Analysis of Heart Disease Dataset

March 14, 2023

PERFORMING THE ESSENTIAL CONFIGURATION

[1]:
```
!pip install pyspark
!pip install pyspark[sql]
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: pyspark in /usr/local/lib/python3.9/dist-packages
(3.3.2)
Requirement already satisfied: py4j==0.10.9.5 in /usr/local/lib/python3.9/dist-
packages (from pyspark) (0.10.9.5)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: pyspark[sql] in /usr/local/lib/python3.9/dist-
packages (3.3.2)
Requirement already satisfied: py4j==0.10.9.5 in /usr/local/lib/python3.9/dist-
packages (from pyspark[sql]) (0.10.9.5)
Requirement already satisfied: pandas>=1.0.5 in /usr/local/lib/python3.9/dist-
packages (from pyspark[sql]) (1.4.4)
Requirement already satisfied: pyarrow>=1.0.0 in /usr/local/lib/python3.9/dist-
packages (from pyspark[sql]) (9.0.0)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.9/dist-packages (from pandas>=1.0.5->pyspark[sql])
(2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.9/dist-
packages (from pandas>=1.0.5->pyspark[sql]) (2022.7.1)
Requirement already satisfied: numpy>=1.18.5 in /usr/local/lib/python3.9/dist-
packages (from pandas>=1.0.5->pyspark[sql]) (1.22.4)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-
packages (from python-dateutil>=2.8.1->pandas>=1.0.5->pyspark[sql]) (1.15.0)
```

This code installs the PySpark library and its SQL module using pip, a package manager for Python.

The exclamation mark (!) at the beginning of each line is a Jupyter Notebook syntax that indicates that the following command should be executed in a shell rather than in Python code.

The first line installs the core PySpark library, which provides a distributed computing framework for processing large datasets across clusters. The second line installs the SQL module for PySpark, which adds support for structured data processing using SQL queries.

By running these commands, the necessary dependencies and packages will be installed in the current environment, allowing you to use PySpark and its SQL capabilities in your Python code.

[2]:
```
# Import the essential spark packages to work with.
from pyspark import SparkConf
from pyspark.sql import SparkSession

from pyspark.sql.functions import lit, array_remove, rand, col, countDistinct
from pyspark.sql.functions import isnan, when, count, explode, array
import pyspark.sql.functions as F

from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler,␣
  ↪StandardScaler
from pyspark.ml.functions import vector_to_array
from pyspark.ml.classification import DecisionTreeClassifier,␣
  ↪RandomForestClassifier, NaiveBayes
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

from pyspark.sql import Row
```

This code imports various packages and modules from the PySpark library that are necessary for working with Spark data processing and machine learning. Here is a breakdown of each line:

from pyspark import SparkConf: This line imports the SparkConf class from the pyspark module. The SparkConf class is used to configure a Spark application. from pyspark.sql import SparkSession: This line imports the SparkSession class from the pyspark.sql module. SparkSession is the entry point for working with structured data in Spark.

from pyspark.sql.functions import lit, array_remove, rand, col, countDistinct: This line imports several functions from the pyspark.sql.functions module, including lit, array_remove, rand, col, and countDistinct. These functions are used to manipulate data in Spark dataframes.

from pyspark.sql.functions import isnan, when, count, explode, array: This line imports several more functions from the pyspark.sql.functions module, including isnan, when, count, explode, and array. These functions are used for filtering and aggregating data in Spark dataframes.

import pyspark.sql.functions as F: This line imports the pyspark.sql.functions module with the alias F. This is a common convention to make it easier to reference the functions in this module later in the code.

from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler, StandardScaler: This line imports several classes from the pyspark.ml.feature module, including OneHotEncoder, StringIndexer, VectorAssembler, and StandardScaler. These classes are used for preparing data for machine learning models in Spark.

from pyspark.ml.functions import vector_to_array: This line imports the vector_to_array function from the pyspark.ml.functions module. This function is used to convert a vector column in a Spark dataframe to an array column. from pyspark.ml.classification import DecisionTreeClassifier, RandomForestClassifier, NaiveBayes: This line imports several classification models from the pyspark.ml.classification module, including DecisionTreeClassifier, RandomForestClassifier, and

NaiveBayes. These are common machine learning models used for classification tasks.

from pyspark.ml import Pipeline: This line imports the Pipeline class from the pyspark.ml module. Pipeline is used to define a sequence of data preparation and modeling steps in Spark.

from pyspark.ml.evaluation import MulticlassClassificationEvaluator: This line imports the MulticlassClassificationEvaluator class from the pyspark.ml.evaluation module. This class is used to evaluate the performance of a classification model.

from pyspark.sql import Row: This line imports the Row class from the pyspark.sql module. Row is used to represent a row of data in a Spark dataframe.

