS: WEB APPLICATION

Setting Up the Backend

1) Navigate to the `backend` directory:
```
cd backend
```

2) Start the Flask API server:
```
python app.py
```

The server will start by default at **http://localhost:5000**.

Setting Up the Frontend

1) Open a new terminal and navigate to the `frontend` directory:
```
cd frontend
```

2) Install the dependencies using npm or yarn:
```
npm install
\# OR
yarn
```

3) Start the React development server:
```
npm start
\# OR
yarn start
```

The frontend will run by default on **http://localhost:3000**.

a. Showing a working user interface (not just code you would expect a user to run)

Our Diabetes Risk Analyzer application provides a complete functional user interface built using React in the frontend and

Flask in the backend. The user will be able to interact with the application through an extremely simple-to-use UI.

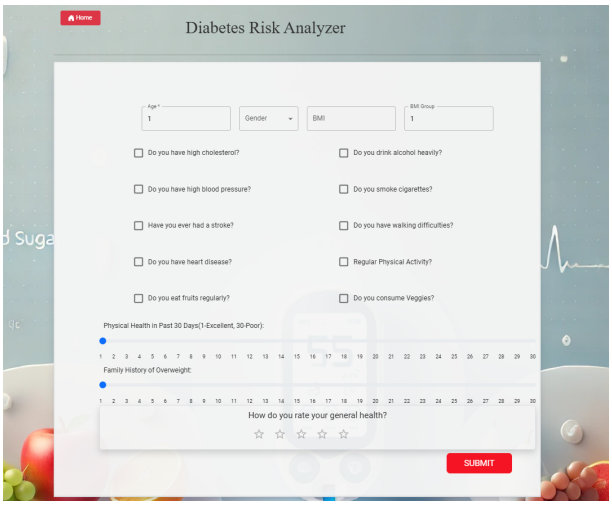
The application includes:

- Homepage: A landing page with options to proceed to the input form or view analytics.
- Input Form: A form for users to enter health-related data.
- Feedback and Analytics Pages: These pages display the predicted results, relevant recommendations, and visualizations based on user input.

The application ensures easy interaction and a step-by-step guide through which the user can assess their diabetes risk.

b. Showing how a user could input their own data

Users can input their health data directly into the application through a form-based GUI that includes the following:



- Fields for numerical input (e.g., age, BMI).
- Dropdowns to select categorical data such as gender.
- Toggles and checkboxes for binary data, including smoking habits or physical activity. for rating health conditions, such as physical and mental health.

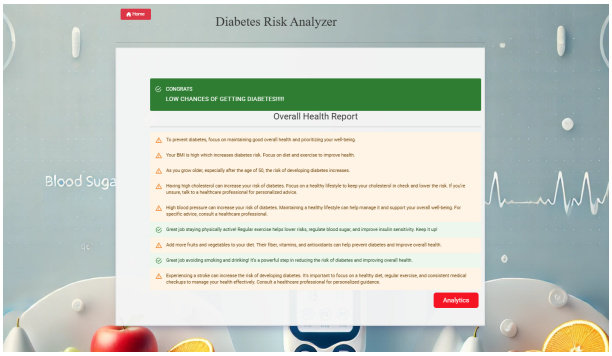
Backend integration to send the form data for predictions whenever users fill out the fields and click on the submit button.The data gets sent back via a POST request to the backend where it can then be processed and return a prediction.

Here, we are using a pickle file generated during the phase-2, where we trained our models on the dataset. Since the XGBoost model achieved the highest accuracy, we utilized it to create the pickle file. This pickle file is then loaded into our backend Flask application which helps us to give predictions based on the user given data.

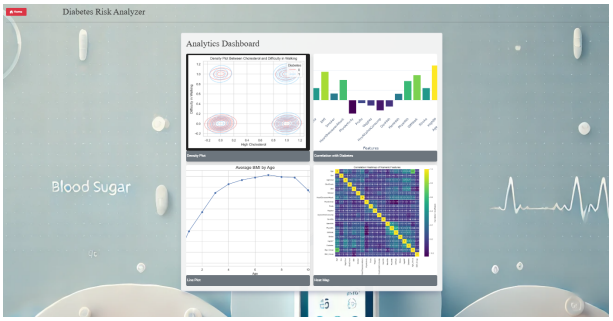
```
try:
    with open('xgboost_sklearn', 'rb') as model_file:
        xgboost_model = pickle.load(model_file)
except FileNotFoundError:
    raise FileNotFoundError("Model file 'xgboost_sklearn' not found.")
```

c. for showing the feedback your product gives, explaining what it means, relevant manipulation/filtering of visualizations, and how a user could use it to help them solve a problem/answer a question.

The application provides a general feedback, which includes:



- Prediction Results: According to the user input, the app predicts whether the user is at high or low risk for diabetes for example:
  - WARNING: ”High chances of getting diabetes.”
  - CONGRATS: ”Low chances of getting diabetes.”
- Recommendations: Customized advice regarding user’s data, like:
  - Maintaining a healthy BMI.
  - Keeping cholesterol and blood pressure in a normal range.
  - Keeping physically active and avoiding harmful habits such as smoking or alcohol consumption.
- Analytics: The App also provides the following visualizations
  - Correlation charts showing factors contributing to diabetes risk.
  - Heatmaps and density plots for visual analysis of health attributes.



The Diabetes Risk Analyzer is helpful in tackling the following challenges:

- Diabetes risk detection at an early stage ensures timely prevention and management
- The system can supplement the advice given by health experts, thus providing better care.
- Can provide users with insights so they can make healthier choices with respect to lifestyle and diet.
- Can help users track their improvements on health and modify their schedule accordingly.
- Can encourage users to lead sustainable health practices, thereby offering regular exercises and balanced nutrition.

REFERENCES

[1] Apache Spark. *Binomial Logistic Regression*. https://spark.apache.org/docs/latest/ml-classification-regression.html#binomial-logistic-regression.
[2] Apache Spark. *Naive Bayes*. https://spark.apache.org/docs/latest/ml-classification-regression.html#naive-bayes.
[3] Apache Spark. *Support Vector Machine*. https://spark.apache.org/docs/latest/ml-classification-regression.html#linear-support-vector-machine.
[4] Apache Spark. *Random Forest Classifier*. https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier.
[5] Apache Spark. *Gradient Boost Classifier*. https://spark.apache.org/docs/latest/ml-classification-regression.html#gradient-boosted-tree-classifier.
[6] Apache Spark. *Decision Tree*. https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-tree-classifier.
[7] Apache Spark. *Multilayer Perceptron*. https://spark.apache.org/docs/latest/ml-classification-regression.html#multilayer-perceptron-classifier.
[8] Apache Spark. *BinaryClassificationEvaluator*. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.evaluation.BinaryClassificationEvaluator.html.
[9] Apache Spark. *MulticlassClassificationEvaluator*. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.evaluation.MulticlassClassificationEvaluator.html.