CSC 369 Final Project Report - Predicting Heart Disease

By: Krishnanshu Gupta, Tashin Wasit Amio, Karen Huang

Introduction:

Heart disease is the leading cause of mortality in the United States and globally according to the Centers for Disease Control and Prevention (CDC), claiming approximately 700,000 lives annually in the US and 18 million worldwide. It accounts for one out of every five deaths in the country, with someone dying from it every 33 seconds. The economic burden of heart disease prevention and treatment amounts to nearly $240 billion annually. This research project aims to address the urgent need for machine learning models in understanding and predicting heart diseases and contributing to preventive measures. By leveraging machine learning algorithms, this project aims to identify patterns and relationships within datasets to predict patients' likelihood and risk of heart disease, in the aim of enhancing diagnostic accuracy.

This study explores the application of Machine Learning techniques using Spark and Scala in predicting heart disease. More specifically, the study employed K Nearest Neighbors and also explored the use of Logistic Regression on a Kaggle Dataset, titled Indicators of Heart Disease.

Methodology:

In this project, we'll initially be parsing and cleaning the CSV data in and converting it to an RDD format. To perform some exploratory data analysis, we'll create some basic charts and visualizations to observe the distributions within the data. Next, we plan to utilize a completely custom implementation of K Nearest Neighbors as our main algorithm for predicting the dataset, following a train-test split of 70% in train. We'll be calculating the metrics of Precision, Recall, Sensitivity, Accuracy, and F1-score, as well as a few others, to evaluate the model. Additionally, we'll also perform a basic implementation of Logistic Regression using the Spark ML library for testing purposes and to compare results to KNN.

Dataset Parsing and Cleaning

We obtained the dataset from Kaggle as mentioned above, and it was pre-formatted as a CSV file and mostly cleaned already. We load the CSV data file into an RDD format using SparkContext (sc.textFile). Each line of the dataset is split by commas (line.split(",")), and the fields are mapped to a case class HealthData. We use pattern matching (fields match) to handle different cases in the data and mapping those to an Array for easier element access.

```
case class HealthData (heartDisease: Double, bmi: Double, smoking: Double, alcohol: Double, stroke: Double,
 physicalHealth: Double, mentalHealth: Double, diffWalking: Double, sex: Double, age: Double, race: Double,
 diabetic: Double, activity: Double, generalHealth: Double, sleep: Double, asthma: Double, kidneyDisease: Double,
 skinCancer: Double)
```

```
val processData = data.flatMap { line =>
  val fields = line.split(",")
  fields match {
    case Array(hd, b, s, a, st, ph, mh, dw, sex, age, race, d, act, gh, sl, ast, kd, sc) =>
      Some(HealthData(
```

We'll then be cleaning and converting all the data into decimal format. For binary data, "Yes" will become 1.0 and "No" becomes 0.0. For the categorical data columns of Age, Race, and General Health, we'll be using ordinal encoding to convert the categories to integer values starting from 1.0 onwards. For the numerical data columns, we'll simply be converting them to double for consistency with the other columns. Listing an example of this process below from the code.

```
ph.toDouble,
mh.toDouble,
if (dw == "Yes") 1.0 else 0.0,
if (sex == "Male") 1.0 else 0.0,
age match {
  case "18-24" => 1.0
  case "25-29" => 2.0
  case "30-34" => 3.0
  case "35-39" => 4.0
  case "40-44" => 5.0
  case "45-49" => 6.0
  case "50-54" => 7.0
  case "55-59" => 8.0
  case "60-64" => 9.0
  case "65-69" => 10.0
  case "70-74" => 11.0
  case "75-79" => 12.0
  case "80 or older" => 13.0
  case _ => 0.0
},
```

If any of the rows in the file doesn't have all 18 of the columns, we'll be removing that from the dataset by mapping it to a None. This allows us to clean all the invalid or malformed data, and handle missing values.

```
case _ => None
```

Next, we'll be saving this cleaned and parsed data into a text file in order to be used in the KNN and Logistic Regression algorithms, which are the next steps. We'll use saveAsTextFile() to do this.