[3]:
```
# Import the data visualization libraries
import seaborn as sns
import matplotlib.pyplot as plt
```

This code imports two popular data visualization libraries: seaborn and matplotlib.pyplot.

import seaborn as sns: This line imports the seaborn library with the alias sns. seaborn is a Python library that provides high-level interface for creating beautiful and informative statistical graphics. It is built on top of matplotlib and integrates well with Pandas dataframes.

import matplotlib.pyplot as plt: This line imports the pyplot module from the matplotlib library, and gives it the alias plt. matplotlib is a Python library for creating static, animated, and interactive visualizations in Python. pyplot is a sub-module of matplotlib that provides a convenient interface for creating common types of plots, such as line plots, scatter plots, and histograms.

[4]:
```
# Creating the Spark Session. SparkSession is the entry point to Spark SQL.
spark = SparkSession.builder\
        .appName("Classification")\
        .getOrCreate()
```

This code creates a new SparkSession object, which is the entry point for working with structured data in Spark SQL.

SparkSession is a class in PySpark that encapsulates the underlying Spark functionality and provides a unified API to interact with Spark.

builder is a method of SparkSession that returns a Builder object, which is used to configure the SparkSession.

appName is a method of the Builder object that sets the name of the Spark application. In this case, the name of the application is "Classification".

getOrCreate is a method of the Builder object that returns an existing SparkSession if there is one, or creates a new one if none exists.

Overall, this code creates a SparkSession object that is configured with the name "Classification". This object can be used to read data from various data sources, create dataframes, and execute SQL queries on them.

[5]:
```
# Import the dataset
```

```
heart_df = spark.read.csv('/content/heart.csv', sep=',', inferSchema=True,␣
↪header=True)
```

To persist a Spark DataFrame into HDFS, where it can be queried using default Hadoop SQL engine (Hive), one straightforward strategy (not the only one) is to create a temporal view from that DataFrame.

This code reads a CSV file called 'heart.csv' into a Spark dataframe named heart_df.

spark is the SparkSession object created earlier.

read is a method of the SparkSession object that reads data from a data source and creates a dataframe.

csv is the format of the data source, and '/content/heart.csv' is the path to the file. sep=',' specifies that the fields are separated by commas.

inferSchema=True tells Spark to automatically infer the data types of the columns based on the data.

header=True indicates that the first row of the CSV file contains the column names.

Overall, this code reads the CSV file into a dataframe with the name heart_df. The dataframe contains the data from the CSV file and can be used for data analysis and machine learning tasks.

[6]:
```
heart_df.createOrReplaceTempView("heart_hdfs")
```

This code creates a temporary view of the heart_df dataframe in Spark SQL, with the name heart_hdfs.

createOrReplaceTempView is a method of a Spark dataframe that creates a temporary view of the dataframe in Spark SQL.

"heart_hdfs" is the name of the temporary view that is being created.

Overall, this code allows us to query the heart_df dataframe using Spark SQL by referring to it as heart_hdfs.

We can execute SQL queries on this temporary view, which provides a more expressive way to analyze and manipulate the data in the dataframe. Note that the temporary view is only available for the duration of the SparkSession and will be deleted when the session ends.

[7]:
```
# Performing the SQL operation.
df = spark.sql("SELECT * FROM heart_hdfs")
```

This code creates a temporary view of the heart_df dataframe in Spark SQL, with the name heart_hdfs.

createOrReplaceTempView is a method of a Spark dataframe that creates a temporary view of the dataframe in Spark SQL.

"heart_hdfs" is the name of the temporary view that is being created.

Overall, this code allows us to query the heart_df dataframe using Spark SQL by referring to it as heart_hdfs.

We can execute SQL queries on this temporary view, which provides a more expressive way to analyze and manipulate the data in the dataframe.

Note that the temporary view is only available for the duration of the SparkSession and will be deleted when the session ends.

