Objective of the Analysis:

Develop a model to predict the presence or absence of cardiovascular disease (CVD) using the patient examination results.

In [ ]:
```
#Use the findspark library to locate spark on our local machine
import pyspark
import findspark
findspark.init()
findspark.find()
import pyspark
```

Creating the Spark Session

Firstly, We need to create a spark container by calling SparkSession. This step is necessary before doing anything

In [ ]:
```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession, SQLContext
spark = SparkSession.builder.master("local[*]").appName("Predictive model").getOrCreate()
```

Import Spark SQL and Spark ML Libraries

In [ ]:
```
import os
import pandas as pd
import numpy as np
from pyspark.sql.types import *
from pyspark.sql.functions import *
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import VectorAssembler, StringIndexer, VectorIndexer, MinMaxScaler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator, CrossValidatorModel
from pyspark.ml.feature import VectorAssembler, StandardScaler
```

In [ ]:
```
%matplotlib inline
import seaborn as sns
import matplotlib.pyplot as plt
```

DataFrame Creation from CSV File

In [ ]:
```
df = spark.read.csv(r"C:\Users\admin\OneDrive\Documents\cardio_train1.csv", header = True,
inferSchema = True)
```

In [ ]:
```
# Show first five rows
df.show(5)
```

In [ ]:
```
# show the schema of the datafram
df.printSchema()
```

EDA/Data Preprocessing

we need to convert SparkDataframe to PandasDataFrame using toPandas()

```
In [ ]: %%time
        df.toPandas().info(memory_usage='deep')
```

Checking null values in Pyspark

```
In [ ]: from pyspark.sql.functions import isnan, when, count, col
        df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.columns]).show()
```

```
In [ ]: df.groupBy('cardio').count().show()
```

I count the number of records for each target class, through the count(). I note that the dataset is balanced.

```
In [ ]: sns.set_style('darkgrid')
        sns.countplot(df.toPandas().cardio, palette='summer')
        plt.xlabel('Presence of cardiovascular disease',fontdict={'fontsize': 15,'color':'Green'},labelpa
        plt.savefig('saved_figure.png')
```

Summary Statistics:

```
In [ ]: # Basics stats from our columns
        df.toPandas().describe()
```

Visualizing the missing values

```
In [ ]: plt.figure(figsize=(18,6))
        sns.heatmap(df.toPandas().isnull(),cbar=False)
```

```
In [ ]: df.toPandas().duplicated().sum()
```

```
In [ ]: # Rename columns to make features more clearly understood
        df=df.withColumnRenamed('ap_hi','systolic')\
        .withColumnRenamed('ap_lo','diastolic')\
        .withColumnRenamed('gluc','glucose')\
        .withColumnRenamed('alco','alcohol')\
        .withColumnRenamed('cardio','label')
        df.printSchema()
```

```
In [ ]: # Increase the size of the heatmap.
        plt.figure(figsize=(16, 8))
        # Store heatmap object in a variable to easily access it when you want to include more features (
        # Set the range of values to be displayed on the colormap from -1 to 1, and set the annotation to
        heatmap = sns.heatmap(df.toPandas().corr(), vmin=-1, vmax=1, annot=True, center=0,
                    annot_kws={'size':13})
        # Give a title to the heatmap. Pad defines the distance of the title from the top of the heatmap
        heatmap.set_title('Correlation Heatmap', fontdict={'fontsize':18}, pad=12);
        plt.savefig('saved_figure1.png')
```

```
In [ ]: #Drop unnecessary columns
        df1=df.drop('id')
        df1.toPandas()
```

```
In [ ]: df2 = df1.filter(~((df1["systolic"]>200) | (df1["diastolic"]>180) | (df1["diastolic"]<50) |
              (df1["systolic"]<=80) | (df1["height"]<=100) | (df1["weight"]<=28)) )
```

```
#df2.show(truncate=False)
df2.toPandas()
```

In [ ]:
```
numeric_features = [t[0] for t in df2.dtypes if t[1] == 'int']
df2.select(numeric_features).describe().toPandas().transpose()
```