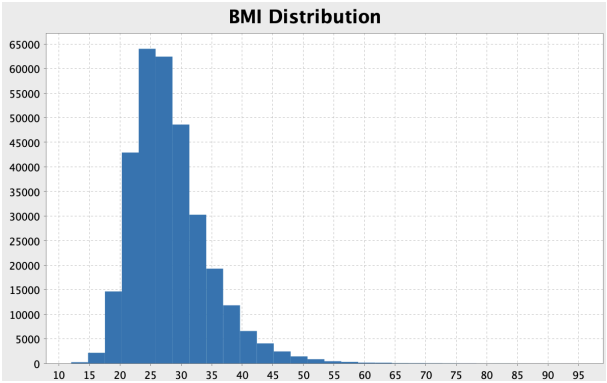
```
val clean_data = processData.map(h => h.productIterator.mkString(", "))
clean_data.saveAsTextFile("/Users/krishnanshugupta/Cal Poly/369_CSC/testspark/src/test.txt")
```

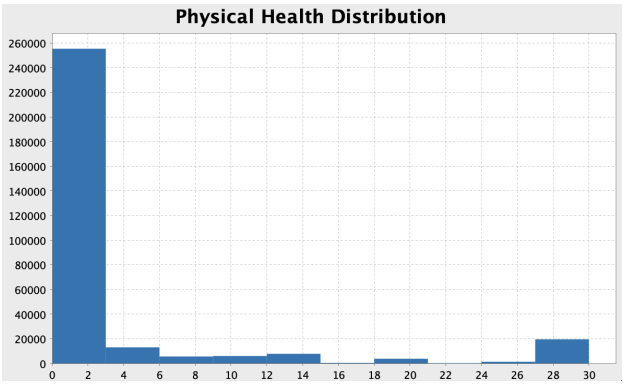
Data Exploration:

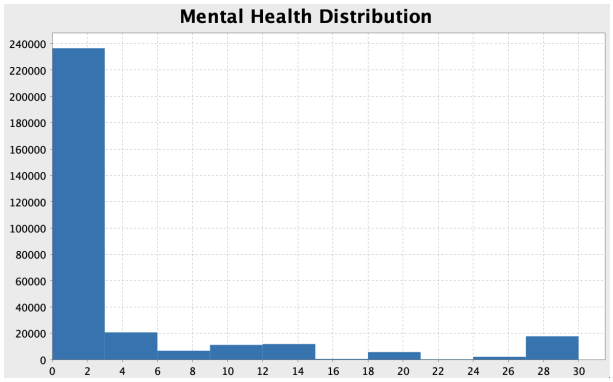
After data parsing and cleaning, we conducted exploratory data analysis (EDA) to gain some insights into the data distribution. We utilized the Breeze Plot library to create histograms for various features such as BMI, physical health, mental health, sleep duration, age, race, and general health. These histograms provide visual representations of the distributions of these variables within the dataset. For the categorical variables of age, race, and general health, the x-axis labels use the ordinal encoding conducted in the previous step. This is an example of how to plot these distributions:

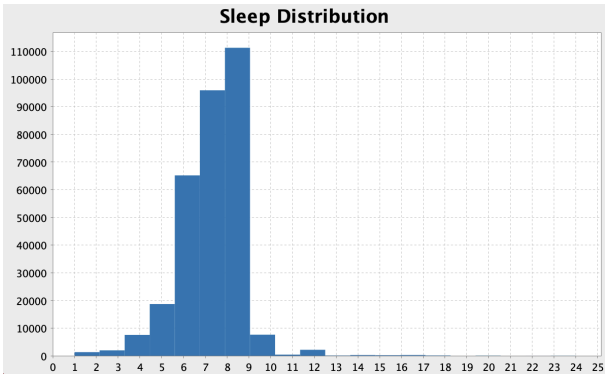
```
val bmi = processData.map(_.bmi).collect()
val f1 = Figure()
val p1 = f1.subplot(0)
p1 += breeze.plot.hist(bmi, 30)
p1.title = "BMI Distribution"
```

