Heart Disease Prediction Empowered with Optimizing Machine Learning Algorithm

Truc Le, Jinghong Chen, Selamawit Berekat, Yuxi Jiang

17 March 2022

1 Abstract

A crucial part of leading a healthy life is to have a proper functioning heart. Cardiovascular diseases are conditions that affect the functions and structure of the heart of affected individuals. The disorder of blood flow in the heart can cause a chronic disease called cardiovascular disease. According to the Centers for Disease Control and Prevention, in 2019 there were 360,900 people who died from coronary artery disease and every year about 805,000people experienced a heart attack [7]. This report proposes the use of several different machine learning algorithms as an effective way to detect heart disease at an early stage. The dataset studied in this project is retrieved from University of California Irvine (UCI) Machine Repository Cleveland Records[6]. Different machine learning algorithms such as: Naive Bayes (NB), Support Vector Machine (SVM), K-nearest neighbor (KNN), Neural Network (NN), Logistic Regression is used to predict whether a patient has heart disease given their features. In addition to that, Gradient Descent algorithm is applied to Neural Network and Logistic regression algorithms to optimize the computational efficiency of the machine learning algorithms. The best machine learning algorithm is picked by comparing the best computation time and accuracy scores among the five prediction algorithms. It is determined that the machine learning algorithms optimized by the Gradient Descent algorithm, Neural Network and Logistic Regression, resulted in the two highest accuracy scores when predicting heart disease in the testing dataset. Therefore, it is proposed that the Gradient Descent optimized machine learning algorithms be used in detecting cardiovascular disease at an early stage.

Keywords
Classification Logistic Regression Model · Support Vector Machine (SVM) · Naive Bayes (NB)· K-nearest neighbor (KNN) · Neural Network (NN) · Gradient Descent Optimization model

2 Introduction

Heart diseases, also known as cardiovascular diseases, are one of the leading deadly and chronic diseases known in the United States and the entire world.[14]. The primary factors to consider that can cause heart disorder are poor eating habits, physical inactivity, smoking and genetics. However, organizations like the American Heart Association have developed several preventative strategies that individuals can use to lower their risk of cardiovascular disease. However, the American Heart Association estimated only a distinctive minority, about 5 percent, follow the recommended strategy for achieving the "ideal" cardiovascular health. [18]. With this pandemic situation happening around the world, the number of deaths due to heart disease is increasing.[21]. Misdiagnosis is a leading concern for patients at risk of heart disease, as it prevents physicians from detecting cardiovascular diseases in patients at the early stage.[10].

The objective of this research is to use machine learning algorithms to improve the predictive accuracy of cardiovascular disease in patients. In addition, the best machine learning algorithm will be chosen at the end based on its accuracy score and computational run time. The heart disease dataset used in this project is a processed version of the Cleveland Heart Disease dataset from the UCI Machine Learning Repository, and was taken from the data science platform, Kaggle.[6]. The dataset contains 303 observations with 13 independent variables, characteristic attributes associated with individuals in the study, with the goal of determining the presence of heart disease in patients. The dependent variable is target and contains a discrete value of 0 for patients not suffering, and values from 1, 2, 3, 4 to indicate presence of the disease. The independent variables use to predict heart disease are: age, sex, chest pain type, resting blood pressure,

serum cholesterol, fasting blood sugar, resting electrocardiographic results, maximum heart rate achieved (thalach), exercise-induced angina (exang), ST depression induced by exercise relative to rest (old peak), the slope of the peak exercise ST segment (slope), number of major vessels (0–3) colored by fluoroscope (ca), and Thallium (thal).[6]. Table 1 below contains the information related to the variables used from the dataset.

Table 1: Heart Disease Dataset Attributes

Attributes	Description	Symbol	Data Type
1. Age	In years	age	Integer
2. Sex	Male = 1, Female = 0	sex	Categorical
3. Chest Pain Type	Values: 0, 1, 2, 3	cp	Categorical
4. Resting Blood Pressure	mmHg	trestpbs	Integer
5. Serum Cholesterol	mg/dl	chol	Integer
6. Fasting Blood Sugar	(>120mg/dl), (true = 1, false = 0)	fbs	Categorical
7. Resting ECG	(mg/dl) (Value: 0, 1, 2)	restecg	Integer
8. Maximum Heart Rate achieved		thalach	Integer
9. Exercised induced angina	yes = 1, no = 0	exang	Categorical
10. ST depression induced by exercise relative to rest		oldpeak	Float
11. The slope of the peak exercise ST segment	Values: 0, 1, 2	slope	Categorical
12. Number of major vessels colored by fluoroscope	Values: 0, 1, 2, 3	ca	Categorical
13. Thallium	Values: 0, 1, 2, and the label	thal	Categorical
14. Target	$yes_{disease} = 1, no_{disease} = 0$	target	Categorical

During the stage of data preprocessing, several observations were deleted due to them either being duplicates or outliers. Outliers were determined by testing the value against the z-score, any observation with a z-score greater than the 3.5 are dropped. This resulted in the dataset being truncated down to 297 observations. Column types of factor level were converted to numeric and the continuous variables were normalized. Following replacing the four discrete values of response variable to be of a binary classification form (1 = yes disease) and (0 = no disease).