[8]:
```
df.show(5)
```

```
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+
|age|sex| cp|trtbps|chol|fbs|restecg|thalachh|exng|oldpeak|slp|caa|thall|output|
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+
| 63|  1|  3|   145| 233|  1|      0|     150|   0|    2.3|  0|  0|    1|     1|
| 37|  1|  2|   130| 250|  0|      1|     187|   0|    3.5|  0|  0|    2|     1|
| 41|  0|  1|   130| 204|  0|      0|     172|   0|    1.4|  2|  0|    2|     1|
| 56|  1|  1|   120| 236|  0|      1|     178|   0|    0.8|  2|  0|    2|     1|
| 57|  0|  0|   120| 354|  0|      1|     163|   1|    0.6|  2|  0|    2|     1|
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+
only showing top 5 rows
```

This code displays the first 5 rows of the DataFrame df using the show method. The output shows the values in each column of the first 5 rows of the DataFrame, which can be helpful for inspecting the data and verifying that the DataFrame was loaded and processed correctly.

[9]:
```
# Correlation matrix of each variable with respect to output variable. Here the␣
 ↪correlation of all
# variables are not too strong, and therefore, we will use all the variables.␣
 ↪One can eleminate the
# chol and fbs variable. By eliminating them, the accuracy of the decision tree␣
 ↪may get low.
heart_df.toPandas().corr().output
```

[9]:
```
age        -0.225439
sex        -0.280937
cp          0.433798
trtbps     -0.144931
chol       -0.085239
fbs        -0.028046
restecg     0.137230
thalachh    0.421741
exng       -0.436757
oldpeak    -0.430696
slp         0.345877
caa        -0.391724
thall      -0.344029
output      1.000000
Name: output, dtype: float64
```

This code calculates the correlation between the output column and all other columns of the heart_df DataFrame using the corr() method.

The toPandas() method is called on the DataFrame to convert it to a Pandas DataFrame, which allows for easier visualization and analysis of the correlation matrix.

Finally, output is appended to the end of the command to display the correlations between the output column and all other columns in the Pandas DataFrame. This can help identify which columns have the strongest correlation with the target variable output.

[10]:
```
# Viewing if there are columns with None, NULL, empty, NaN values
heart_df.select([count(when(col(c).contains('None') | \
                            col(c).contains('NULL') | \
                            (col(c) == '' ) | \
                            col(c).isNull() | \
                            isnan(c), c
                           )).alias(c)
                    for c in heart_df.columns]).show()
```

```
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+
|age|sex| cp|trtbps|chol|fbs|restecg|thalachh|exng|oldpeak|slp|caa|thall|output|
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+
|  0|  0|  0|     0|   0|  0|      0|       0|   0|      0|  0|  0|    0|     0|
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+
```

This code selects all the columns from the heart_df dataframe and counts the number of values that are None, NULL, empty, or NaN. It then creates a new dataframe that shows the count for each column.

select is a method of a Spark dataframe that selects a subset of the columns and applies a transformation to each row.

[count(when(…)).alias(c) for c in heart_df.columns] is a list comprehension that applies the count and when functions to each column in the dataframe, and renames the resulting count column with the original column name. count is a Spark SQL function that counts the number of non-null values in a column.

when is a Spark SQL function that applies a condition to a column and returns a value if the condition is true. In this case, it checks if the column contains None, NULL, an empty string, or NaN, and returns the column value if the condition is true.

col(c) is a function that refers to the cth column in the dataframe. contains is a Spark SQL function that checks if a string contains another string.

isNull is a Spark SQL function that checks if a column value is null. isnan is a function that checks if a value is NaN (not a number). alias(c) is a method that renames the resulting count column with the original column name.

Overall, this code checks for missing values in the heart_df dataframe by counting the number of None, NULL, empty, or NaN values in each column. The resulting dataframe shows the count for each column, which can be used to identify columns with missing values that need to be handled before further analysis or modeling.

[11]:

```
# Viewing the values of categorical variables. Here null 303 means that
# there are no null values.
heart_df.cube('output').count().show()
heart_df.cube('cp').count().show()
heart_df.cube('fbs').count().show()
heart_df.cube('restecg').count().show()
heart_df.cube('exng').count().show()
heart_df.cube('slp').count().show()
heart_df.cube('caa').count().show()
heart_df.cube('thall').count().show()
```

```
+------+-----+
|output|count|
+------+-----+
|     1|  165|
|     0|  138|
|  null|  303|
+------+-----+

+----+-----+
|  cp|count|
+----+-----+
|   1|   50|
|   0|  143|
|null|  303|
|   3|   23|
|   2|   87|
+----+-----+

+----+-----+
| fbs|count|
+----+-----+
|   1|   45|
|   0|  258|
|null|  303|
+----+-----+

+-------+-----+
|restecg|count|
+-------+-----+
|      1|  152|
|      0|  147|
|   null|  303|
|      2|    4|
+-------+-----+

+----+-----+
|exng|count|
+----+-----+
|   1|   99|
|   0|  204|
|null|  303|
+----+-----+

+----+-----+
| slp|count|
+----+-----+
|   1|  140|
|   0|   21|
|null|  303|
|   2|  142|
+----+-----+

+----+-----+
| caa|count|
+----+-----+
|   1|   65|
|   0|  175|
|null|  303|
|   4|    5|
|   3|   20|
|   2|   38|
+----+-----+

+-----+-----+
|thall|count|
+-----+-----+
|    1|   18|
|    0|    2|
| null|  303|
|    3|  117|
|    2|  166|
+-----+-----+
```