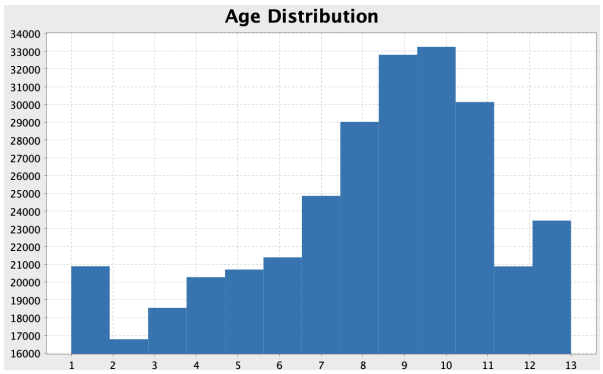
Attaching the plots below:

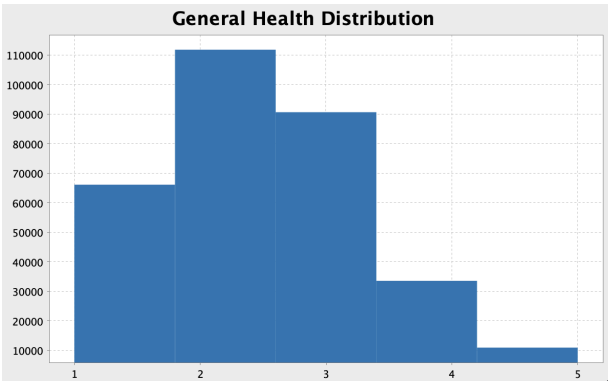


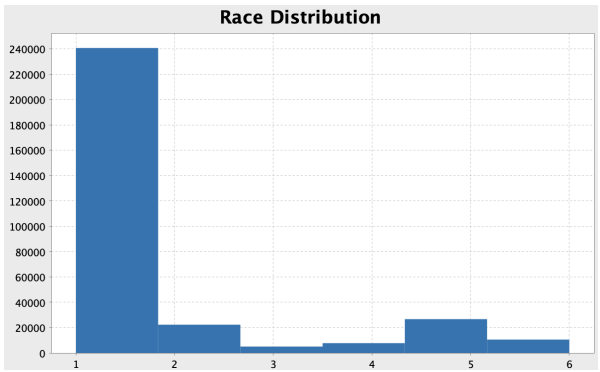








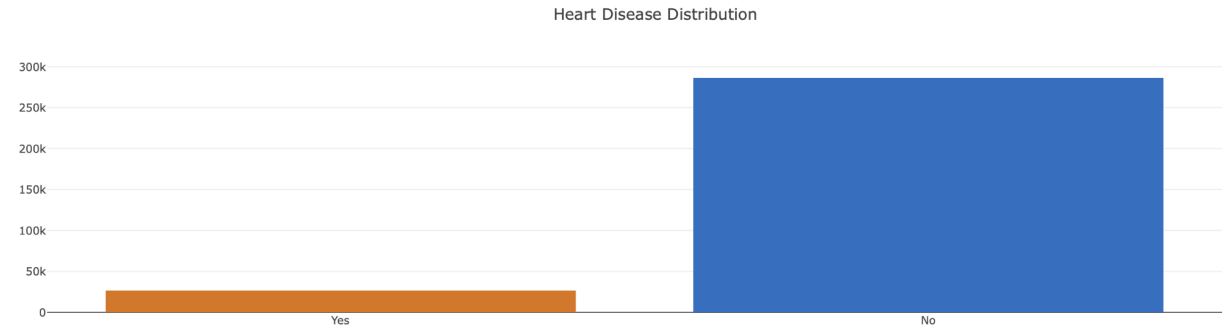




Additionally, we visualize the distribution of the heart disease target column using Plotly, where we calculate the counts of "Yes" and "No" instances and create a bar chart to illustrate the distribution. This is the code we used to plot this:

```
val yesCount = processData.map(_.heartDisease).sum().toInt
val noCount = processData.count().toInt - yesCount
val chart = Seq(
 Bar(Seq("Yes"), Seq(yesCount)).withMarker(Marker().withColor(Color.RGB(214,
120, 47))),
 Bar(Seq("No"), Seq(noCount)).withMarker(Marker().withColor(Color.RGB(56, 111,
194)))
)
var layout = Layout(title = "Heart Disease Distribution")
Plotly.plot(traces = chart, layout = layout, path =
"/Users/krishnanshugupta/Cal Poly/369_CSC/testspark/src/charts.html")
```

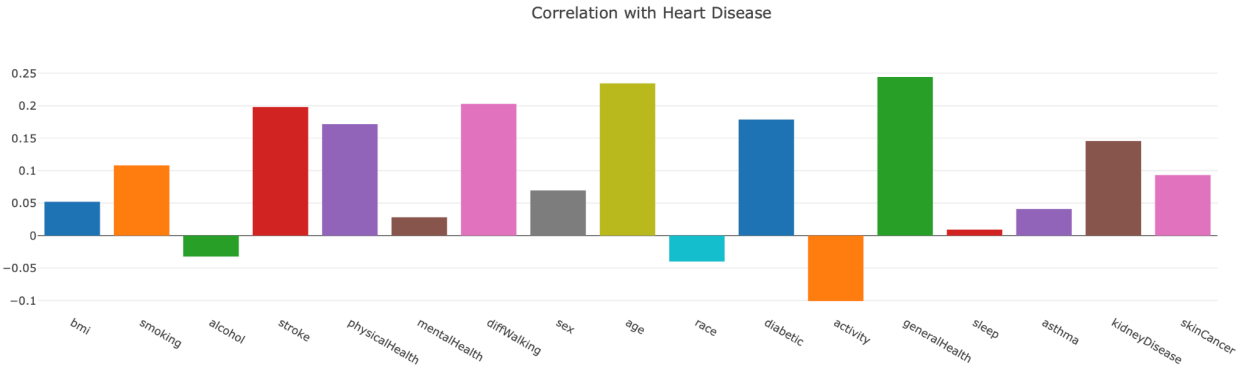
This is the plot created:



Furthermore, we analyzed the correlation between different features and heart disease using Spark SQL. We first converted the RDD from before to a Spark Dataframe, and then calculated the correlation coefficient for each feature with the target column using df.stat.corr, and generated a bar chart to display the correlations. This can help display a basic understanding of the relationship between each feature and heart disease. This is the code we used to convert the RDD to Dataframe and then plot it using Plotly:

```
val df: DataFrame = processData.toDF()
