Heart Disease Predictive Modeling

June 2023

Abstract

Heart disease is a leading cause of mortality worldwide, emphasizing the need for accurate and reliable predictive modeling techniques to identify individuals at high risk [2]. Heart disease predictive modeling involves statistical and machine learning algorithms to analyze large datasets and predict the likelihood of an individual developing heart disease. The aim of this report is to try and predict the probability of heart disease by implementing two machine learning models. One of the methods that were used was Logistic Regression which output a model with high accuracy and fast computational speed after implementing Gradient Descent into the model. The second method is Support Vector Machine which performs better in accuracy by implementing Stochastic Gradient Descent. SVM is able to achieve almost a 10% increase in accuracy compared to Logistic Regression after Stochastic Gradient Descent is applied to the original model. With these findings, the best model is selected by not only focusing on accuracy and speed but also by focusing on the prediction of false positives and false negatives.

1 Introduction

Data analysis plays a crucial role in various fields, including medicine, where it can uncover new insights and validate existing knowledge. American industries are increasingly recognizing the importance of data science and data analysis, leading many companies in the U.S. to strengthen their departments in these areas to enhance their decision-making processes. One area that can significantly benefit from data analysis is healthcare, particularly in assessing the risk of heart disease. The possibility of experiencing heart disease can have a profound impact on an individual's life, both personally and on a global scale.

Through the analysis of the data set, we aim to uncover significant patterns, correlations, and risk factors associated with the likelihood of a heart attack. This knowledge can then be utilized to build a robust machine learning model that optimizes the prediction of heart attack possibilities. Such a model has the potential to enhance the accuracy and efficiency of diagnosing the risk of heart disease, leading to improved patient outcomes and potentially substantial cost savings for both individuals and the healthcare system as a whole.

Accurate heart disease predictive models offer several potential applications in clinical settings. They aid healthcare professionals in timely interventions and targeted preventive measures while providing insights into risk factor prevalence and distribution. Additionally, heart disease predictive models can support public health initiatives and policy-making by providing insights into the prevalence and distribution of heart disease risk factors.

However, challenges exist in heart disease predictive modeling, such as data quality issues, feature selection biases, model interpretability, and ethical considerations. Ensuring data privacy and security, addressing bias and fairness concerns, and maintaining transparency in model development and deployment are crucial for the ethical and responsible use of predictive modeling in heart disease management.

1.1 Description of the dataset

The data set used for this project comes from the UCI Machine Repository, which extracted data from a Cleveland database. A subset consisting of 14 variables was utilized[1].

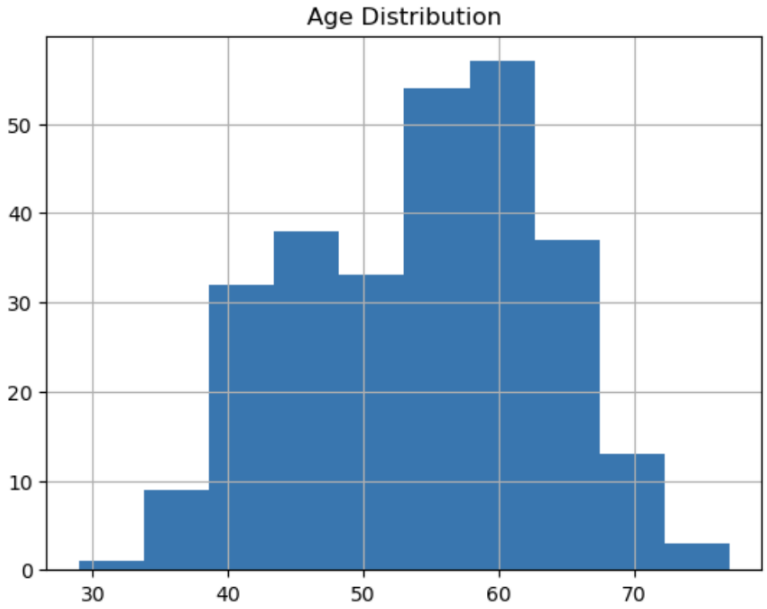
Table 1: Heart Disease Data set

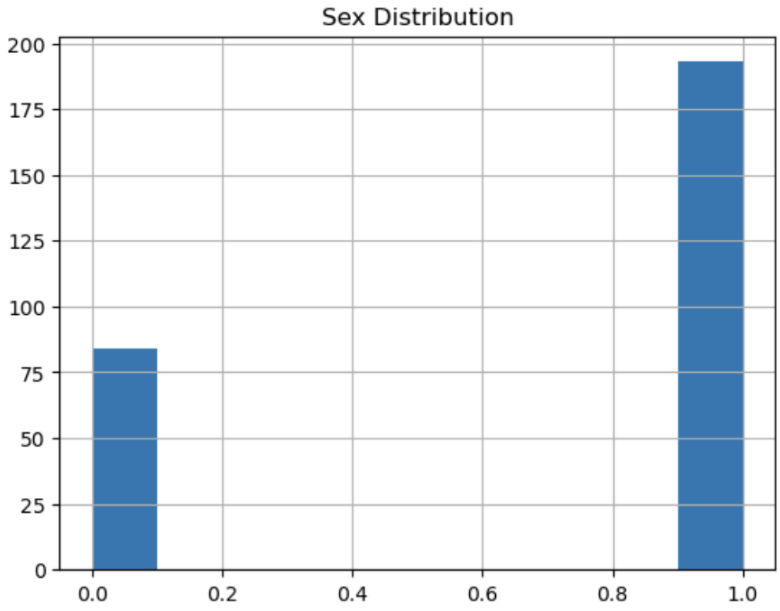
Feature	Description
Age	Age of the patient
Sex	1=male; 0=female
CP	Chest pain type (1=typical anigma; 2=atypical anigma; 3=non-anigmal pain; 4=asymptomatic)
Trestbps	Resting blood pressure
Chol	Serum cholestoral in mg/dl
FBS	Fasting blood sugar over 120 mg/dl (1 = true; 0 = false)
RestECG	Electrocardiographic results (0=normal; 1= having ST-T wave abnormality; 2= ventricular hypertrophy)
Thalach	Maximum heart rate achieved
Exang	Exercise induced angina (1 = yes; 0 = no)
Oldpeak	ST depression induced by exercise relative to rest
Slope	The slope of the peak exercise ST segment (1=upsloping; 2=flat; 3=downsloping)
CA	Number of major vessels (0-3) colored by flourosopy
Thal	3 = normal; 6 = fixed defect; 7 = reversable defect
Target	Presence of heart disease in patient (0=Benign; 1=Manignant)