This code generates a summary of the output column in the heart_df dataframe using the cube and count methods, and then displays the result using the show method.

cube is a method of a Spark dataframe that creates a multi-dimensional cube for aggregating data. It generates a summary of the data by grouping the rows based on the specified columns, and computing the aggregate function(s) for each group.

‘output’ is the name of the column on which we are performing the cube operation. It specifies that we want to group the rows based on the unique values in the output column.

count is a Spark SQL function that counts the number of non-null values in a column. In this case, we are using it to count the number of rows in each group generated by the cube operation.

show is a method that displays the resulting dataframe in a tabular format. Overall, this code generates a summary of the output column in the heart_df dataframe by grouping the rows based on the unique values in the column, and counting the number of rows in each group.

The resulting dataframe shows the count for each unique value in the output column. This summary can be used to understand the distribution of the output values in the dataset, and to identify any class imbalances that may need to be addressed during modeling.

CREATING THE PIPELINE

[12]:
```
# Converting the categorical variables into one-hot encoded variables
# Here stages mean pipeline stage. By adding and removing the names from
# any of three lists give below, the accuracy will change.

# First List
categoricalColumns = ['slp', 'cp', 'restecg', 'caa', 'thall', 'fbs']
stages = []
for categoricalCol in categoricalColumns:
    encoder = OneHotEncoder(inputCols=[categoricalCol],␣
  ↪outputCols=[categoricalCol + "classVec"])
    stages += [encoder]
```

This code defines a list of categorical columns in a dataset, and then creates a list of transformation stages using the OneHotEncoder method from the PySpark ML library.

categoricalColumns is a list of column names that represent categorical variables in the dataset. In this example, the categorical variables are slp, cp, restecg, caa, thall, and fbs. stages is an empty list that will be used to store the transformation stages for the OneHotEncoder.

A for-loop is used to iterate over the categorical columns. For each column, a new OneHotEncoder object is created with the input column name as the inputCols parameter and a new output column name created by appending "classVec" to the original column name.

The new OneHotEncoder object is added to the stages list using the += operator. Overall, this code creates a list of transformation stages using the OneHotEncoder method for a specified list of categorical columns.

These transformation stages can be used to preprocess the data before applying a machine learning algorithm that requires numerical data. The OneHotEncoder converts each categorical variable into a vector of binary values, where each possible category is represented by a single element in the vector.

[13]:
```
# Performing the standard scalar to each of the variables mentioned in the␣
  ↪columnsToScale

# Second List
columnsToScale = ['age', 'chol', 'thalachh', 'oldpeak', 'trtbps']

for columnToScale in columnsToScale:
   va = VectorAssembler(inputCols=[columnToScale], outputCol=columnToScale +␣
  ↪"VAStdScaler")
```

```
    stages += [va]
    sScaler = StandardScaler(
      withMean=False, withStd=True, inputCol=columnToScale + "VAStdScaler",␣
  ↪outputCol=columnToScale + "StdScaler")
    stages += [sScaler]
```

This code performs standard scaling on specified variables in the columnsToScale list, which involves transforming the variables so that they have a mean of zero and a standard deviation of one.

columnsToScale is a list of column names that represent variables to be scaled. In this example, the variables are age, chol, thalachh, oldpeak, and trtbps. A for-loop is used to iterate over the columns to be scaled.

For each column, a new VectorAssembler object is created with the input column name as the inputCols parameter and a new output column name created by appending "VAStdScaler" to the original column name. The new VectorAssembler object is added to the stages list using the += operator.

A StandardScaler object is created with the inputCol parameter set to the output column name from the VectorAssembler object created in the previous step and a new output column name created by appending "StdScaler" to the original column name.

The new StandardScaler object is added to the stages list using the += operator. Overall, this code creates a list of transformation stages that standardize the specified variables in the columnsToScale list. These transformation stages can be used to preprocess the data before applying a machine learning algorithm that requires scaled data. The VectorAssembler and StandardScaler methods are from the PySpark ML library.