In [ ]:
```
#Count the number of distinct rows in df
df2.distinct().count()
```

In [ ]:
```
#Remove Duplicate Values
df3= df2.dropDuplicates()
df3.count()
```

Feature Engineering

In [ ]:
```
year=365
df4=df3.withColumn('age', df3['age']/year)\
.withColumn('BMI', df3['weight'] / df3['height'] / df3['height'] * 10000)\
.withColumn('pulse pressure', df3['systolic'] - df3['diastolic'])\
.withColumn('gender', df3['gender']%2)
df4.toPandas()
```

Use a VectorAssembler to put features into a feature vector column:

In [ ]:
```
# Assemble all the features with VectorAssembler
required_features = ['age','gender','height','weight','systolic','diastolic',
'cholesterol','glucose','smoke','alcohol','active','BMI','pulse pressure']
from pyspark.ml.feature import VectorAssembler
assembler = VectorAssembler(inputCols=required_features, outputCol='features')
assembled_df=assembler.transform(df4)
# Initialize the `standardScaler`
standardScaler = StandardScaler(inputCol="features", outputCol="features_scaled")
# Fit the DataFrame to the scaler
scaled_df = standardScaler.fit(assembled_df).transform(assembled_df)
```

In [ ]:
```
assembled_df.printSchema()
```

Standardization

Next, we can finally scale the data using StandardScaler. The input columns are the features, and the output column with the rescaled that will be included in the scaled_df will be named "features_scaled":

In [ ]:
```
# Initialize the `standardScaler`
standardScaler = StandardScaler(inputCol="features", outputCol="features_scaled")
```

In [ ]:
```
# Fit the DataFrame to the scaler
scaled_df = standardScaler.fit(assembled_df).transform(assembled_df)
```

In [ ]:
```
# Inspect the result
scaled_df.select("features", "label").toPandas().head()
```

Data Splitting

In [ ]:

```
# Split the data into train and test sets
train_data, test_data = scaled_df.randomSplit([.8,.2], seed=2018)
print(f"Size of train Dataset : {train_data.count()}" )
print(f"Size of test Dataset : {test_data.count()}" )
```

Applying Classification Models

In [ ]:

```
# Create binary evaluator object
evaluator = BinaryClassificationEvaluator(metricName = 'areaUnderPR')
```

Logistic Regression Model

In [ ]:

```
# Create a logistic regression object
lr = LogisticRegression(featuresCol = 'features_scaled', labelCol = 'label', maxIter=3)
```

In [ ]:

```
# Train the logistic regression model without parameter tuning
lrModel = lr.fit(train_data)
lrpredicted = lrModel.transform(test_data)
print('Test Area Under PR', evaluator.evaluate(lrpredicted))
```

In [ ]:

```
# Create ParamGrid for Cross Validation
lrparamGrid = (ParamGridBuilder()
             .addGrid(lr.regParam, [0.01, 0.5, 2.0])
             .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])
             .addGrid(lr.maxIter, [1, 5, 10])
             .build())

lrcv = CrossValidator(estimator=lr, estimatorParamMaps=paramGrid,
                      evaluator=evaluator, numFolds=5)

# Run cross validations
lrcvModel = lrcv.fit(train_data)
predictions = lrcvModel.transform(test_data)
predictions_pandas = predictions.toPandas()
print('Test Area Under PR: ', evaluator.evaluate(predictions))
```

In [ ]:

```
# Calculate and print f1, recall and precision scores
from sklearn.metrics import precision_score, recall_score, f1_score
f1 = f1_score(predictions_pandas.label, predictions_pandas.prediction)
recall = recall_score(predictions_pandas.label, predictions_pandas.prediction)
precision = precision_score(predictions_pandas.label, predictions_pandas.prediction)

print('F1-Score: {}, Recall: {}, Precision: {}'.format(f1, recall, precision))
```

In [ ]:

```
#Visualize Confusion Matrix for LogisticRegression
import sklearn
from sklearn.metrics import confusion_matrix
f, ax = plt.subplots(figsize=(6,4))
y_true = lrpredicted.select("label")
y_true = y_true.toPandas()

y_pred = lrpredicted.select("prediction")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred)
cnf_matrix
```

```
sns.heatmap(cnf_matrix, fmt=".0f", annot=True, linewidths=0.2, linecolor="purple",
            ax=ax, annot_kws={'size':13})
ax.set_title("LogisticRegression", fontsize=14)
plt.xlabel("Predicted values", fontsize=15)
plt.ylabel("Actual Values", fontsize=15)
plt.show()
#plt.savefig('books_read.png')
#plt.savefig('foo1.png', bbox_inches='tight')
fig1 = plt.gcf()
fig1.savefig('a.jpeg', bbox_inches='tight', pad_inches=0)
#plt.savefig("a.jpeg")
```