2 Data Exploration

To clean the data, categorical and binary variables that were once defined as integers were changed to objects. This was done to ensure that the variables to be used in regression were most accurately represented. As a result, numerous missing values needed to be removed. For example, the variable CA ranges from 0-3 vessels; however, the data set included a 4th vessel, so those values had to be removed. Outstanding outliers were also removed, reducing the data set from 303 entries to 277.

The summary statistics were computed regarding the demographics of the patient and found that the age of most of the patients in the dataset are around their 60s. It was also observed that twice as many men as women were represented in the data.





2.1 Conditional Entropy

Conditional entropy measures the amount of uncertainty or randomness in the target variable (in this case, the possibility of getting heart disease) given the values of a particular feature. A lower conditional entropy indicates that the feature provides more information about the target variable and helps reduce uncertainty. The goal is to find most significant risk factors that affect getting heart disease.
Here, $P(y, x)$ is the joint probability distribution of Y and X, $P(y|x)$ is the conditional probability of Y given X, and $P(y)$ is the marginal probability distribution of Y.

$$H(Y|X) = -\sum_{y \in Y} \sum_{x \in X} P(y, x) \log \frac{P(y|x)}{P(y)}$$

The conditional entropy of the 'Target' was calculated which is a binary variable with 0 = Less chance of getting heart disease and 1 = Greater chance of getting heart disease; for every uni-variate combination. The purpose of the table below is to help select the features with the highest amount of information to gain with respect to predicting a heart disease.

Table 2: Conditional Entropy of Target Given Features

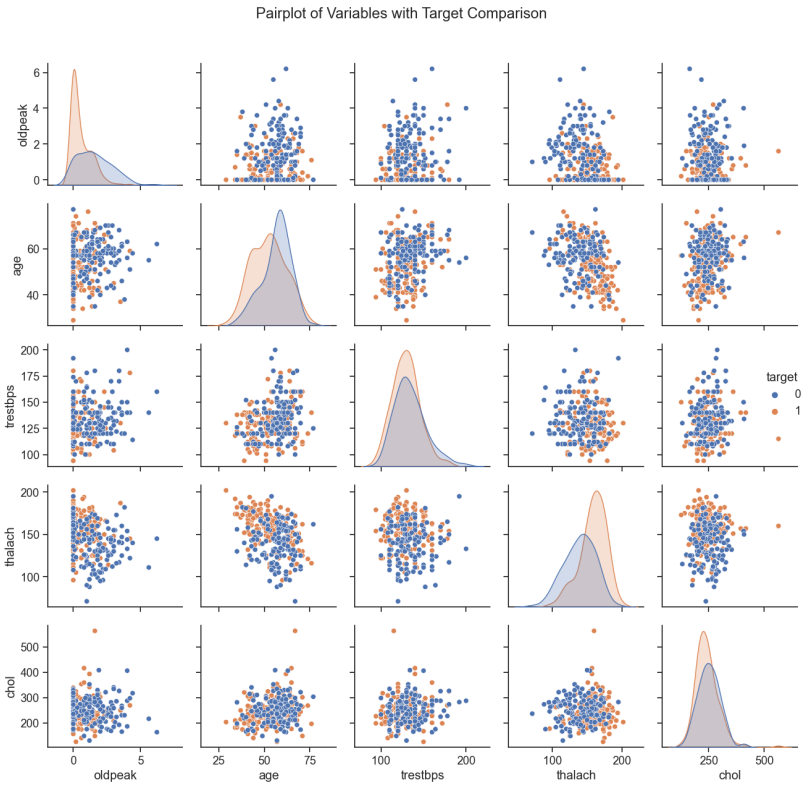
Feature	Conditional Entropy
Age	0.163
Sex	0.523
CP	0.357
Trestbps	0.162
Chol	0.132
FBS	0.498
RestECG	0.397
Thalach	0.143
Exang	0.542
Oldpeak	0.172
Slope	0.399
CA	0.324
Thal	0.347

Now, the top 5 features with the lowest conditional entropy values need to be picked: Cholestrol = 0.132, Thalatach = 0.143, Age = 0.163, Trestbps = 0.162, Oldpeak = 0.172. These features, with their low conditional entropy values, suggest that they hold valuable information and are important factors to consider when assessing the likelihood of a heart attack.