3 Proposed Methods

There are two basic approaches in machine learning, supervised and unsupervised learning. The main difference between the two machine learning approaches is that the former uses labeled data sets to train the algorithms to accurately classify data or predict outcomes, while the latter analyzes unlabelled datasets[13]. This report focuses on five supervised machine learning algorithms: Logistic Regression, Neural Network, Support Vector Machine (SVM), Naïve Bayes, and k-nearest neighbor (K-NN), and observes each classifier's performance at classifying the heart disease dataset. This section aims at providing a general idea of how each of the five supervised learning algorithms works and their parameters. The heart disease dataset is divided using the 80/20 ratio; 80% of the original dataset is used to train the algorithms, and 20% of the data is reserved as training samples to evaluate the performance of the classifiers. The mathematical proof of each supervised machine learning algorithm can be found in the appendix.

3.1 Logistic Regression

Binary logistic regression (LR) is a classification method used to analyze the relationship between a dichotomous target variable and the predictor variables.[3]. For this project, the target variable is the presence of heart disease, and the patient's features are the predictor variables. Before a logistic model can be fitted, it is necessary to test whether the logistic regression assumptions will hold for the experimental data. The

four main logistic regression assumptions are: independence of error, linearity between predictor variables and the logit response variable, absence of collinearity, and lack of strongly influential outliers.[20]. The lack of influential outliers assumption holds true since all outliers within the data have been removed in the data pre-processing step. Based on the results from the distribution plot, QQ-plot, and the correlation matrix, the assumptions of independence of error, linearity between predictor and response variables, and absence of multicollinearity holds true.

The logistic model works by first computing the weighted sum of the predictor variables and then estimating the probability of a given observation belonging to a specific classification. In addition, the classifier assumes that the response variable is conditionally Bernoulli distributed given the inputted values of the predictor variables. To achieve this, a sigmoid function, taking the model parameters (theta) as inputs, maps any real values into a constraint interval of 0 and 1 [20]. For this report, the value outputted by the sigmoid function is interpreted as the probability of heart disease being present in an observed patient. The default threshold value of 0.5 is chosen as the decision boundary for the logistic model. Hence, any observations with a logit value of 0.5 or greater are labeled 1, heart disease present, otherwise 0, no heart disease. A cost function is used to evaluate how well the regression model did at predicting the correct outcomes compared to the actual outcomes. The objective is to minimize the cost function in order to maximize the model prediction accuracy.

Gradient descent, an iterative optimization algorithm, optimizes the logistic regression model by finding the optimal values for theta that minimize the loss function. [5]. The optimization algorithm updates the parameters' values at each iteration until it converges to the local minima of the cost function; it does this by moving in the opposite direction of the gradient at a constant step-size. [12]. The algorithm converges to the local minima when the difference between the updated and old parameters' values is less than the tolerance value of 0.001. The max tolerance condition terminates the function once convergence happens, preventing the algorithms from having to complete the specified max iterations; this early termination minimizes the computational time.

3.2 Neural Network using Gradient Descent

Neural Networks is the core of deep learning algorithms, and it is a part of machine learning. They can mimic how human brain neurons signal with each other. They are a form of layers with nodes, and each node contains an input layer, multiple hidden layers, and an output layer. A threshold value is defined at first and if the output of each node is above the threshold value, that makes the node activated and continues to the next layer. And if not activated, there is no data that transfers along to the following layers of the network. This method works assuming each node has one input layer, threshold, weights, and one output layer. They can be viewed as having their own logistic regression model. Every time a node from an input is determined, then they will be assigned its weight. Input layer in this case is the layer consisting of the number of columns in the X matrix. Output layer that is of vector form consists of one column that represents the response variable. Weight is the measurement assigned to determine the importance of the variable. The output is passed through an activation or the sigmoid function once the inputs are multiplied by their weights and summed. Here, the sigmoid function computes the error representation of the model using a negative log-likelihood to solve for the coefficient estimators. Training the model using the neural network with input layer and output layer produces the output of the coefficient estimators. This process of actively transferring data from layer to layer is the main process in neural networks.