Decision Tree Classifier

In [ ]:
```
# Create decision tree classifier
dt = DecisionTreeClassifier(featuresCol = 'features', labelCol = 'label')
```

In [ ]:
```
dtModel = dt.fit(train_data)
dtpredicted = dtModel.transform(test_data)
print('Test Area Under PR', evaluator.evaluate(dtpredicted))
```

In [ ]:
```
# Create ParamGrid for Cross Validation
dtparamGrid = (ParamGridBuilder()
             .addGrid(dt.maxDepth, [2, 4, 6])
             .addGrid(dt.maxBins, [20, 60])
             .addGrid(dt.impurity, ['gini', 'entropy'])
             .build())
dtcv = CrossValidator(estimator=dt, estimatorParamMaps=dtparamGrid,
                      evaluator=evaluator, numFolds=5)

# Run cross validations
dtcvModel = dtcv.fit(train_data)
predictions = dtcvModel.transform(test_data)
predictions_pandas = predictions.toPandas()
print('Test Area Under PR: ', evaluator.evaluate(predictions))
```

In [ ]:
```
# Calculate and print f1, recall and precision scores
f1 = f1_score(predictions_pandas.label, predictions_pandas.prediction)
recall = recall_score(predictions_pandas.label, predictions_pandas.prediction)
precision = precision_score(predictions_pandas.label, predictions_pandas.prediction)

print('F1-Score: {}, Recall: {}, Precision: {}'.format(f1, recall, precision))
```

In [ ]:
```
#Visualize Confusion Matrix for Decision Tree Classifier
import sklearn
from sklearn.metrics import confusion_matrix
f, ax = plt.subplots(figsize=(6,4))
y_true = dtpredicted.select("label")
y_true = y_true.toPandas()

y_pred = dtpredicted.select("prediction")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred)
cnf_matrix
sns.heatmap(cnf_matrix, fmt=".0f", annot=True, linewidths=0.2, linecolor="purple",
            ax=ax, annot_kws={'size':13})
ax.set_title("Decision Tree ", fontsize=14)
plt.xlabel("Predicted Values", fontsize=15)
```

```
plt.ylabel("Actual Values", fontsize=15)
plt.show()
```

Random Forest Classifier

In [ ]:
```
#Random Forest Classifier
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(featuresCol='features_scaled',labelCol="label", numTrees=100)
```

In [ ]:
```
rfModel = rf.fit(train_data)
rfpredicted = rfModel.transform(test_data)
print('Test Area Under PR', evaluator.evaluate(rfpredicted))
```

In [ ]:
```
# Create ParamGrid for Cross Validation
rfparamGrid = (ParamGridBuilder()
             .addGrid(rf.maxDepth, [2, 5, 10])
             .addGrid(rf.maxBins, [5, 10, 20])
             .addGrid(rf.numTrees, [5, 20, 50])
             .build())

# Create 5-fold CrossValidator
rfcv = CrossValidator(estimator = rf, estimatorParamMaps = rfparamGrid,
                      evaluator = evaluator, numFolds=5)

# Run cross validations
rfcvModel = rfcv.fit(train_data)
predictions = rfcvModel.transform(test_data)
predictions_pandas = predictions.toPandas()
print('Test Area Under PR: ', evaluator.evaluate(predictions))
```

In [ ]:
```
# Calculate and print f1, recall and precision scores
from sklearn.metrics import precision_score, recall_score, f1_score
f1 = f1_score(predictions_pandas.label, predictions_pandas.prediction)
recall = recall_score(predictions_pandas.label, predictions_pandas.prediction)
precision = precision_score(predictions_pandas.label, predictions_pandas.prediction)

print('F1-Score: {}, Recall: {}, Precision: {}'.format(f1, recall, precision))
```