[14]:
```
# Wraping everything variables into single feature variables
label_stringIdx = StringIndexer(inputCol = 'output', outputCol = 'label')
stages += [label_stringIdx]

# Adding those variables that does not require to change
# into one-hot encoded variable and they are not contineous variable either.

# Third List
numericCols = ['sex']
assemblerInputs = [c + "classVec" for c in categoricalColumns] + [c +␣
  ↪"StdScaler" for c in columnsToScale] + numericCols
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
stages += [assembler]
```

This code creates a single feature vector by combining all the different input variables. The label_stringIdx object is a StringIndexer that creates a new column called "label" with indexed values based on the values in the "output" column.

The categorical columns in the categoricalColumns list are one-hot encoded using the OneHotEncoder method and added to the stages list. The output column names for each encoded categorical column are created by appending "classVec" to the original column name.

The continuous variables in the columnsToScale list are scaled using VectorAssembler and StandardScaler methods and added to the stages list. The output column names for each scaled column are created by appending “VAStdScaler” and “StdScaler” to the original column name, respectively.

Finally, all the one-hot encoded and scaled variables are combined into a single feature vector using the VectorAssembler method with the inputCols parameter set to the output column names from the previous steps. The output column name for the feature vector is “features” and it is added to the stages list.

[ ]:
```python
# Creating Pipeline
pipeline = Pipeline(stages = stages)

pipelineModel = pipeline.fit(heart_df)
heart_df = pipelineModel.transform(heart_df)
heart_df.show(5)
```

```
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+-------------+-------------+---------------+
|age|sex| cp|trtbps|chol|fbs|restecg|thalachh|exng|oldpeak|slp|caa|thall|output|  slpclassVec|   cpclassVec|restecgclassVec|
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+-------------+-------------+---------------+
| 63|  1|  3|   145| 233|  1|      0|     150|   0|    2.3|  0|  0|    1|     1|(2,[0],[1.0])|    (3,[],[])|  (2,[0],[1.0])|
| 37|  1|  2|   130| 250|  0|      1|     187|   0|    3.5|  0|  0|    2|     1|(2,[0],[1.0])|(3,[2],[1.0])|  (2,[1],[1.0])|
| 41|  0|  1|   130| 204|  0|      0|     172|   0|    1.4|  2|  0|    2|     1|    (2,[],[])|(3,[1],[1.0])|  (2,[0],[1.0])|
| 56|  1|  1|   120| 236|  0|      1|     178|   0|    0.8|  2|  0|    2|     1|    (2,[],[])|(3,[1],[1.0])|  (2,[1],[1.0])|
| 57|  0|  0|   120| 354|  0|      1|     163|   1|    0.6|  2|  0|    2|     1|    (2,[],[])|(3,[0],[1.0])|  (2,[1],[1.0])|
+---+---+---+------+----+---+-------+--------+----+-------+---+---+-----+------+-------------+-------------+---------------+
only showing top 5 rows
```

```
+-------------+-------------+-------------+--------------+--------------------+---------------+--------------------+-------------------+
|  caaclassVec|thallclassVec|  fbsclassVec|ageVAStdScaler|        ageStdScaler|cholVAStdScaler|       cholStdScaler|thalachhVAStdScaler|
+-------------+-------------+-------------+--------------+--------------------+---------------+--------------------+-------------------+
|(4,[0],[1.0])|(3,[1],[1.0])|    (1,[],[])|        [63.0]| [6.936720927293358]|        [233.0]| [4.495400810500686]|            [150.0]|
|(4,[0],[1.0])|(3,[2],[1.0])|(1,[0],[1.0])|        [37.0]|[4.0739472112675275]|        [250.0]| [4.823391427575843]|            [187.0]|
|(4,[0],[1.0])|(3,[2],[1.0])|(1,[0],[1.0])|        [41.0]| [4.514373936809963]|        [204.0]| [3.935887404901888]|            [172.0]|
|(4,[0],[1.0])|(3,[2],[1.0])|(1,[0],[1.0])|        [56.0]| [6.165974157594095]|        [236.0]|[4.5532815076315964]|            [178.0]|
|(4,[0],[1.0])|(3,[2],[1.0])|(1,[0],[1.0])|        [57.0]| [6.276080838979705]|        [354.0]| [6.829922261447394]|            [163.0]|
+-------------+-------------+-------------+--------------+--------------------+---------------+--------------------+-------------------+
```

```
+--------------------+------------------+--------------------+-----------------+-------------------+-----+--------------------+
|   thalachhStdScaler|oldpeakVAStdScaler|    oldpeakStdScaler|trtbpsVAStdScaler|    trtbpsStdScaler|label|            features|
+--------------------+------------------+--------------------+-----------------+-------------------+-----+--------------------+
| [6.548742409951069]|             [2.3]|[1.9809228140160955]|          [145.0]|[8.267694107738977]|  0.0|(21,[0,5,7,12,15,...|
| [8.164098871072333]|             [3.5]| [3.014447760459276]|          [130.0]|[7.412415406938393]|  0.0|(21,[0,4,6,7,13,1...|
| [7.509224630077226]|             [1.4]|[1.2057791041837103]|          [130.0]|[7.412415406938393]|  0.0|(21,[3,5,7,13,14,...|
| [7.771174326475268]|             [0.8]|[0.6890166309621203]|          [120.0]|[6.842229606404671]|  0.0|(21,[3,6,7,13,14,...|
|[7.1163000854801615]|             [0.6]|[0.5167624732215902]|          [120.0]|[6.842229606404671]|  0.0|(21,[2,6,7,13,14,...|
+--------------------+------------------+--------------------+-----------------+-------------------+-----+--------------------+
```

