Name: Emmanuel Akama

Student ID: S2229758

Title: ML pipeline for Diabetes Prediction Analysis

Background

The objective of this report is to build a prototype machine learning (ML) data pipeline for diabetes prediction analysis using an ML data pipeline that stitches together the end-to-end operation consisting of capturing the data, transforming it into insights, training a model, delivering insights, applying the model whenever and wherever the action needs to be taken to achieve the business goal.

The diabetes prediction analysis data model will be based on the diabetes prediction dataset from Kaggle available at https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset/data (https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset/data). In addition to the main diabetes prediction dataset, we will incorporate a mock-up JSON data file to perform some data transformations within the ML data pipeline.

The diabetes prediction dataset is a collection of medical and demographic data from patients, along with their *diabetes* status (positive or negative). The data includes features such as *age*, *gender*, *body mass index (BMI)*, *hypertension*, *heart disease*, *smoking history*, *HbA1c level*, and *blood glucose level*. The dataset provides a good use-case to build machine learning models to predict diabetes in patients based on their medical history and demographic information. This can be useful for healthcare professionals in identifying patients who may be at risk of developing diabetes and in developing personalized treatment plans.

A typical big data pipeline has five stages:

- *Data capture* from internal and external data sources such as mobile apps, websites, web apps, microservices and IoT devices.
- *Data ingestion* through batch jobs or stream sources (HTTP, MQTT, message queue, etc). For our use-case, we will import the data from the Databricks file system (DBFS).
- *Data storage* in Data Lakes and Data Warehouses. For our use-case, we will store the data in temporary tables in the Databricks file system (DBFS).
- *Compute* for data analysis and modelling tasks.
- *Use case* such as ML data pipeline for a diabetes prediction analysis

Data Capture, Ingestion & Storage

This covers the first three (3) stages of our ML data pipeline. The storage management for data capture and ingestion will be done on DBFS.

To facilitate storage management with DBFS, we will import the diabetes prediction datasset using the upload option in Databricks. After files are upload, it is customary to confirm that uploaded files are in the designated path.

Step 1 - Load the dataset

We will use Databricks utilities to read the data.

```
# show contents of DBFS file store
display(dbutils.fs.ls("/FileStore/tables/CW02/"))
```

Table

	path	name	size	modificationTime
1	dbfs:/FileStore/tables/CW02/diabetes_prediction_dataset.csv	diabetes_prediction_dataset.csv	3810356	1714659158000
2	dbfs:/FileStore/tables/CW02/diabetes_prediction_dataset_parquet/	diabetes_prediction_dataset_parquet/	0	0

2 rows

```
# read the data
df = spark.read \
.options(delimiter=',', header="true") \
.csv("/FileStore/tables/CW02/diabetes_prediction_dataset.csv")

# preview the data
df.show(10)
```

```
+------+----+------------+-------------+---------------+-----+-----------+-------------------+--------+
|gender| age|hypertension|heart_disease|smoking_history|  bmi|HbA1c_level|blood_glucose_level|diabetes|
+------+----+------------+-------------+---------------+-----+-----------+-------------------+--------+
|Female|80.0|           0|            1|          never|25.19|        6.6|                140|       0|
|Female|54.0|           0|            0|        No Info|27.32|        6.6|                 80|       0|
|  Male|28.0|           0|            0|          never|27.32|        5.7|                158|       0|
|Female|36.0|           0|            0|        current|23.45|        5.0|                155|       0|
|  Male|76.0|           1|            1|        current|20.14|        4.8|                155|       0|
|Female|20.0|           0|            0|          never|27.32|        6.6|                 85|       0|
|Female|44.0|           0|            0|          never|19.31|        6.5|                200|       1|
|Female|79.0|           0|            0|        No Info|23.86|        5.7|                 85|       0|
|  Male|42.0|           0|            0|          never|33.64|        4.8|                145|       0|
|Female|32.0|           0|            0|          never|27.32|        5.0|                100|       0|
+------+----+------------+-------------+---------------+-----+-----------+-------------------+--------+
only showing top 10 rows
```

```
# view the new schema
df.printSchema()
```

```
root
 |-- gender: string (nullable = true)
 |-- age: string (nullable = true)
 |-- hypertension: string (nullable = true)
 |-- heart_disease: string (nullable = true)
```

```
 |-- smoking_history: string (nullable = true)
 |-- bmi: string (nullable = true)
 |-- HbA1c_level: string (nullable = true)
 |-- blood_glucose_level: string (nullable = true)
 |-- diabetes: string (nullable = true)
```

From the data preview, it is clear that the data wasn't formatted using the correct data types. This could be based on the assumption that the data was collected from an unstructured data source.