3.3 Support Vector Machine

This is a linear model used for both regression and classification-based problems. It is known for its ability to produce significant accuracy for linear or non-linear models with less computation running time. Main structure of this model is to use a hyperplane to separate the input data based on the target class label 1 also identified as suffering from heart disease, and target class label -1 identified as not suffering from heart disease. This establishes a margin level for the SVM model to be between -1 and 1. The process starts

with finding the distance difference between the data points and the classified labels for a given weight and a constant bias to select the maximum boundary margin. The hinge loss function is used to determine the maximized margin of decision boundary. If the data point belongs to a certain class and that class is a positive sign, then the cost function of the error becomes zero since the evaluated data is greater than or equal to one. On the other hand, if the data point belongs to a certain class label where it is indicated by a negative sign, then the hinge function calculates the distance difference and returns the loss/error value. A regularization parameter represented by the Lambda symbol is added to the function to balance between the decision boundary and the loss values.[8]. And the weights of the data are updated simultaneously using the gradients that take partial derivatives of loss function with respect to the weights. This process of separating the data based on the class label and using a hyperplane to maximize the boundary margin between the classes and minimize the loss using gradient descent is the support vector machine algorithm applied in this experiment.[4].

3.4 Naive Bayes

This method is one of the classification algorithms for supervised machine learning. It uses the Bayes theorem formula to find the probability of the classified labels multiplying by its joint conditional for the feature given the classified labels. The theorem provides a way of computing the posterior probability $P(A|B)$ from the class prior probability P(A) and the likelihood P(B—A). The mathematical formula is represented as follows $P(A|B) = \frac{(P(B|A)P(A))}{P(B)}$". [9]
The word naive is derived from the fact that the joint conditional distribution of the features is conditionally independent. This assumption in the heart disease prediction can be translated as each data point for the features being independent from each other given a specified label. For example, if a data point is classified as suffering from a heart disease label, there won't be information about the probability of other values from the other features.

3.5 K-NN with cross validation

The learning process in machine learning can be simplified by assumptions and can limit what can be learned. In a parametric model, the model summarizes the data using a fixed size of parameter. In this experiment, the logistic model first picks a value for k then predicts based on k observation in the training data that are close to the output of the unknown variable. In another way the idea is the classifier takes each data from the testing dataset to compare it with all the data in the training dataset then predicts the label of the output based on the nearest label on training data by computing the distance difference. It continues accordingly while evaluating how well the prediction did. This step of evaluating is the k-fold cross-validation where the training dataset is divided into few parts and first trains the model on some of the training data and tests on the remaining part of the testing dataset. For this experiment, the process iterates over several possible k values to determine which K is a better value that predicts and runs efficiently.

4 Simulated Study

The five classifiers were tested on two simulated data sets to observe whether they could perform as expected. The two datasets are modeled after the heart disease dataset used in this report, each dataset containing 13 predictor variables and one binary response variable. However, the two datasets differ in size. The first dataset, the "small data," only contains 300 observations, while the second dataset, the "big data," contains 10000 observations. The purpose of the two datasets is to observe the impact of the sample size on the classifiers' performance, in particular, the prediction accuracy and computation time. Each classifier was run 100 times on both simulated datasets to get the average prediction accuracy, the variance of accuracy, and the average run time. The results from the 100 runs confirm that all five classifiers performed as expected.

It is worth noting that the average prediction accuracy for all classifiers decreases as the dataset size increases. In the small dataset, the classifiers' average accuracy percentages were in the high 80s and lower 90s. However, the average accuracy percentages drastically decreased to the high 70s and lower 80s with the big dataset, about a 10 percent decrease in the prediction accuracy of each classifier. The increase in training sample size could result in over fitting due to the added noise in the training sample set, making it difficult for the classification models to accurately predict the specific class the testing samples belong to, resulting in lower prediction accuracy's.

In addition, the computation times observed from the 100 runs confirmed that the classifiers optimized by the gradient descent algorithm could conserve computation time when increasing the size of the training dataset. All five classifiers saw an increase in the computation time when testing them on the big dataset, which makes sense since there is an increase in the volume of data being analyzed by the machine learning algorithms. However, the increase in computational time was not proportional between the five classifiers.