2.2 Pairwise Comparison

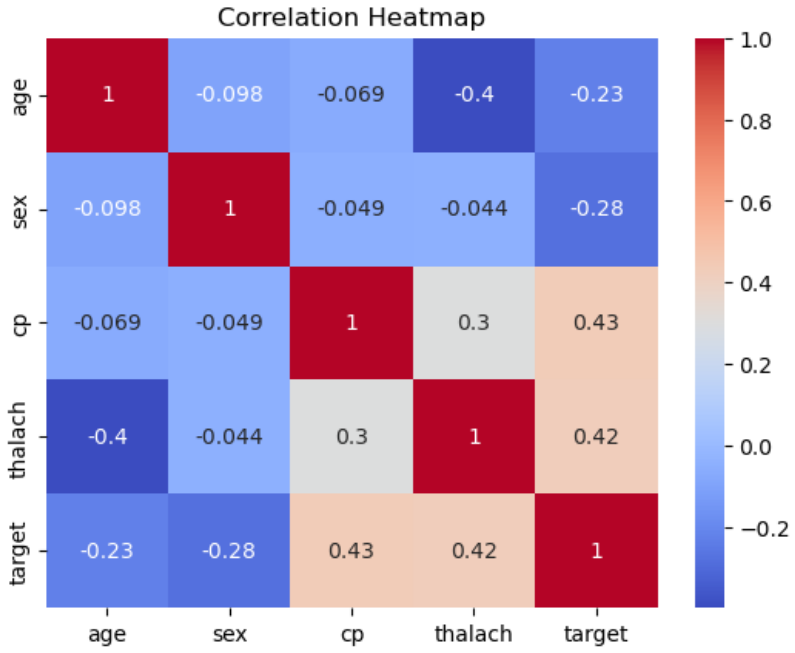
Understanding the relationship between the five variables with the lowest conditional entropy and their impact on the "Target" variable is crucial. It is necessary to investigate the relationship between selected variables and the target groups using a pair-plot analysis. By visualizing the selected variables against the target groups, we aim to gain insights into potential patterns, distributions, and discriminatory power of the variables. From the scatter-plots, it can be identified whether or not the variables have a positive or negative relationship with each other. For example, Age and Chol having a positive relationship with each other. As for the distributions they all seem to be normal.

This analysis is pretty helpful to explore the relationship between selected variables and the target groups. The results provided initial insights into potential patterns and distributions among the variables. These findings can serve as a basis for further investigation and feature selection in predictive modeling or risk assessment related to heart attack probability.



2.3 Correlation

After the preliminary steps of looking at the correlation with each other are done. The results can be expressed in a heatmap for better visualization.



The correlation heatmap is generated using the Pearson correlation coefficient. The formula to calculate the Pearson correlation coefficient between two variables X and Y is as follows:

The Pearson correlation coefficient, denoted as r, is calculated using the following formula:

$$r = \frac{\sum((X_i - \overline{X})(Y_i - \overline{Y}))}{\sqrt{\sum(X_i - \overline{X})^2} \cdot \sqrt{\sum(Y_i - \overline{Y})^2}}$$

where X_i and Y_i are the individual values of variables X and Y respectively, and $\overline{X}$ and $\overline{Y}$ are their respective means.

3 Proposed Methods

3.1 Logistic Regression

In the analysis, Logistic Regression was the chosen model to predict the accuracy of our dataset. Logistic regression is particularly well-suited for modeling binary outcomes, which aligns with the nature of the dataset used in the project. The dataset consists of various variables, including the "target" variable, which serves as the outcome of interest. The "target" variable is binary, indicating the presence or absence of a particular condition or event. Logistic regression is an ideal choice when working with such binary outcomes because it allows us to estimate the probability of an event occurring based on the values of the predictor variables. By using logistic regression, the aim is to model the relationship between the selected predictor variables and the likelihood of the "target" variable being 0 or 1. This approach is used to help understand the impact of each predictor on the binary outcome and make predictions based on their influence. The advantage of Logistic Regression lies in its ability to handle binary outcomes efficiently and provide interpretative results. It estimates the odds of the event occurring and transforms them into probabilities using the logistic function. The logistic function restricts the predicted values to the range between 0 and 1, aligning with the binary nature of the "target" variable. By analyzing the coefficients and significance levels obtained from the logistic regression model, we can determine the variables that have a significant impact

on the likelihood of the binary outcome. This knowledge will help us understand which factors are crucial in predicting the presence or absence of the condition under investigation.

For the project, the Sigmoid function was applied. It works by squishing any value to fit between 0 and 1:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Its output is interpreted as the predictability of a person developing heart disease. At the same time, the use of a cost function is needed to see how good the model is at predicting who has heart disease, and how big or small the error is. The goal is to minimize said error and it's done by using Gradient Descent. This is going to iterate the function until it converges to the local minima and then it will stop[3].

3.2 Support Vector Machine

Finding a balance between increasing accuracy and still managing computational time is a difficult task, so implementing a Soft Margin Support Vector Machine "SVM" classification model was a natural choice because it is extremely versatile and can handle high dimensional data efficiently. This was also validated when finding other authors that worked on the same data set finding SVM as a successful method in determining the given target [STA 141C Example 2].

An SVM algorithm finds separating hyperplane that will separate the pre-defined probability of a heart attack. There is a margin that exists between the defined hyperplane and the first data point. It is imperative that a hyperplane margin for an SVM is optimized because then the model will have higher accuracy. This relationship is due to the hyperplane being able to best determine the target variable more effectively if there is a larger margin between the targets.