This code creates a Pipeline object with the stages defined earlier, which includes data preprocessing steps such as one-hot encoding and scaling of the features.

Then, the Pipeline object is fitted to the heart_df DataFrame using the fit method to create a pipelineModel. The transform method is called on the pipelineModel object to transform the heart_df DataFrame by applying all the preprocessing steps defined in the stages list.

The transformed DataFrame is then printed using the show method, which displays the first 5 rows of the DataFrame. The output of the show method shows the transformed DataFrame with the original columns and the new label and features columns that were created during the preprocessing step.

[16]:
```python
# Selecting features, having values of all the variables, and output variable
heart_df = heart_df.select('features', 'output')
heart_df.printSchema()
```

```
root
```

```
 |-- features: vector (nullable = true)
 |-- output: integer (nullable = true)
```

This code selects only the features and output columns from the heart_df DataFrame and creates a new DataFrame with only those columns.

The printSchema() method is called on the new DataFrame to display the schema of the DataFrame. This displays the data types of the columns and any nested structure of the DataFrame.

[17]:
```
heart_df.show(5)
```

```
+--------------------+------+
|            features|output|
+--------------------+------+
|(21,[0,5,7,12,15,…|     1|
|(21,[0,4,6,7,13,1…|     1|
|(21,[3,5,7,13,14,…|     1|
|(21,[3,6,7,13,14,…|     1|
|(21,[2,6,7,13,14,…|     1|
+--------------------+------+
only showing top 5 rows
```

This code displays the first 5 rows of the DataFrame heart_df using the show() method. The output shows the values in each column of the first 5 rows of the DataFrame, which can be helpful for inspecting the data and verifying that the DataFrame was loaded and processed correctly.

TRAINING THE MODEL

[18]:
```
# Train and test split
train_data, test_data = heart_df.randomSplit([0.7, 0.3], seed = 42)
print("Training Dataset Count: " + str(train_data.count()))
print("Test Dataset Count: " + str(test_data.count()))
```

```
Training Dataset Count: 224
Test Dataset Count: 79
```

This code splits the preprocessed heart_df DataFrame into two parts: a training set and a test set, using a 70:30 ratio. The randomSplit() method is used to randomly divide the data. The seed parameter is set to 42 to ensure that the split is reproducible.

The sizes of the resulting training and test sets are printed using the count() method. The output shows the number of rows in each of the datasets, which can be helpful for verifying that the split was performed correctly.

[19]:
```
# Use mostly defaults to make this comparison "fair"
# For decision Tree
dtc = DecisionTreeClassifier(labelCol='output',featuresCol='features')
# For Random Forest Tree
rfc = RandomForestClassifier(labelCol='output',featuresCol='features')
```

```
# For Naive Bayes Classification
nbc = NaiveBayes(labelCol='output',featuresCol='features')
```

The code defines three classification models:

DecisionTreeClassifier: A decision tree algorithm for classification tasks. It takes two parameters, the input feature column and the label column, and creates a tree model that can be used to make predictions on new data.

RandomForestClassifier: A random forest algorithm for classification tasks. It takes the same two parameters as the DecisionTreeClassifier, but it creates an ensemble of decision trees to improve accuracy and prevent overfitting.