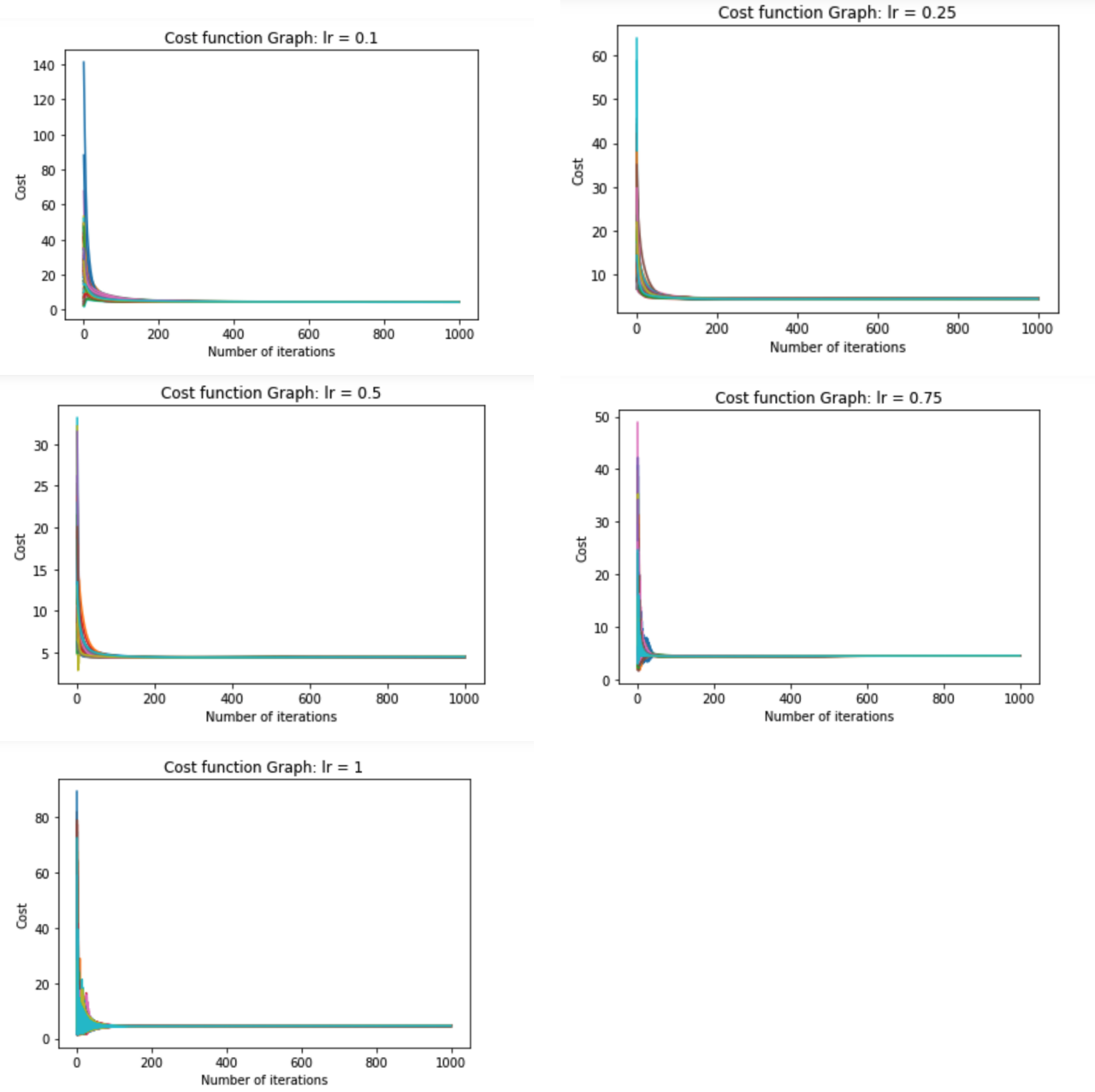
Table 2: Performance of each algorithm running 100 times with simulation data: small data n = 300 and big data n = 10,000

Algorithm	Average Accuracy	Variance of Accuracy	Run Time (seconds)
1. KNN k=10, small data	0.8925	0.0013	11.4683
2. KNN k=10, big data	0.8387	0.0016	600.2637
3. Neural network, small data $\alpha : 0.01, e = 5e-4$	0.9237	0.0011	2.4251
4. Neural network, big data $\alpha : 0.01, e = 5e-4$	0.8171	0.0019	32.4583
5. SVM, small data	0.9138	0.0011	105.4096
6. SVM, big data	0.8086	0.0013	518.9568
7. Naive Bayes, small data	0.9247	0.001	0.2623
8. Naive Bayes, big data	0.785	0.0023	0.4597
9. Logistic regression, small data $\alpha : 0.1, e = 1e-3$	0.9157	0.0009	1.4554
10. Logistic regression, big data $\alpha : 0.1, e = 1e-3$	0.8197	0.0016	17.4149