Sometimes creating a hyperplane that is capable of perfectly separating the target variables are infeasible due to noise or outliers in the data or it can be less accurate because this may cause the model to have a difficult time assessing overlapping targets. It is important to understand that the data points that are within our project are not perfect and since there is high dimensionality it is difficult to understand all aspects of the data, so applying a soft margin can alleviate some of these issues. A soft margin SVM introduces a slack variable that allows for misclassification and thus you can maximize the margin because it accounts for variables being on the wrong side of the hyperplane.

A Soft Margin Support Vector Machine is a linear programming optimizations problem looking to achieve:

$$min_{w,\xi_i,\beta} \lambda ||w||_2^2 + ||\xi||_1$$

$$s.t. \;\; y_i(w^T x_i + \beta) \geq 1 - \xi_i \;\; for \; i = 1, \ldots, N$$

$$\xi_i \geq 0 \;\; for \; i = 1, \ldots, N$$

where w is the margin, λ is the regularization parameter, ξ is the error for misclassification. To apply an SVM the target variables were changed from 0/1 to -1/1 because this would yield the highest accuracy

When applying a soft margin SVM to a real world problem, there are multiple methods in python that are capable of applying such an algorithm. However, the following will be exploring the difference between using the commonly used package scikit-learn versus applying it using base packages. The purpose of the

project is to optimize an algorithm so that it can determine the target accurately and thus health care providers can use these findings to better help assess patients. Thus when comparing the package scikit-learn SVM method to the base package method all variables and methods outside of the SVM algorithm will stay the same. For example, using the StandardardScaler() from scikit-learn helped increase the accuracy significantly and wasn't costly to the efficiency, so a similar algorithm was built using the base packages for the custom algorithm. There were multiple methods that were used to find an optimal solution.

When determining methods that would help increase the efficiency of an SVM algorithm, it seemed clear that Gradient Descent would be an efficient and simple method that would work with large data sets. Gradient Descent is a method to optimize algorithms by using iterations to adjust parameters so that the cost function is minimized in such a way that the error is decreased. This is a very powerful optimization tool and there are multiple methods that can be used to apply it, the following two algorithms are exploring batch Gradient Descent(GD) versus Stochastic Gradient Descent (SGD). The reason these two methods are valuable to explore is because it illuminates how different gradient descents in the algorithm can vastly change the accuracy.

A batch gradient descent uses the entire data set for training and updates the parameter in respect to the entirety of the data set. A stochastic gradient descent uses random small subsets of the data to train on and then updates the parameters, and it repeats this process with all instances of the data. Clearly, this will result in different accuracy's because the methods vary.

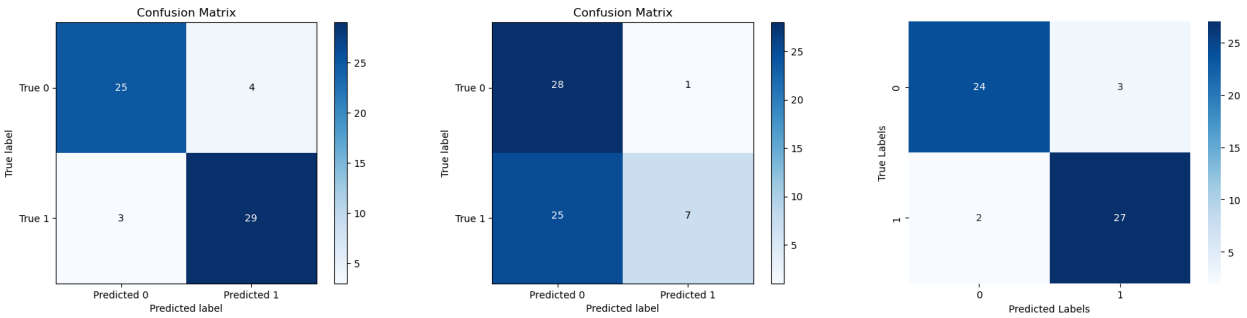
Thus, scikit-learn, SVM with GD, and SVM with SGD will be compared by the computational time and accuracy to determine a method that preforms the best.

4 Data Analysis Study

4.1 Logistic Regression

For the analysis, the model used was regular Logistic Regression optimized through Gradient Descent to predict the accuracy of our dataset. This approach was implemented with and without the aid of pre-existing packages. Gradient Descent is an iterative optimization algorithm that seeks to minimize the cost or loss function associated with Logistic Regression. It updates the coefficients of the Logistic Regression equation by iteratively moving in the direction of steepest descent to find the optimal values. By adjusting the coefficients, the model can better predict the binary outcome. By employing both manual implementation and pre-existing packages, it is posible to compare the results, validate the implementation, and ensure the accuracy in the prediction. The comprehensive approach gives insights into the Gradient Descent optimization process and leverage the advantages of available Logistic Regression packages.

The Logistic Regression model from scratch has a learning rate of 0.1. Then a forward-propagation method was constructed so that when an input data is given, the result of the predicted probability from the Sigmoid function will be the output. To make the prediction, the classification was that any probability bigger than 0.6 would be classified as 1 and any result less or equal to 0.6 would be classified as 0.



(a) Logistic Regression (b) Gradient Descent (c) Gradient Descent No Packages

Figure 1: Confusion Matrices

Summary of Findings:

Method	Accuracy	Computational Time
scikit-learn Logistic Regression	88.56%	0.027
scikit-learn Gradient Descent	57.27%	0.063
Logistic Regression w/ GD No packages	91.07%	0.002

After trying to optimize the data using Gradient Descent from packages it performed considerably worse than doing a normal Logistic Regression in all criteria. However, it is worth noting that for this dataset Logistic Regression seems like a safe option. When dealing with big data and as this data set will probably get more and more complex a Logistic Regression which is a parametric model may encounter limitations in terms of scalability. Luckily, after optimizing the model by doing Logistic Regression and Gradient Descent from scratch it increased the accuracy, and the computation speed was the fastest by a huge margin. In this scenario, that is the model it should always be used.