We will perform some clean up on the data by modifiying it's data schema. Next, we will save the dataset again to the DBFS.

```
# define a new schema based on the correct data types
schema = """`gender` STRING,
    `age` DOUBLE,
    `hypertension` INT,
    `heart_disease` INT,
    `smoking_history` STRING,
    `bmi` DOUBLE,
    `HbA1c_level` DOUBLE,
    `blood_glucose_level` INT,
    `diabetes` STRING"""

# read the CSV file with schema
df = spark.read.format("csv") \
    .option("header", "true") \
    .schema(schema) \
    .load("/FileStore/tables/CW02/diabetes_prediction_dataset.csv")

# preview the data
df.printSchema()
```

```
root
 |-- gender: string (nullable = true)
 |-- age: double (nullable = true)
 |-- hypertension: integer (nullable = true)
 |-- heart_disease: integer (nullable = true)
 |-- smoking_history: string (nullable = true)
 |-- bmi: double (nullable = true)
 |-- HbA1c_level: double (nullable = true)
 |-- blood_glucose_level: integer (nullable = true)
 |-- diabetes: string (nullable = true)
```

The schema has been applied to the dataset.

Now, let's store it in a parquet directory

```
# write the parquet to DBFS
df.write.parquet("/FileStore/tables/CW02/diabetes_prediction_dataset_parquet")
display(dbutils.fs.ls("/FileStore/tables/CW02/diabetes_prediction_dataset_parquet"))
```

Table

	path	name
1	dbfs:/FileStore/tables/CW02/diabetes_prediction_dataset_parquet/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/CW02/diabetes_prediction_dataset_parquet/_committed_7259385049128777271	_committed_7259385049128777271
3	dbfs:/FileStore/tables/CW02/diabetes_prediction_dataset_parquet/_started_7259385049128777271	_started_7259385049128777271
4	dbfs:/FileStore/tables/CW02/diabetes_prediction_dataset_parquet/part-00000-tid-7259385049128777271-9644c644-80e5-423a-874e-510608a5c1fa-767-1-c000.snappy.parquet	part-00000-tid-7259385049128777271-9644c644-80e5-423a-874e-510608a5c1fa-767

4 rows

We will read the data again to simulate progression along the data pipeline

```
# read the data again
df = spark.read.parquet("/FileStore/tables/CW02/diabetes_prediction_dataset_parquet")

# view the first 5 rows
df.show(5)

# also, print the new schema
df.printSchema()
```

```
|gender| age|hypertension|heart_disease|smoking_history|  bmi|HbA1c_level|blood_glucose_level|diabetes|
+------+----+------------+-------------+---------------+-----+-----------+-------------------+--------+
|Female|80.0|           0|            1|          never|25.19|        6.6|                140|       0|
|Female|54.0|           0|            0|        No Info|27.32|        6.6|                 80|       0|
|  Male|28.0|           0|            0|          never|27.32|        5.7|                158|       0|
|Female|36.0|           0|            0|        current|23.45|        5.0|                155|       0|
|  Male|76.0|           1|            1|        current|20.14|        4.8|                155|       0|
+------+----+------------+-------------+---------------+-----+-----------+-------------------+--------+
only showing top 5 rows

root
 |-- gender: string (nullable = true)
 |-- age: double (nullable = true)
 |-- hypertension: integer (nullable = true)
 |-- heart_disease: integer (nullable = true)
 |-- smoking_history: string (nullable = true)
 |-- bmi: double (nullable = true)
 |-- HbA1c_level: double (nullable = true)
 |-- blood_glucose_level: integer (nullable = true)
 |-- diabetes: string (nullable = true)
```