NaiveBayes: A Naive Bayes algorithm for classification tasks. It assumes that the features are independent of each other and calculates the probabilities of each class based on the joint probabilities of the features. It takes the same two parameters as the other classifiers.

```
[20]: # Train the models (its three models, so it might take some time)
dtc_model = dtc.fit(train_data)
rfc_model = rfc.fit(train_data)
nbc_model = nbc.fit(train_data)
```

The code trains three classification models on the training dataset: a Decision Tree Classifier (dtc), a Random Forest Classifier (rfc), and a Naive Bayes Classifier (nbc).

The fit() method is called on each of these models with the train_data dataset as the input to train them on this data. This is done to learn the relationships between the input features and the output labels.

```
[21]: # Make predictions
dtc_predictions = dtc_model.transform(test_data)
rfc_predictions = rfc_model.transform(test_data)
nbc_predictions = nbc_model.transform(test_data)
```

This code makes predictions on the test data using three previously trained classification models: decision tree classifier (dtc_model), random forest classifier (rfc_model), and Naive Bayes classifier (nbc_model). The predictions are saved in three variables: dtc_predictions, rfc_predictions, and nbc_predictions.

The predictions are made by applying the trained models to the test data using the transform() method. The output of the transform() method is a DataFrame that contains the original columns of the test data plus additional columns for the predicted labels and probability scores.

```
[22]: # Select (prediction, true label) and compute test error
acc_evaluator = MulticlassClassificationEvaluator(labelCol="output",␣
  ↪predictionCol="prediction", metricName="accuracy")
```

This code creates a MulticlassClassificationEvaluator object named acc_evaluator to evaluate the accuracy of the classification models.

The labelCol parameter is set to the name of the column containing the true labels and the predictionCol parameter is set to the name of the column containing the predicted labels.

The metricName parameter is set to "accuracy" to compute the classification accuracy, which is the ratio of correctly classified instances to the total number of instances.

[23]:
```
# Accuracy of each model
dtc_acc = acc_evaluator.evaluate(dtc_predictions)
rfc_acc = acc_evaluator.evaluate(rfc_predictions)
nbc_acc = acc_evaluator.evaluate(nbc_predictions)
```

This code calculates the accuracy of each of the three models (decision tree, random forest, and Naive Bayes classifier) using the MulticlassClassificationEvaluator function from PySpark's pyspark.ml.evaluation module.

The MulticlassClassificationEvaluator is used to evaluate the accuracy of classification models that predict categorical labels. It requires the label column and prediction column as input, along with the metric used for evaluation, which is "accuracy" in this case.

The code computes the accuracy of each model by passing the test data predictions to the MulticlassClassificationEvaluator. The resulting accuracy scores are stored in dtc_acc, rfc_acc, and nbc_acc variables for the decision tree, random forest, and Naive Bayes classifier models, respectively.

[24]:
```
print("Here are the results!")
print('-'*80)
print('A single decision tree had an accuracy of: {0:2.2f}%'.
  ↪format(dtc_acc*100))
print('-'*80)
print('A random forest ensemble had an accuracy of: {0:2.2f}%'.
  ↪format(rfc_acc*100))
print('-'*80)
print('A Naive Bayes had an accuracy of: {0:2.2f}%'.format(nbc_acc*100))
```

```
Here are the results!
--------------------------------------------------------------------------------
A single decision tree had an accuracy of: 84.81%
--------------------------------------------------------------------------------
A random forest ensemble had an accuracy of: 88.61%
--------------------------------------------------------------------------------
A Naive Bayes had an accuracy of: 83.54%
```

This code prints out the accuracy of each model in a formatted way. The accuracy is calculated using the MulticlassClassificationEvaluator from the pyspark.ml.evaluation module.

The accuracy is printed in percentage format with two decimal places.

The results are separated by dashes.