4.2 Soft Margin Support Vector Machines

The following are the results found from the analysis of the Soft Margin Support Vector Machine. These are some techniques that were assessed but were not ultimately used:

- Changing kernel to linear, poly, and sigmoid, these methods were not used because they decreased accuracy significantly
- Changing kernel coefficient from a method that uses $\frac{1}{n_{features} * X.var()}$, this did note help accuracy. (This method was something expressed in the scikit-learn package as a possible change that could be made)
- Regularization parameter to L1 regularization, this ultimately didn't work because it didn't increase accuracy
- Cross Validation using Leave-one-out, this method didn't significantly help the accuracy and increased the computational speed.

After applying different algorithms to see what would increase the accuracy and decrease the computational speed for both the scikit learn method and for the base package method, the following was applied to both:

- Changing kernel to radial basis function
- Changing kernel coefficient from a method that uses $\frac{1}{n_{features}}$

- Regularization parameter to L2 regularization
- Cross Validation using K-means, with k = 7.

4.2.1 Support Vector Machines without Packages

Note: LU decomposition and QR factorization were assessed to see if they would help decrease the computational time, however they did little to effect it and would increase it in some occasions. This is most likely because the data was so large that this break down was not efficient. Moreover this method does take more memory then applying an algorithm without it, so this means that it is very memory expensive for an already large data set. These methods are computationally expensive. Thus the following algorithms were built without it.

4.2.2 Support Vector Machine with Gradient Descent

Using a SVM with batch gradient descent yielded an accuracy of 94.00% with computational time of 0.022 seconds.

4.2.3 Support Vector Machine with Stochastic Gradient Descent

Using a SVM with stochastic gradient descent yielded an accuracy of 99.08% with computational time of 0.023 seconds.

4.2.4 Support Vector Machines with scikit-learn Packages

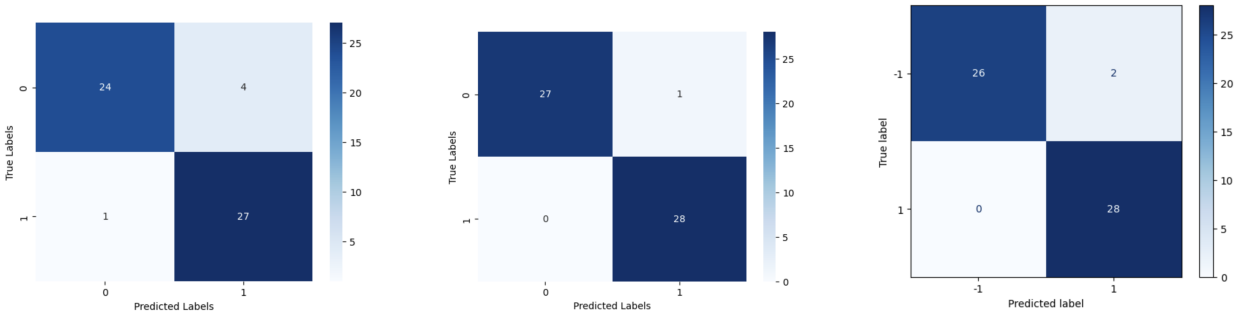
Scikit-learn SVM algorithm imposes an Sequential Minimal Optimization (SMO).
A SMO is a optimize method that creates multiple smaller problems out of the main optimization problem. Then the breakdown helps to find optimal Lagrange multipliers that are correlated to the support vectors, and are iterativly updated till it converges towards the solution. This is specifically useful in SVM because it assists in finding optimal support vectors.

This algorithm was much simpler when it came to coding as we used the scikit-learn packages to obtain an accuracy of 95.65% and a computational time of 0.023 seconds.

4.2.5 Support Vector Machine Findings

Summary of Findings:

Method	Accuracy	Computational Time
SVM w/ GD	94.00%	0.022
SVM w/ SGD	99.08%	0.023
scikit-learn SVM	95.65%	0.023



(a) SVM w/ GD (b) SVM w/ SGD (c) scikit-learn

Figure 2: Confusion Matrices

After exploring using the scikit-learn packages and building two manually, it is clear that creating the customized SVM using stochastic gradient descent is the best method because it has the highest accuracy and has a small computational time. Even though the custom SVM with gradient descent technically had the fastest computational time, it is evident that the accuracy suffered. Moreover, the scikit-learn method had a lower accuracy then the custom SVM with SGD and the exact same computational time. Thus the overall best preforming method is the SVM with Stochastic Gradient Descent.

A possible reason as to why the SVM stochastic gradient descent preformed better then the scikit-learn is because of the optimization methods used in both. SMO, the optimization method in scikit-learn, most likely couldn't find a more accurate model because it sometimes gets stuck at a singular optima and can't find others when the data is more complex. However, SGD can find other solutions because it randomly selects smaller batches of the data set and is able to ultimately find a better optima. Thus it converges towards a more accurate solution. Even tho SMO methods do tend to converge to the solution faster, SGD is better on larger data sets because it has parallel processing so it is able to find a solution more efficiently. So it is likely that the data set was too large for scikit-learn to efficiently and accurately find a solution.