val correlations = df.columns.filter(_ != "heartDisease").map { column =>
 val correlation = df.stat.corr("heartDisease", column)
 (column, correlation)
}
val chartData = correlations.map { case (column, correlation) =>
 Bar(Seq(column), Seq(correlation))
}
layout = Layout(title = "Correlation with Heart Disease")
Plotly.plot(traces = chartData, layout = layout, path =
"/Users/krishnanshugupta/Cal Poly/369_CSC/testspark/src/charts.html")
```

This is the created correlation chart, which displays GeneralHealth as the column with the highest correlation with the target column.



K Nearest Neighbors:

The KNN algorithm is a simple, yet effective method used for classification and regression tasks.

It is a good fit for our chosen dataset for the following reasons:

- *Simplicity and Intuitiveness:* KNN is inherently simple and easy to understand. It makes predictions based on the 'closeness' of data points in the feature space.
- *Non-linear Relationships*: KNN can capture non-linear relationships between variables without the need for explicit modeling of these relationships. This is beneficial for our dataset since the relationship between indicators (like BMI, sleep, etc.) and outcome (heart disease) can be complex and non-linear.
- *Versatility in Feature Types*: KNN can handle a mix of continuous and categorical features, which is common in medical datasets. For example, the Personal Key Indicators of Heart Disease dataset includes a variety of feature types such as age (continuous), smoking status (categorical), and BMI (continuous).