In [ ]:
```
#Visualize Confusion Matrix for Random Forest Classifier
import sklearn
from sklearn.metrics import confusion_matrix
f, ax = plt.subplots(figsize=(6,4))
y_true = rfpredicted.select("label")
y_true = y_true.toPandas()

y_pred = rfpredicted.select("prediction")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred)
cnf_matrix
sns.heatmap(cnf_matrix, fmt=".0f", annot=True, linewidths=0.2, linecolor="purple",
            ax=ax, annot_kws={'size':13})
ax.set_title("Random Forest ", fontsize=14)
plt.xlabel("Predicted Values", fontsize=15)
plt.ylabel("Actual Values", fontsize=15)
plt.show()
```

Gradient-Boosted Tree Classifier

In [ ]:
```
# Create gradient-boosted tree classifier object
from pyspark.ml.classification import GBTClassifier
gbt = GBTClassifier(featuresCol = 'features', labelCol = 'label')
```

In [ ]:
```
gbtModel = gbt.fit(train_data)
gbtpredicted = gbtModel.transform(test_data)
print('Test Area Under PR', evaluator.evaluate(gbtpredicted))
```

In [ ]:
```
# Create ParamGrid for Cross Validation
gbtparamGrid = (ParamGridBuilder()
             .addGrid(gbt.maxDepth, [2, 4, 6])
             .addGrid(gbt.maxBins, [20, 60])
             .addGrid(gbt.maxIter, [10, 20])
             .build())
gbtcv = CrossValidator(estimator=gbt, estimatorParamMaps=gbtparamGrid,
                       evaluator=evaluator, numFolds=3)

# Run cross validations
gbtcvModel = gbtcv.fit(train_data)
predictions = gbtcvModel.transform(test_data)
predictions_pandas = predictions.toPandas()
print('Test Area Under PR: ', evaluator.evaluate(predictions))
```

In [ ]:
```
# Calculate and print f1, recall and precision scores
f1 = f1_score(predictions_pandas.label, predictions_pandas.prediction)
recall = recall_score(predictions_pandas.label, predictions_pandas.prediction)
precision = precision_score(predictions_pandas.label, predictions_pandas.prediction)

print('F1-Score: {}, Recall: {}, Precision: {}'.format(f1, recall, precision))
```

In [ ]:
```
# Visualize Confusion Matrix for Gradient-Boosted Tree Classifier
import sklearn
from sklearn.metrics import confusion_matrix
f, ax = plt.subplots(figsize=(6,4))
y_true = gbtpredicted.select("label")
y_true = y_true.toPandas()

y_pred = gbtpredicted.select("prediction")
y_pred = y_pred.toPandas()

cnf_matrix = confusion_matrix(y_true, y_pred)
cnf_matrix
sns.heatmap(cnf_matrix, fmt=".0f", annot=True, linewidths=0.2, linecolor="purple",
            ax=ax, annot_kws={'size':13})
ax.set_title("Gradient-Boosted Tree ", fontsize=14)
plt.xlabel("Predicted Values", fontsize=15)
plt.ylabel("Actual Values", fontsize=15)
plt.show()
```

Accuracy - Evaluation Metric

In [ ]:
```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
acc_evaluator = MulticlassClassificationEvaluator(predictionCol='prediction',
                                                  labelCol='label', metricName='accuracy')

acc_lr = acc_evaluator.evaluate(lrpredicted)
acc_dt = acc_evaluator.evaluate(dtpredicted)
acc_rf = acc_evaluator.evaluate(rfpredicted)
acc_gbt = acc_evaluator.evaluate(gbtpredicted)
```

```python
print('Logistic Regression accuracy: ', '{:.2f}'.format(acc_lr*100), '%', sep='')
print('Decision Tree Classifier: ', '{:.2f}'.format(acc_dt*100), '%', sep='')
print('Random Forest Classifier: ', '{:.2f}'.format(acc_rf*100), '%', sep='')
print('Gradient-Boosted Tree Classifier: ', '{:.2f}'.format(acc_gbt*100), '%', sep='')
```

In [ ]:
```python
spark.stop()
```

Results

The best performing model is Gradient-Boosted Tree Classifier.