Another possible explanation for the high accuracy is because there is less overhead, because the model was fit perfectly for the custom build, small changes can be made to help this specific data set to converge towards a more optimal solution.

A possible reason that the stochastic gradient descent was able to find a better solution then the standard batch gradient descent is because SGD's smaller sized random batches can explore more of the data and can find more solutions to the problem, similar to the issues found in SMO. Since batch gradient descent uses the whole data set it is not able to have the benefit of exploring all the aspects of the data, like stochastic gradient descent.

SGD was the best method for this problem, but make note that SGD stereotypical can make data more noisy and can cause issues for inaccuracy, but for the current data and target it is overwhelmingly the best method.

5 Conclusion

This project compared the accuracy and computational speed of two machine learning algorithms to find the best model to predict whether a person would have heart disease or not. After running the data analysis study, it can be concluded that Support Vector Machine with Stochastic Gradient Descent performs the

best to predict the probability of heart disease, because of its accuracy but also because it predicted more false positives than false negatives. And when dealing with health care issues it is better for a person to be predicted with the probability of heart disease and then after getting checked it resulted in a false positive than getting a false negative which can become detrimental to some people's health if it goes unnoticed. Even though, Logistic Regression with Gradient Descent had a significantly faster computational speed, its accuracy did not performed as good as SVM. It should be kept in mind, that if the data set increases in size, it would be a good idea to revisit Logistic Regression to see if its computational speed and accuracy perform better and faster than SVM.

6 References

[1] Bhat, Naresha. Health care: Heart attack possibility. 2020. 10 05 2023.
https://www.kaggle.com/datasets/nareshbhat/health-care-data-set-on-heart-attack-possibility.

[2] Clinic, Mayo. Heart Disease. 25 08 2022. 05 06 2023.
https://www.mayoclinic.org/diseases-conditions/heart-disease/symptoms-causes/syc-20353118.

[3] Prasad, Ashwin. Logistic Regression with Gradient Descent Explained. 14 06 2021. 27 05 2023.
https://medium.com/analytics-vidhya/logistic-regression-with-gradient-descent-explained-machine-learning-a9a12b38d710.

[4] Solutions, Statistics. Assumptions of Logistic Regression. 2023. 27 05 2023.
https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/assumptions-of-logistic-regression/.

7 Appendix/Code

The code was optimized by making the models by scratch and by implementing Gradient Descent and Stochastic Gradient Descent.

7.1 Data Exploration

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report
import time
import os
import warnings
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, roc_curve, roc_auc_score

heart = pd.read_csv('heart.csv')
heart.dtypes

# change categorical/binary variables to objects
heart['sex'] = heart.sex.astype(object)
heart['cp'] = heart.cp.astype(object)
heart['fbs'] = heart.fbs.astype(object)
heart['restecg'] = heart.restecg.astype(object)
heart['exang'] = heart.exang.astype(object)
heart['slope'] = heart.slope.astype(object)
heart['ca'] = heart.ca.astype(object)
heart['thal'] = heart.thal.astype(object)
heart['target'] = heart.target.astype(object)
heart.dtypes
```

```

heart['ca'].unique()
# ca is only from 0-3, so we have to remove rows with 4
heart[heart['ca']==4]
heart.loc[heart['ca']==4,'ca'] = np.NaN
heart['ca'].unique()

# thal is from 1-3, so we have to remove rows with 0
heart['thal'].unique()
heart[heart['thal']==0]
heart.loc[heart['thal']==0,'thal'] = np.NaN
heart['thal'].unique()

# check for missing values
heart.isna().sum() # 7

heart = heart.dropna()
heart.isna().sum()
# drop missing values

heart.describe()
# look at summary stats to help detect outliers

# removing outliers
indexchol = heart[heart['chol']>390].index
heart.drop(indexchol , inplace=True)

indextrest = heart[heart['trestbps']>170].index
heart.drop(indextrest, inplace=True)

indexpeak = heart[heart['oldpeak']>=4].index
heart.drop(indexpeak, inplace=True)

# visualization
heart['age'].hist().plot(kind='bar')
plt.title('Age Distribution')
# mostly 60-70 aged people

# twice as many men as women
heart['sex'].hist().plot(kind='bar')
plt.title('Sex Distribution')
```

7.2 Support Vector Machine Code

```

########################################
# SVM ##################################
########################################
# Splitting the data the same for both methods
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state = 0)
```

```
8
9  ########################################
10 # SVM sklearn ##########################
11 ########################################
12
13 from sklearn.model_selection import cross_val_score
14 from sklearn.pipeline import make_pipeline
15 from sklearn.metrics import accuracy_score
16 from sklearn.preprocessing import StandardScaler
17 from sklearn.svm import SVC
18
19 start_time = time.time()
20 clf = make_pipeline(StandardScaler(), SVC(kernel = 'rbf' ,gamma='auto'))
21 clf.fit(X_train, y_train)
22 #evaluating the problem
23 y_pred = clf.predict(X_test)
24 ac = accuracy_score(y_test, y_pred)
25 # Perform k-fold cross-validation
26 k = 7  # Number of folds
27 scores = cross_val_score(clf, X, y, cv=k)
28 # Calculate and print the mean accuracy score
29 mean_accuracy = scores.mean()
30 print("Mean accuracy:", mean_accuracy)
31
32
33 print("Time: {:.3f} seconds".format(elapsed_time))
34
35 #Visual, Confusion Matrix
36 from sklearn.metrics import ConfusionMatrixDisplay, plot_confusion_matrix
37 plot_confusion_matrix(clf, X_test, y_test, cmap='Blues')
38
39
40 ########################################
41 #SVM w/ GD#############################
42 ########################################
43 start_time = time.time()
44
45 class MyScaler:
46     def __init__(self, with_mean=True, with_std=True):
47         self.with_mean = with_mean
48         self.with_std = with_std
49         self.mean_ = None
50         self.scale_ = None
51
52     def fit(self, X):
53         if self.with_mean:
54             self.mean_ = np.mean(X, axis=0)
55         if self.with_std:
56             self.scale_ = np.std(X, axis=0)
57             self.scale_[self.scale_ == 0] = 1
58
59     def transform(self, X):
60         if self.with_mean:
61             X -= self.mean_
```