Next, we will store the data in a temporary view, so we can run queries for further analysis

```
# create a temporary view 'diabetes_prediction'
df.createOrReplaceTempView("diabetes_prediction")
```

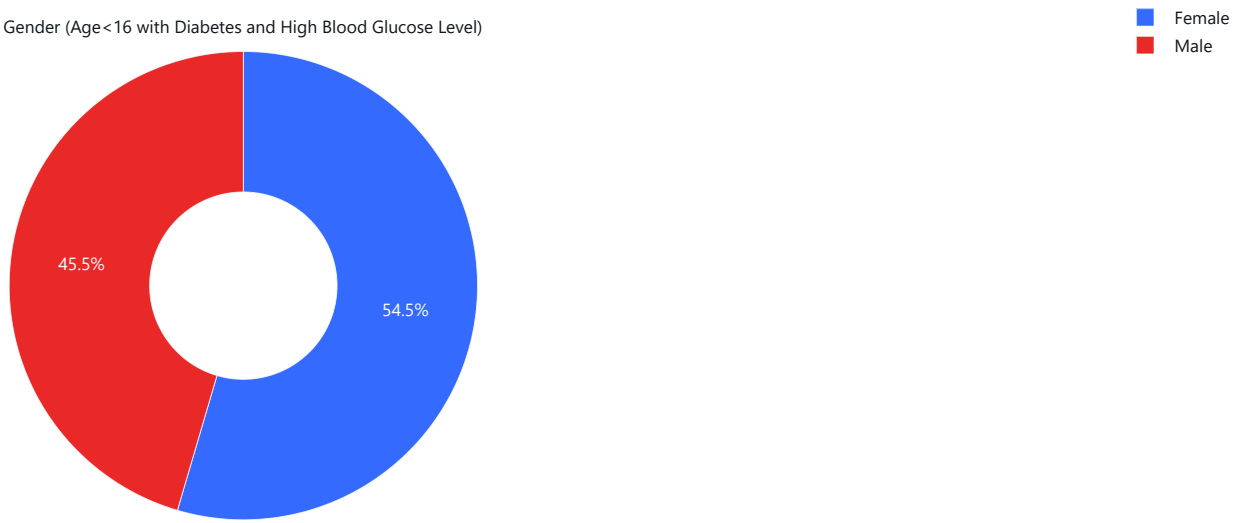
Query 1

Let's query the dataset to find out the gender that has the highest positive status for diabetes with blood glucose level greater or equal to 200 and age less than 16

```
%sql

SELECT gender, count(*) AS count
FROM diabetes_prediction WHERE (age < 16 AND blood_glucose_level >= 200 AND diabetes == 1)
GROUP BY gender
ORDER BY gender
```

Table | Gender (Age<16 with Diabetes and High Blood Glucose Level)