The program is structured to perform the following operations:

1. *Data Loading and Preprocessing:*
   Loads the dataset from the specified files and parse it. This step transforms the raw text data into a usable format for ML (in this case, an RDD of numeric arrays), and then shuffles the data to randomize the order, which is important for splitting into training and test sets later.

```
val filePaths = "src/output/part-00000,src/output/part-00001"
val rawData = sc.textFile(filePaths)
val parsedData = rawData.map(_.split(", ").map(_.toDouble))

val collectedData = parsedData.collect()
val shuffledData = Random.shuffle(collectedData.toList)
```

2. *Splitting the Data:*
   Determine the index for splitting the shuffled data into training and test sets based on a predefined fraction. This is crucial for evaluating the model's performance on unseen data.

```
val fraction = 0.7
val split_idx = round(shuffledData.size * fraction).toInt
val test = shuffledData.take(split_idx)
val train = shuffledData.drop(split_idx)
```

3. *Model Training and Prediction:*
   Broadcast the K-value and the training dataset across all nodes in the Spark cluster to ensure that these are readily available for distributed computation. Use the K-Nearest

Neighbors algorithm to predict the label of each test data point based on the majority vote of its K nearest neighbors in the training set. For our implementation, we calculated nearest neighbors using Euclidean distance.

```
val k = 21
val k_val = sc.broadcast(k)
val train_info = sc.broadcast(train)

val test_rdd = sc.parallelize(test)
val entryLblList = test_rdd.map(x => ...)
```

4. *Evaluation:*
   Evaluate the model's predictions by calculating precision, recall, and F1 score based on the counts of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). These metrics provide insights into the model's performance, specifically its accuracy and ability to handle imbalanced data.

```
val counts = results.countByValue()

val analyze: Map[String, Int] = counts.map {
  case (key, value) => (key, value.toInt)
}.toMap

// Extract the counts
val TP = analyze.getOrElse("TP", 0)
val TN = analyze.getOrElse("TN", 0)
val FP = analyze.getOrElse("FP", 0)
val FN = analyze.getOrElse("FN", 0)

// Calculation of various metrics
val accuracy = if (TP + TN + FP + FN > 0) (TP + TN).toDouble / (TP + TN + FP + FN) else 0.0
val precision = if (TP + FP > 0) TP.toDouble / (TP + FP) else 0.0
val recall = if (TP + FN > 0) TP.toDouble / (TP + FN) else 0.0
val f1Score = if (precision + recall > 0) 2 * (precision * recall) / (precision + recall) else 0.0
val specificity = if (TN + FP > 0) TN.toDouble / (TN + FP) else 0.0
val negativePredictiveValue = if (TN + FN > 0) TN.toDouble / (TN + FN) else 0.0
val falsePositiveRate = if (FP + TN > 0) FP.toDouble / (FP + TN) else 0.0
val falseDiscoveryRate = if (TP + FP > 0) FP.toDouble / (TP + FP) else 0.0
val falseNegativeRate = if (TP + FN > 0) FN.toDouble / (TP + FN) else 0.0
val mcc = if ((TP + FP) * (TP + FN) * (TN + FP) * (TN + FN) > 0)
  ((TP * TN) - (FP * FN)).toDouble / Math.sqrt(((TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)).toDouble)
else 0.0
```

Logistic Regression:

We also explored using a Logistic Regression algorithm which is a statistical method used for binary classification. It models the probability that a given input belongs to a particular category (e.g., presence or absence of heart disease) based on one or more independent variables. The output is transformed using a logistic function to ensure it ranges between 0 and 1, representing the probability of belonging to the positive class.