The naive Bayes algorithm has the best computation efficiency, only seeing an increase from 0.26 seconds to 0.46 seconds when testing the algorithm on the 10,000 observations dataset. The computation efficiency of the naive Bayes algorithm can be attributed to its fast convergence rate. Although a very fast algorithm, Naive Bayes also suffers from low prediction accuracy when the assumption that all features are independent of each other is violated. Conversely, the K-nearest neighbor (KNN) and Support Vector Machine (SVM) algorithms, the two non-optimized classification algorithms in this project, have the worst computation efficiency, taking around 519 and 600 seconds, respectively, to finish computing the dataset. The classifiers optimized by the gradient descent algorithm, the neural network and logistic regression methods, do not experience a significant increase in their computation times when tested on the big dataset. This conserved computation efficiency is attributed to the gradient descent algorithm stopping the iterations process once the local minima is found, allowing for both optimized algorithms to have better computation time. To evaluate the performance of the gradient descent optimized logistic regression model, the cost function plots for the step-sizes of 0.1, 0.25, 0.5, 0.75, 1, were produced as shown below. The plot shows that the gradient descent algorithm was able to reach local minima before max iterations were achieved.

From the simulated data, it can be concluded that the classifiers perform better when dealing with the smaller dataset that contains 300 observations. Therefore, it is expected for the five classification algorithms to perform well on the real dataset since the simulated small dataset was modeled after the heart disease dataset being used in this project. In addition, given the results from the simulation data study, it is expected that the Naive Bayes algorithm to be the best supervised machine learning algorithm in terms of classification accuracy and computation time to classify the actual dataset.

Figure 1: Cost Function of Learning Rate(s) for Simulated Data



5 Real Data Study

The analysis on the actual dataset was performed by first understanding the data for the logistic regression model. The aim of exploring the dataset helps to check that the models' assumptions have not been violated. As illustrated in the table below, the first five observation tables present the variables included for analysis that are mainly 13 independent predictors and 1 response variable. Which follows with a descriptive statistics table that summarizes the given data. For instance, in this experiment the age of patients is between minimum 29 years old up to maximum 77 years old.

Table 3: First Five Observation of Data

age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0. 52	1	0	125	212	0	1	168	0	1.0	2	2	3	0
1. 53	1	0	140	203	1	0	155	1	3.1	0	0	3	0
2. 70	1	0	145	174	0	1	125	1	2.6	0	0	3	0
3. 61	1	0	148	203	0	1	161	0	0.0	2	1	3	0
4. 62	0	0	138	294	1	1	106	0	1.9	1	3	2	0

Table 4: Summary Statistics of Variables

age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
count 1025.00	1025.00	1025.00	1025.00	1025.00	1025.00	1025.00	1025.00	1025.00	1025.00	1025.00	1025.00	1025.00	1025.00
mean 54.43	0.70	0.94	131.61	246.00	0.15	0.53	149.11	0.34	1.07	1.39	0.75	2.32	0.51
std 9.07	0.46	1.03	17.52	51.59	0.36	0.53	23.01	0.47	1.18	0.62	1.03	0.62	0.50
min 29.00	0.00	0.00	94.00	126.00	0.00	0.00	71.00	0.00	0.00	0.00	0.00	0.00	0.00
25% 48.00	0.00	0.00	120.00	211.00	0.00	0.00	132.00	0.00	0.00	1.00	0.00	2.00	0.00
50% 56.00	1.00	1.00	130.00	240.00	0.00	1.00	152.00	0.00	0.80	1.00	0.00	2.00	1.00
75% 61.00	1.00	2.00	140.00	275.00	0.00	1.00	166.00	1.00	1.80	2.00	1.00	3.00	1.00
max 77.00	1.00	3.00	200.00	564.00	1.00	2.00	202.00	1.00	6.20	2.00	4.00	3.00	1.00

Skewness of the data among variables can be determined by comparing the mean and medians of each variable. Since the mean can be sensitive to outliers and provide a wrong conclusion, the duplicated values and outliers have been removed resulting in a decrease in the number of observations. To ensure the independence and association between variables, the correlation matrix displays how some variables have strong positive correlation, moderate positive correlation and with some having a negative or zero correlation. For instance, there is a 0.55 strong correlation between the variable old peak and slope. Correlation plot available in the last page.