2 rows

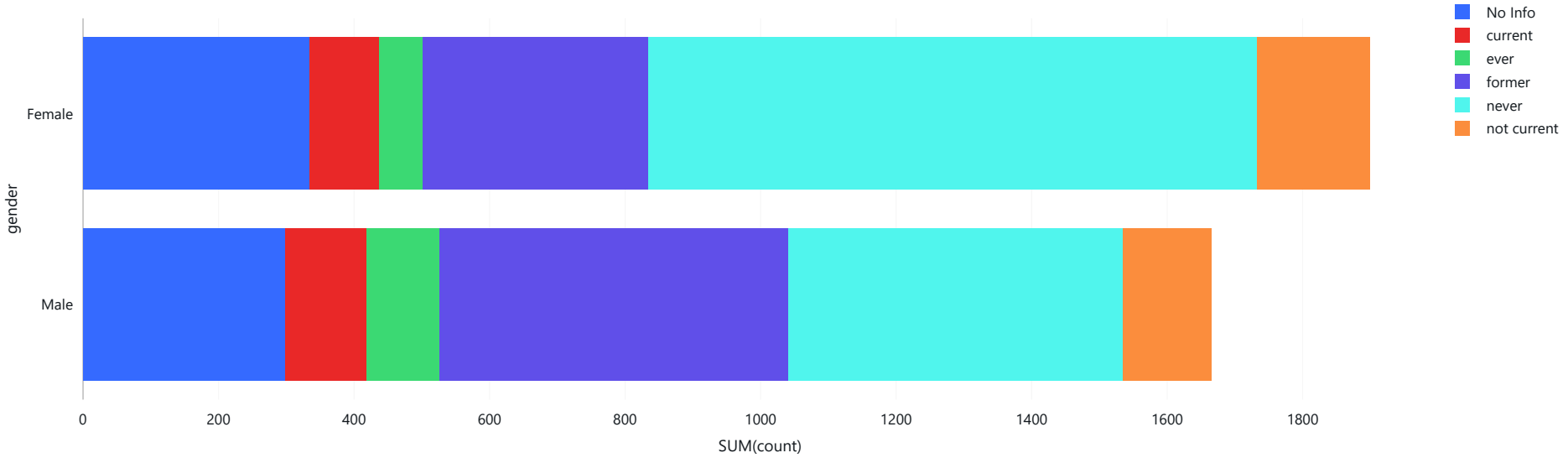
Query 2

Next, let's query the dataset to find out the gender (with smoking history) that has the highest positive status for diabetes when their age is 65 or higher

```
%sql

SELECT gender, smoking_history, count(*) AS count
FROM diabetes_prediction WHERE (age > 65 AND diabetes == 1)
GROUP BY gender, smoking_history
ORDER BY gender
```

Table | Gender (Age>65 with Diabetes vs Smoking History)



12 rows

```
# query to retrieve patients (age > 65) with diabetes
df2 = df.filter("age > 65" and "diabetes == 1").select(["gender", "smoking_history"]).groupBy(["gender", "smoking_history"]).count()
df2.show()
```

```
+------+---------------+-----+
|gender|smoking_history|count|
+------+---------------+-----+
|  Male|         former|  941|
|Female|          ever|  211|
|Female|         former|  649|
|Female|        No Info|  754|
|  Male|          ever|  261|
|  Male|        No Info|  700|
|Female|          never| 2002|
|  Male|        current|  497|
|Female|    not current|  394|
|Female|        current|  451|
|  Male|    not current|  296|
|  Male|          never| 1344|
```

```
+------+---------------+-----+
```

From the above preliminary analysis, we see that 'females' are more susceptible to testing postive to diabetes

Data Modeling & Analysis

To create the diabetes prediction analysis model, we will need training and testing datasets.

In the following cell, we will split the dataset into training and testing sets, and set a seed for reproducibility. It is necessary to split the data before performing further preprocessing to allow for better simulations with new data when evaluating the model

```
  # randomly split the data into training and testing sets
  train_df, test_df = df.randomSplit([0.8, 0.2], seed=42)
  print(train_df.cache().count()) # cache because accessing training data multiple times
  print(test_df.count())
```

```
79901
20099
```

```
  # preview the training and testing data
  train_df.show(10)
  test_df.show(10)
```

```
|Female|0.08|           0|           0|        No Info|14.43|        6.5|                160|       0|
|Female|0.08|           0|           0|        No Info|27.32|        3.5|                130|       0|
+------+----+------------+------------+---------------+-----+-----------+-------------------+--------+
only showing top 10 rows

+------+----+------------+------------+---------------+-----+-----------+-------------------+--------+
|gender| age|hypertension|heart_disease|smoking_history|  bmi|HbA1c_level|blood_glucose_level|diabetes|
+------+----+------------+------------+---------------+-----+-----------+-------------------+--------+
|Female|0.08|           0|           0|        No Info| 12.5|        4.5|                155|       0|
|Female|0.08|           0|           0|        No Info|13.03|        6.2|                158|       0|
|Female|0.08|           0|           0|        No Info|13.39|        6.1|                 90|       0|
|Female|0.08|           0|           0|        No Info|27.32|        4.5|                160|       0|
|Female|0.16|           0|           0|        No Info| 12.3|        5.8|                140|       0|
|Female|0.16|           0|           0|        No Info|13.01|        5.8|                 90|       0|
|Female|0.16|           0|           0|        No Info| 14.4|        4.8|                 85|       0|
|Female|0.16|           0|           0|        No Info|16.95|        5.8|                159|       0|
|Female|0.24|           0|           0|        No Info|12.88|        6.6|                 85|       0|
|Female|0.24|           0|           0|        No Info|12.96|        4.5|                130|       0|
+------+----+------------+------------+---------------+-----+-----------+-------------------+--------+
only showing top 10 rows
```