Logistic Regression is fitting for predicting heart disease because it directly models the

probability of occurrence of an event (having or not having heart disease) based on input variables (such as age, BMI, lifestyle factors). Its output, which ranges between 0 and 1, is interpretable as the likelihood of heart disease presence. **However**, as we dove deeper into the implementation of Logistic Regression, we encountered a series of challenges that ultimately led us to reconsider the choice. One significant hurdle was the extensive reliance on third-party libraries which did not align with the project's requirements/preferences. But the most disheartening revelation came from evaluating the model's performance. Despite our efforts in fine-tuning the logistic regression model by trial and error and ensuring the data was appropriately preprocessed, the accuracy of our predictions was disappointingly similar to that of a coin toss.

```
// Assemble the features to be used by the logistic regression model
val assembler = new VectorAssembler()
      .setInputCols(Array("bmi", "smoking", "alcohol", "stroke", "physicalHealth",
        "mentalHealth", "diffWalking", "sex", "age", "race", "diabetic", "activity",
        "generalHealth", "sleep", "asthma", "kidneyDisease", "skinCancer"))
      .setOutputCol("features")
```

```
// Evaluate the model using area under ROC
val evaluator = new BinaryClassificationEvaluator()
  .setLabelCol("heartDisease")
  .setRawPredictionCol("prediction")
  .setMetricName("areaUnderROC")

val accuracy = evaluator.evaluate(predictions)
println(s"Area under ROC = $accuracy")
```

```
Area under ROC = 0.5002725880783842
```

Results and Discussion:

```
TP: 2650
TN: 200335
FP: 6532
FN: 1974
Accuracy: 0.9598
Precision: 0.2886
Recall(sensitivity): 0.5731
F1 Score: 0.3839
Specificity: 0.9684
Negative Predictive Value: 0.9902
False Positive Rate: 0.0316
False Discovery Rate: 0.7114
False Negative Rate: 0.4269
Matthews Correlation Coefficient (MCC): 0.3886
```

Quite frankly, the results from KNN we obtained were subpar. As far as pure accuracy goes, our value was impressively high due to the sheer number of true negatives we have. However, given that we were working with an imbalanced dataset, accuracy is not a particularly good measure for the performance of our model. Better indicators would be the F1 score and Matthews Correlation Coefficient, which show that our model is not performing well.

To analyze the results we've obtained in reference to the real world data, our model is extremely good at accurately recognizing people who do not have heart disease as not having heart disease (specificity). The more important factor is being able to accurately detect patients who have heart disease as truly having heart disease so that they can be treated. From this snapshot, it's clear that our model is not doing a great job at this. As seen from sensitivity/recall, our model is only accurately detecting about 57% of the people who have heart disease and marking the other 43% as not having heart disease. From the precision, we can see that out of the people detected as having heart disease, only 28% of them actually have it. In this scenario, we prioritize recall over precision, given that it's more important that the people with heart disease are detected and treated over the inconvenience to the other ~6000 people who were incorrectly flagged. It's good that our recall is higher than our precision, but regardless, both values are much lower than we'd like to see.

So why is our model doing so poorly?

For one, KNN doesn't work well with imbalanced datasets. This is because for every new datapoint (test value), it will compute a distance to every point in the training set and obtain the nearest k points. The label for the test value is determined by whichever label the majority of the nearest k points have. As a result, if there is a majority class, there will be a higher chance that those labels will be present in the nearest k points. Since we did not separate the data by label and take a sample from each prior to shuffling and splitting, our model has a bias for marking points as negative for heart disease. While this accounts for some of our false negatives, this doesn't explain our large number of false positives.

Another problem with our model is while we converted the categorical variables in numerical values to make it possible to compute distances, we did not normalize the data. Since KNN works by finding distances, any large values will drastically affect the distance calculation. For example, in our data, BMI values tended to be very large. Thus, even if every other feature in the new test value matched with many training values that were negative for heart disease, they may not be marked as a nearest neighbor if their BMI values differed enough.

Finally, KNN suffers from the "curse of dimensionality," in which performance heavily degrades if there are too many features. When there are too many features, every point essentially becomes an "outlier" in that there will be very few points nearby. Distances will also essentially become meaningless, as points will start becoming equidistant to each other. Since we have 17 features, there is reason to believe that this may also contribute to our poor performance. Some possible solutions here would be to use a different distance function or to perform feature selection to reduce the number of features.