```
        if self.with_std:
            X /= self.scale_

        X[np.isnan(X)] = 0
        return X

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)



#Creating the SVM class
class SVM:
    #initating variables
    def __init__(self, learning_rate=0.01, num_iterations=1000, with_mean=True, with_std=True):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.weights = None
        self.bias = None
    #defining a fit and predict method
    def fit_predict(self, X_train, y_train):
        n_samples, n_features = X_train.shape
        w = np.zeros(n_features)
        b = 0
        #hinge loss function
        def hinge_loss(w, b, X, y):
            loss = 1 - y @ (np.dot(X, w) + b)
            loss = np.maximum(0, loss)
            mean_loss = np.mean(loss)
            return mean_loss, -np.dot(y, X.dot(loss > 0)), -np.sum(y * (loss > 0))


        for _ in range(self.num_iterations):
            loss, grad_w, grad_b = hinge_loss(w, b, X_train, y_train)
            w -= self.learning_rate * grad_w
            b -= self.learning_rate * grad_b

        # Define the number of folds
        k = 7

        # Shuffle the training data
        shuffled_indices = np.random.permutation(len(X_train))
        X_shuffled = X_train[shuffled_indices]
        y_shuffled = y_train[shuffled_indices]

        # Calculate the size of each fold
        fold_size = len(X_train) // k

        accuracies = []

        # Iterate over the folds
        for i in range(k):
```

```
            # Split the data into training and validation sets for the current fold
            v_start = i * fold_size
            v_end = (i + 1) * fold_size
            X_v = X_shuffled[v_start:v_end]
            y_v = y_shuffled[v_start:v_end]

            train_indices = np.concatenate((np.arange(v_start), np.arange(v_end, len(X_train))))
            X_train_fold = X_shuffled[train_indices]
            y_train_fold = y_shuffled[train_indices]

            # accuracy for current fold
            y_pred = np.sign(np.dot(X_v, w) + b)
            accuracy = np.sum(y_pred == y_v) / len(y_v)
            accuracies.append(accuracy)

        # average accuracy across all folds
        average_accuracy = np.mean(accuracies)

        return average_accuracy, w, b

#using my custom scaler to scale X
scaler = MyScaler()
X_test = scaler.fit_transform(X_test)
X_train = scaler.fit_transform(X_train)
#applying to data
svm = SVM(learning_rate=0.01, num_iterations=1000, with_mean=True, with_std=True)
average_accuracy,w,b = svm.fit_predict(X_train, y_train)
print(average_accuracy)

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time
print("Time: {:.3f} seconds".format(elapsed_time))

#Visual
from sklearn.metrics import confusion_matrix
import seaborn as sns
cm = confusion_matrix(y_test, y_pred)
# Create a heatmap of the confusion matrix
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()


########################################
#SVM w/ SGD#############################
########################################
start_time = time.time()

class MyScaler:
```

```
    def __init__(self, with_mean=True, with_std=True):
        self.with_mean = with_mean
        self.with_std = with_std
        self.mean_ = None
        self.scale_ = None

    def fit(self, X):
        if self.with_mean:
            self.mean_ = np.mean(X, axis=0)
        if self.with_std:
            self.scale_ = np.std(X, axis=0)
            self.scale_[self.scale_ == 0] = 1

    def transform(self, X):
        if self.with_mean:
            X -= self.mean_
        if self.with_std:
            X /= self.scale_

        X[np.isnan(X)] = 0
        return X

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)




class SVM:
    #intiating the original variables
    def __init__(self, learning_rate=0.01, num_iterations=1000, with_mean=True, with_std=True):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.weights = None
        self.bias = None
    #fit_predict function
    def fit_predict(self, X, y):
        n_samples, n_features = X_train.shape
        w = np.zeros(n_features)
        b = 0
    #hinge_loss function
        def hinge_loss(w, b, X, y):
            loss = 1 - y * (np.dot(X, w) + b)
            loss = np.maximum(0, loss)
            mean_loss = np.mean(loss)
            return mean_loss, -np.dot(y, X.T.dot(loss > 0)), -np.sum(y * (loss > 0))
        num_iterations = 1000
        learning_rate=0.01
        for i in range(num_iterations):
            j = np.random.randint(0, n_samples)
            loss, grad_w, grad_b = hinge_loss(w, b, X_train[j], y_train[j])
            w -= learning_rate * grad_w
```

```
            b -= learning_rate * grad_b
        # Define the number of folds
        k = 7

        # Shuffle the training data
        shuffled_indices = np.random.permutation(len(X_train))
        X_shuffled = X_train[shuffled_indices]
        y_shuffled = y_train[shuffled_indices]


        # Calculate the size of each fold
        fold_size = len(X_train) // k

        accuracies = []