Step 2 - Feature Preprocessing

The goal of this notebook is to build a diabetes prediction model that predicts the status of a patient from features contained in the dataset. To create the model, we first need to manipulate (or preprocess) the features so they are in the required MLlib format.

Convert categorical variables to numeric

Some machine learning algorithms, such as linear and logistic regression, require numeric features. The diabetes prediction dataset includes categorical features such as gender, and smoking_history.

In the following code block, we will use *StringIndexer* and *OneHotEncoder* to convert categorical variables into a setof numeric variables that only take on values 0 and 1.

StringIndexer converts a column of string values to a column of label indexes. For example, it might convert the values"red", "blue", and "green" to 0, 1, and 2. *OneHotEncoder* maps a column of category indices to a column of binary vectors, with at most one "1" in each row thatindicates the category index for that row.

```
# import the required MLlib packages
from pyspark.ml.feature import StringIndexer, OneHotEncoder

# define the categorical columns
categoricalCols = ["gender", "smoking_history"]

stringIndexer=StringIndexer(inputCols=categoricalCols, outputCols=[x + "Index" for x in categoricalCols])
encoder=OneHotEncoder(inputCols=stringIndexer.getOutputCols(), outputCols=[x + "OHE" for x in categoricalCols])

# NB: The label column ("diabetes") is also a string value - it has two possible values, "0" or "1".
# convert it to a numeric value using StringIndexer.
labelToIndex=StringIndexer(inputCol="diabetes",outputCol="label")
```

Combine all feature columns into a single feature vector

Most MLlib algorithms require a single features column as input. Each row in this column contains a vector of data points corresponding to the set of features used for prediction. MLlib provides the *VectorAssembler* transformer to create a single vector column from a list of columns. For our use case, this includes both the numeric columns and the one-hot encoded binary vector columns in our dataset.

```
# import the required MLlib package
from pyspark.ml.feature import VectorAssembler

# include both the numeric columns and the one-hot encoded binary vector columns in our dataset.
numericCols = ["age", "hypertension", "heart_disease", "bmi", "HbA1c_level", "blood_glucose_level"]
assemblerInputs = [c + "OHE" for c in categoricalCols] + numericCols

vecAssembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
```

Step 3 - Define the model

This notebook uses a logistic regression model.

```
# import the required MLlib package
from pyspark.ml.classification import LogisticRegression

# create the model
lr = LogisticRegression(featuresCol="features", labelCol="label", regParam=1.0)
```

Step 4 - Build the pipeline

A Pipeline is an ordered list of transformers and estimators.

In the following code block, we will define a pipeline to automate the transformations to be applied to a dataset. In this step, we define the pipeline and then apply it to the test dataset.

```
# import the required MLlib package
from pyspark.ml import Pipeline

# define the pipeline based on the stages created in previous steps.
pipeline = Pipeline(stages=[stringIndexer, encoder, labelToIndex, vecAssembler, lr])

# define the pipeline model
pipelineModel = pipeline.fit(train_df)
pred_df = pipelineModel.transform(test_df)
```

Step 5 - Evaluate the model

To evaluate the model, we use the *BinaryClassificationEvaluator* to evalute the area under the ROC curve and *MulticlassClassificationEvaluator* to evalute the accuracy.