The assumption of normality for the distribution of data is verified using a histogram plot where it separates the categorical variables for a barplot, and the numerical variables for a histogram. There are two distribution plots in this analysis, however the key takeaway is some of the variables are highly skewed indicating the direction of the outliers. In addition the assumption of distribution for the residuals is verified with the normal QQ plot, as it illustrates how far the distribution of both the variables deviates from the fitted line. For example, the variable oldpeak deviates from the fitted line largely on the left tail. Distribution plots available in the last page.

The following stage of analysis after understanding the data becomes the pre-processing where the dataset is randomly split into training and testing to evaluate the performance of the logistic model. In this experiment, 80% of the data is assigned for training the model whereas the 20% of it is assigned for testing the model. The five implemented algorithms are to be tested on the actual training and testing data for the binary classification problem. For each implemented algorithm as explained in the above sections, the running time, average and variance of the accuracy scores were compared.

The results obtained for each algorithm are as displayed below, and compared to the simulated data most of the classifiers have slightly lower accuracy scores and increased the required running time as well.

Table 5: Performance for each algorithm running 100 times:

Algorithm	Average accuracy	Variance accuracy's	Run time (seconds)
1. KNN k=9	0.803	0.002	11.3156
2. Neural network: $\alpha : 0.01, e = 5e-4$	0.8328	0.002	5.7612
3. SVM	0.8288	0.0022	117.0620
4. Naive Bayes	0.8167	0.0021	0.3143
5. Logistic regression: $\alpha : 0.1, e = 1e-3$	0.834	0.0017	4.1023
Sklearn Logistic regression: iterations: 10,0000	0.8313	0.0048	0.8859
Best model accuracy:	Neural Network optimized by Gradient Descent.	Max average accuracy:	83.62%
Best model by time:	Naive Bayes.	Fast time:	0.3143 seconds

6 Conclusion

This report compared five supervised machine learning algorithms to find the best model to predict the presence of cardiovascular disease in patients. From the experimental results, it is concluded that the generative probabilistic and gradient descent optimized algorithms are the optimal models at predicting cardiovascular disease in patients. The Naive Bayes classifier is chosen since it has the fastest computation time among the five classification algorithms, and its average accuracy percentage is relatively close to the other classifiers' average accuracy percentages. The optimized logistic regression and neural network classifiers are chosen for their high average accuracy scores and computation time. In addition, it should be noted that the optimized classifiers are able to achieve roughly the same average prediction accuracy as the SKLearn logistic regression function. The report shows that machine learning techniques can be implemented to help detect heart disease in patients given their medical records.

It should be noted that the experimental results from this study are obtained from a limited population size, therefore, the results are only applicable to this restricted population. In addition, a larger dataset would be useful to increase the prediction accuracy of the classifiers, since there would be more samples to train the classifiers with.

7 Appendix

7.1 Code

7.1.1 Explanation of Optimization

For neural network, we start with random weights and biases. We use backpropagation to train our neural network. Backpropagation computes the gradient of the loss function with respect to the network's weights. Gradient descent is used to update our weights and biases and find the minimum of the cost function. Gradient descent is a popular iterative algorithm to find the local minimum. It works by moving the same little step each time toward the minimum. When updating our weights, W new = W old – (* dE/dW), is the step size we set, and dE/dW is the partial derivative of the error for each Xs calculated by backpropagation. We update weights and biases step by step until we minimize the error. However, we do not know when the algorithm will reach the minimum since the starting point is random, so we usually set a big number to iterate that is more than enough to find the minimum. After finding the minimum, the algorithm continues and will wiggle around the minimum. We can find a way to stop the algorithm earlier to be more efficient. We set a small tolerance value for e. When norm(new result - old result)/norm(new result) is less than e, the

new result is not much different from the old result, and the algorithm reaches the minimum. This method allows us to stop early, resulting in computation efficiency.

The logistic regression function starts by generating a random theta vector with 14 entries, this vector will be use by the gradient descent algorithm as a starting point. For each iteration, the gradient descent will update the parameter values using the previous iteration parameter values. The independent variables are then multiplied with its corresponding paramater values and summed up, this value is then passed to the sigmoid function to return a value between 0 and 1. The cost value is then calculated by taking the difference between the expected and actual response value. At the end of each iteration, the difference between the updated theta vector and the old theta vector is compared to the max tolerance value of 0.001. This process repeats itself for each iteration until the difference between the two theta vectors is less than 0.001, in that case, the algorithm has converged to the local minimum of the cost function. Once convergence happens, gradient descent stops iterating over the function and returns the updated theta vectors containing the optimal parameter values. Using the gradient descent algorithm with a stop condition allows for computation efficiency, since the algorithm would only run for as long as it needs to converge.