        # Iterate over the folds
        for i in range(k):
            # Split the data into training and validation sets for the current fold
            v_start = i * fold_size
            v_end = (i + 1) * fold_size
            X_v = X_shuffled[v_start:v_end]
            y_v = y_shuffled[v_start:v_end]

            train_indices = np.concatenate((np.arange(v_start), np.arange(v_end, len(X_train))))
            X_train_fold = X_shuffled[train_indices]
            y_train_fold = y_shuffled[train_indices]

            # accuracy for current fold
            y_pred = np.sign(np.dot(X_v, w) + b)
            accuracy = np.sum(y_pred == y_v) / len(y_v)
            accuracies.append(accuracy)

        #average accuracy across all folds
        average_accuracy = np.mean(accuracies)

        return average_accuracy, w , b

#using my custom scaler to scale X
scaler = MyScaler()
X_test = scaler.fit_transform(X_test)
X_train = scaler.fit_transform(X_train)

svm = SVM(learning_rate=0.01, num_iterations=1000, with_mean=True, with_std=True)
average_accuracy,w,b = svm.fit_predict(X_train, y_train)
print(average_accuracy)


y_pred = np.sign(np.dot(X_test, w) + b)


# Stop the timer
end_time = time.time()

```

```
# Calculate the elapsed time
elapsed_time = end_time - start_time

print("Time: {:.3f} seconds".format(elapsed_time))

#Visual, confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns
cm = confusion_matrix(y_test, y_pred)
# Create a heatmap of the confusion matrix
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

7.3 Logistic Regression Code

```

########################################
#Logistic Regression w/ GD###############
########################################

#Set up parameters
X = heart.drop(columns = "target") #drop the column 'target'
y = heart["target"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = 124)

# normalize the independent variables
X_train = (X_train - np.mean(X_train, axis=0)) / np.std(X_train, axis = 0)
X_test = (X_test - np.mean(X_test, axis=0)) / np.std(X_test, axis= 0 )

from sklearn.base import BaseEstimator, ClassifierMixin

class LogisticRegression(BaseEstimator, ClassifierMixin):

        def __init__(self, learning_rate = 0.1, num_iterations = 1000):
                self.learning_rate = learning_rate
                self.num_iterations = num_iterations


        def sigmoid(self, z):
            return 1 / (1 + np.exp(-z))

        def initialize_weights(self, n_features):
              self.weights = np.zeros((n_features,))
              self.bias = 0

        def forward_propagation(self,X):
              linear_output = np.dot(X, self.weights) + self.bias
              y_pred = self.sigmoid(linear_output)
```

```
            return y_pred

        def backward_propagation(self, X, y_pred, y_true):
            n_samples = X.shape[0]
            dw = (1 / n_samples) * np.dot(X.T, (y_pred - y_true))
            db = (1 / n_samples) * np.sum(y_pred - y_true)
            return dw, db

        def update_weights(self, dw, db):
            self.weights = self.weights - self.learning_rate * dw
            self.bias = self.bias - self.learning_rate * db

        def fit(self,X,y):
            n_samples, n_features = X.shape
            self.initialize_weights(n_features)

            for i  in range(self.num_iterations):
                y_pred = self.forward_propagation(X)
                dw, db = self.backward_propagation(X, y_pred, y)
                self.update_weights(dw, db)

            return self

        def predict(self,X):
            y_pred = self.forward_propagation(X)
            y_pred_class = np.where(y_pred > 0.6, 1, 0)
            return y_pred_class

        def predict1(self,X):
            y_pred= self.forward_propagation(X)
            return y_pred

        def predict_proba(self, X):
            y_pred = self.forward_propagation(X)
            return np.column_stack((1 - y_pred, y_pred))

#Testing the model
log_reg = LogisticRegression(learning_rate=0.1, num_iterations=1000)
log_reg.fit(X = X_train, y = y_train)
log_fitted = log_reg.fit(X = X_train, y = y_train)

start_time = time.time()

y_pred = log_reg.predict(X = X_test)
y_pred_prob = log_reg.predict1(X = X_test)
log_odds = np.log(y_pred_prob / (1 - y_pred_prob))

end_time = time.time()

print("Accuracy:" ,metrics.accuracy_score(y_pred, y_test))

execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")
```

```

y_pred_prob
print(classification_report(y_test, y_pred))

#Confusion Matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns
cm = confusion_matrix(y_test, y_pred)
# Create a heatmap of the confusion matrix
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()

# Logistic Regression
import time
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, roc_curve, roc_auc_score

# Load the dataset
data = pd.read_csv("heart.csv")

# Select the relevant variables for analysis
selected_vars = data.columns.tolist()
data = data[selected_vars]

# Split the data into features and target variable
X = data.drop('target', axis=1)
y = data['target']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create an instance of the LogisticRegression model
model = LogisticRegression()

# Start the timer
start_time = time.time()

# Fit the model to the training data
model.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = model.predict(X_test)
y_pred_prob = model.predict_proba(X_test)[:, 1]

# Stop the timer
end_time = time.time()

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
```

```
print(f"Accuracy: {accuracy}")

# Calculate the time taken
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")



from sklearn.metrics import accuracy_score, confusion_matrix

# Create the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Display the confusion matrix
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
plt.xticks([0, 1], ['Predicted 0', 'Predicted 1'])
plt.yticks([0, 1], ['True 0', 'True 1'])
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.show()

# Gradient Descent

from sklearn.metrics import accuracy_score, confusion_matrix

# Create the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Display the confusion matrix
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
plt.xticks([0, 1], ['Predicted 0', 'Predicted 1'])
plt.yticks([0, 1], ['True 0', 'True 1'])
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.show()
```