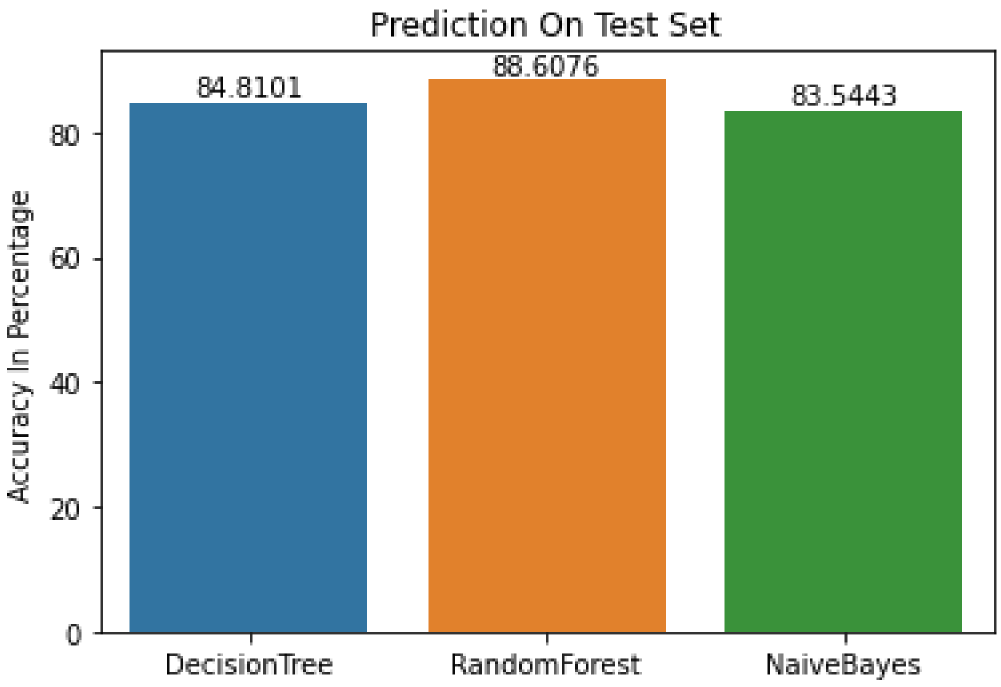
[25]:
```
# Creating PySpark dataframe having the accuracy values
accuracy_df = spark.createDataFrame([
    Row(DecisionTree=dtc_acc*100, RandomForest=rfc_acc*100,␣
  ↪NaiveBayes=nbc_acc*100)
])
```

This code creates a PySpark DataFrame named accuracy_df with the accuracy values obtained for the three models trained in the previous code.

A Row object is created with the accuracy values for each model as its fields, and then this Row object is passed as a parameter to the createDataFrame() method of the SparkSession object named spark.

The resulting DataFrame has three columns, one for each model, and one row with the accuracy values.

[26]:
```
# Visualizing the accuracy values
ax = sns.barplot(data = accuracy_df.toPandas())
for i in ax.containers:
    ax.bar_label(i,)
# Show the plot
plt.title("Prediction On Test Set")
plt.ylabel("Accuracy In Percentage")
plt.show()
```



INSIGHTS

- The figure shows the barplot for each of the classifier. The x-axis representS the classifiers, and the y-axis representS accuracy values.

This code is using the Python libraries seaborn and matplotlib to create a barplot visualizing the accuracy values of the three machine learning models on the test set.

First, a PySpark dataframe accuracy_df is created with the accuracy values for each model. Then, accuracy_df is converted to a Pandas dataframe with the toPandas() method to make it compatible with Seaborn.

Next, a barplot is created with sns.barplot() function from Seaborn, with the data parameter set to the Pandas dataframe. The accuracy values for each model are plotted as vertical bars on the plot.

Lastly, ax.bar_label() is called to add the accuracy values as labels to each bar. plt.title(), plt.ylabel() and plt.show() functions from Matplotlib are called to add a title, a y-axis label and display the plot respectively.

A set of empirical hypotheses that can be tested by analysing the data.

Through this data, one can perform the multi-label classification using the categorical data. One could select any one of the continue variables as the target variable, and use the other continuous variables, cateogrical, and binary variables as independent variable. One can perform the binary classification by selecting the binary variables as target variable.

The statistical analyses you will need to carry out to answer your questions

The correlation matrix to see how the features are correlated with the target vaue. Checking if there are any NULL values in any of the columns, viewing and counting unique elements of categorical variables to know if they are actually categorical or not. Performing the standard scalar to scale the contineous variables. Data Visualization for visualizing the accuracy.

What technologies you intend to use or implement to carry out your analysis.

They are PySpark libraries. For Correlation matrix, used the expression .toPandas().corr().output to convert the spark dataframe into Panda and then get the correlation. Then OneHotEncoder and StandardScaler to perform the one-hot encoding on categorical variables and standarding vaules of the columns of continuous variables, respectively. Using dataframe.cube().count().show() to check if they are actually categorical for not. Matplotlib and Seaborn to perform the data visualization.

Technical challenge of the project

Testing and selecting the variables to know if the accuracy increases or not because by eliminating one variable the accuarcy will increase in one model but at the same time it will decrease in other model. Creating pipeline as checking that pipeline variables are taking right input at every stage and giving right output. Selecting the right scalar because, Naive Bayes does not work properly if the values are negative.