7.1.2 Code

(1) Import Python Libraries

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats
import statsmodels.api as sm
from sklearn.datasets import make_classification
from sklearn.model_selection import KFold
from scipy.stats import mode
from sklearn.model_selection import train_test_split
import math
import sklearn
from sklearn.linear_model import LogisticRegression
import scipy
import scipy.linalg
from scipy.linalg import *
from sympy import *
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix, f1_score
import warnings
warnings.filterwarnings("ignore")
import time
```

(2) Data Exploratory

```
data = pd.read_csv('heart.csv')
data.head(5)
data.describe().round(2)

# No missing values
print(data.isnull().values.any())

# row 164 is duplicated and drop it
print(data.duplicated()[data.duplicated()].index)
data = data.drop_duplicates()
```

```
# clean outliers with z-score > 3.5
data_clean = data[(np.abs(stats.zscore(data)) < 3.5).all(axis=1)]
#delete 6 outliers
print(data.shape)
print(data_clean.shape)
corr_matrix = data_clean.corr().round(4)
upper = corr_matrix.abs().where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))

# no parameters are strongly correlate
upper
print((upper>0.8).values.sum())
plt.figure(figsize=(20,10))
sns.heatmap(corr_matrix,cmap="BrBG",annot=True)

# distribution of data
fig, axs = plt.subplots(4, 4, figsize=(10,10))
count = 0
for i in range(4):
    for j in range(4):
        sns.histplot(data=data_clean, x=data_clean.columns[count], kde=True,ax=axs[i,j])
        if count >= 13:
            break
        count += 1
plt.suptitle('Distribution of all variables')
plt.show()

Y = data_clean.iloc[: , -1:]
X = data_clean.iloc[:,:-1]

#normalize
X_numeral =  X.iloc[:,[0,3,4,7,9]]
X_normal = X.copy()
for i in X_numeral.columns:
    X_normal[i] = (X_numeral[i]-X_numeral[i].mean())/np.sqrt(X_numeral[i].var())

# distribution after normalze
fig, axs = plt.subplots(2, 3, figsize=(10,10))
sns.histplot(data=X_normal, x="age", kde=True,ax=axs[0, 0])
sns.histplot(data=X_normal, x="trestbps", kde=True,ax=axs[0, 1])
sns.histplot(data=X_normal, x="chol", kde=True,ax=axs[0, 2])
sns.histplot(data=X_normal, x="thalach", kde=True,ax=axs[1, 0])
sns.histplot(data=X_normal, x="oldpeak", kde=True,ax=axs[1, 1])
plt.suptitle('Distribution of all numeral variables after normalize')
plt.show()

# QQ plot
numeral = [0,3,4,7,9]
X_numeral = X.iloc[: , numeral]
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(20,10))
ax= axes.flatten()
for i in range(5):
    sm.qqplot((X_numeral.iloc[:,i]), fit = True, line='q', ax=ax[i])
    ax[i].title.set_text(X_numeral.columns[i])
```

```
plt.suptitle('QQ plot for all numeral variables')
plt.show()

# cleaned data
data = pd.read_csv("archive\heart.csv")
data = data.drop_duplicates()

data = np.array(data)
data_clean = data[(np.abs(stats.zscore(data)) < 3.5).all(axis=1)]

# normalize data
num_col = [0,3,4,7,9]
for i in num_col:
    data_clean[:,i] = (data_clean[:,i]-data_clean[:,i].mean())
                      /np.sqrt(data_clean[:,i].var())

# simulated small and large data
# small data
X_sim, Y_sim = make_classification(n_features=13,n_samples=300,random_state=0)
# big data
X_sim1, Y_sim1 = make_classification(n_features=13,n_samples=10000,random_state=0)
```

(3) K-nearest neighbors algorithm

```
"""
*    Title: K-Nearest Neighbors from Scratch with Python
*    Publisher: AskPython
*    Date: 2022
*    Code version: Original
*    Availability: https://www.askpython.com/python/examples/k-nearest-neighbors
                   -from-scratch
"""
# The following code will build a KNN classifier. We will use euclidean distance
# to calculate the kth closest training observations to the testing observation.
# We will choose the mode class of the kth nearest observation as the protection
# of the training observation.