For further reference, see

- BinaryClassificationEvaluator (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.evaluation.BinaryClassificationEvaluator (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.evaluation.BinaryClassificationEvaluator))
- MulticlassClassificationEvaluator (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.evaluation.MulticlassClassificationEvaluator (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.evaluation.MulticlassClassificationEvaluator))

```
# import the required MLlib package
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator

# define the binary and multi-class classification evaluators
bcEvaluator = BinaryClassificationEvaluator(metricName="areaUnderROC")
print(f"Area under ROC curve: {bcEvaluator.evaluate(pred_df)}")

mcEvaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print(f"Accuracy: {mcEvaluator.evaluate(pred_df)}")
```

```
Area under ROC curve: 0.9482973939029896
Accuracy: 0.9158664610179611
```

Step 6 - Hyperparameter tuning

MLlib provides methods to facilitate hyperparameter tuning and cross validation.

For hyperparameter tuning, we will use *ParamGridBuilder* to define a grid search over a set of model hyperparameters.

For cross validation, we will use *CrossValidator* to specify an estimator (the pipeline to apply to the input dataset), an evaluator, a grid space of hyperparameters, and the number of folds to use for cross validation.

For further reference, see

- Model selection using cross-validation (https://spark.apache.org/docs/latest/ml-tuning.html (https://spark.apache.org/docs/latest/ml-tuning.html))
- ParamGridBuilder (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#module-pyspark.ml.tuning (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#module-pyspark.ml.tuning))
- CrossValidator (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator))

```
# import the required MLlib package
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

paramGrid = (ParamGridBuilder()
    .addGrid(lr.regParam, [0.01, 0.5, 2.0])
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])
    .build())

# create a 3-fold CrossValidator
cv = CrossValidator(estimator=pipeline, estimatorParamMaps=paramGrid, evaluator=bcEvaluator, numFolds=3, parallelism=4)

# run cross validations. NB: This step takes a few minutes and returns the best model found from the cross validation.
cvModel = cv.fit(train_df)
```

Step 7 - Make predictions and evaluate model performance

Next, we use the best model identified by the cross-validation to make predictions on the test dataset, and then evaluate the model's performance using the area under the ROC curve.

```
# use the model to make predictions on the test dataset
cvPred_df = cvModel.transform(test_df)

# evaluate model performance
print(f"Area under ROC curve: {bcEvaluator.evaluate(cvPred_df)}")
print(f"Accuracy: {mcEvaluator.evaluate(cvPred_df)}")

# create a temporary view
cvPred_df.createOrReplaceTempView("finalPredictions")
```

```
Area under ROC curve: 0.9599511570445647
Accuracy: 0.9587044131548833
```

SQL queries and Data Visualization

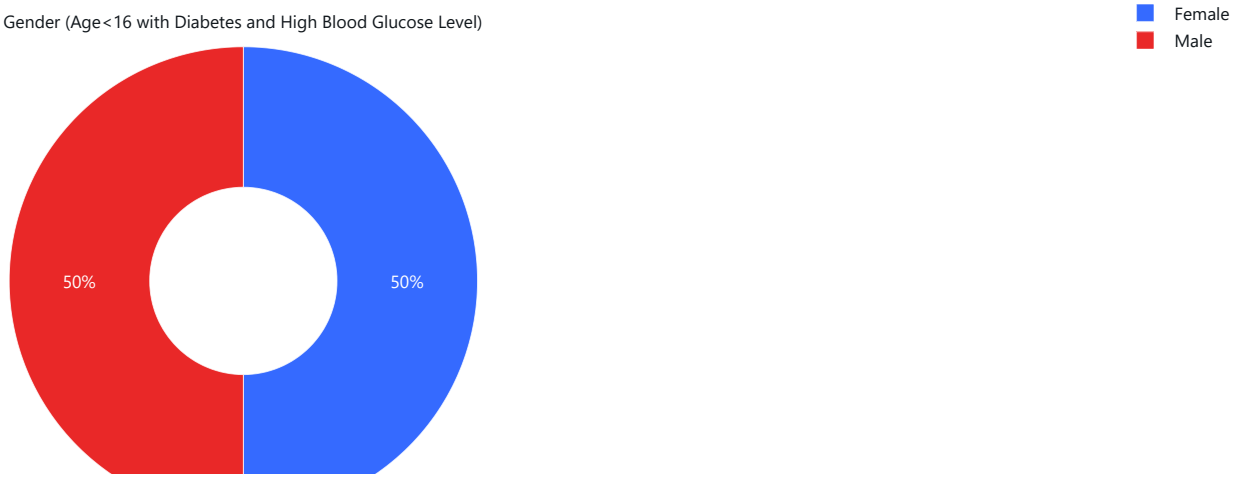
Using SQL commands, we can display diabetes predictions with the model. This requires creating a temporary view of the predictions dataset.

Query 3

Let's query the dataset to find out the gender that has the highest positive status for diabetes with blood glucose level greater or equal to 200 and age less than 16

```
%sql
SELECT gender, count(*) AS count
FROM finalPredictions WHERE (age < 16 AND blood_glucose_level >= 200 AND diabetes == 1)
GROUP BY gender
ORDER BY gender
```

Table | Gender (Age<16 with Diabetes and High Blood Glucose Level)



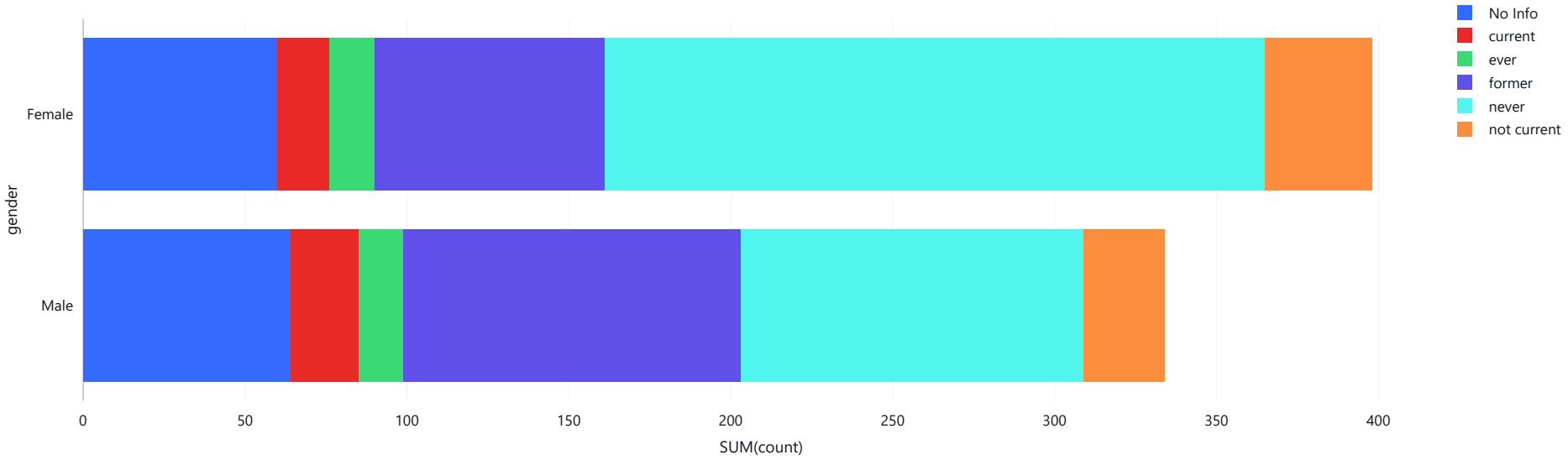
2 rows

Query 4

Next, let's query the dataset to find out the gender (with smoking history) that has the highest positive status for diabetes when their age is 65 or higher

```
%sql
SELECT gender, smoking_history, prediction, count(*) AS count
FROM finalPredictions WHERE (age > 65 AND diabetes == 1)
GROUP BY gender, smoking_history, prediction
ORDER BY  gender
```

Table | Gender (Age>65 with Diabetes vs Smoking History)



24 rows

```
# query to retrieve patients (age > 65) with diabetes
df4 = cvPred_df.filter("age > 65" and "diabetes == 1").select(["gender", "smoking_history", "prediction"]).groupBy("gender").count()
df4.show()
```

```
+------+-----+
|gender|count|
+------+-----+
|Female|  903|
|  Male|  788|
```

```
+------+-----+
```

Conclusion

From the diabetes prediction analysis, we see that 'females' older than 65 are more susceptible to test positive to diabetes than 'males' and the 'other' groups