def eucledian_dist(p1,p2):
    dist = np.sqrt(np.sum((p1-p2)**2))
    return dist

def KNN_prediction(x_train, y , x_test, k):
    predictions = []

    for item in x_test:
        point_dist = []

        for j in range(len(x_train)):
            distances = eucledian_dist(np.array(x_train[j,:]) , item)
            point_dist.append(distances)
        point_dist = np.array(point_dist)

        dist = np.argsort(point_dist)[:k]
        labels = y[dist]
```

```
        lab = mode(labels)
        lab = lab.mode[0]
        predictions.append(lab)

    return predictions

# knn model testing with simulate data
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim=
    train_test_split(X_sim, Y_sim, test_size=0.2, random_state=i)
    Y_pred_sim = KNN_prediction(X_train_sim, Y_train_sim, X_test_sim , 10)
    accuracy_list.append(sum(Y_test_sim == Y_pred_sim)/X_test_sim.shape[0])
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim=
    train_test_split(X_sim1, Y_sim1, test_size=0.007, random_state=i)
    Y_pred_sim = KNN_prediction(X_train_sim, Y_train_sim, X_test_sim , 10)
    accuracy_list.append(sum(Y_test_sim == Y_pred_sim)/X_test_sim.shape[0])
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

# model selection for best number of k by k-flod cross validation
# The following code use k fold cross-validation k = 20 to find the
# best k for the KNN method. We will find the best model considering
# the highest accuracy and f score.

X = data_clean[:,:13]
Y = np.matrix(data_clean[:,13]).T

kf = KFold(n_splits = 20, shuffle = True, random_state = 0)
k_flod_lst = list(kf.split(data_clean))

max_accuracy = 0
max_accuracy_n = 0
max_f = 0
max_f_n = 0
for i in range(10):
    accuracy_list = []
    f_list = []
    for j in range(20):
        X_train = X[k_flod_lst[j][0],:]
        Y_train = Y[k_flod_lst[j][0],:]
        X_test = X[k_flod_lst[j][1],:]
        Y_test = Y[k_flod_lst[j][1],:]
```

```
        Y_pred = KNN_prediction(X_train, Y_train, X_test , i+1)
        accuracy_list.append(sum(Y_test == Y_pred)/X_test.shape[0])
        f_list.append(f1_score(Y_test, Y_pred))

    mean_accuracy = np.mean(accuracy_list)
    if mean_accuracy > max_accuracy:
        max_accuracy = mean_accuracy
        max_accuracy_n = i+1

    mean_f = np.mean(f_list)
    if mean_f > max_f:
        max_f = mean_f
        max_f_n = i+1

#print(str(i+1) + 'nearest neighbor accuracy: ' + str(mean_accuracy) +
'F score: ' + str(mean_f))
print("Best n by accuracy is " + str(max_accuracy_n) + ', max accuracy: ' +
str((max_accuracy*100).round(2)) + "%")
print("Best n by f score is " + str(max_f_n) + ', max f score: ' +
str(max_f.round(4)))

# KNN modeling performance
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train, X_test, Y_train, Y_test =
    train_test_split(X, Y, test_size=0.2, random_state=i)
    Y_pred = KNN_prediction(X_train,Y_train,X_test , 8)
    accuracy_list.append(sum(Y_test == Y_pred)/X_test.shape[0])
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))
```

[2]

(4) Neural network algorithm

```
"""
*     Title: Simple MNIST NN from scratch (numpy, no TF/Keras)
*     Author: Samson Zhang
*     Date: 2021
*     Code version: 1.0
*     Availability: https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-
                from-scratch-numpy-no-tf-keras/notebook
"""
# The following code will build a neural network model with on hidden layer.
# First, randomly generate the correct number of weights and biases
# corresponding number of hidden layer nodes we choose. Then, we use forward
# propagation and backward propagation to find the gradient of the loss
# function with respect to the weights and biases. Next, we use the gradient
# descent method to iterate and update the weights and biases, to find the
# minimum of the loss function. We choose ReLU as the activation function
# for the hidden layer and softmax as the activation function for the output layer.
```

```
def start_weights_biases(features,classes,node):
    np.random.seed(0)
    W1 = np.random.rand(node, features) - 0.5
    b1 = np.random.rand(node, 1) - 0.5
    W2 = np.random.rand(classes, node) - 0.5
    b2 = np.random.rand(classes, 1) - 0.5
    return W1, b1, W2, b2

def ReLU(Z):
    return np.maximum(Z, 0)

def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A

def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

def ReLU_deriv(Z):
    return Z > 0

def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, 2))
    one_hot_Y[np.arange(Y.size), Y.astype(int)] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y

def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    m = Y.size

    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2

def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2

def neural_networks_prediction(X_train,Y,X_test,node,alpha,iterations=10000,e=5e-4):
    W1, b1, W2, b2 = gradient_descent(X_train,Y,node,alpha)
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X_test)
```

```
    return np.argmax(A2, 0)

def gradient_descent(X, Y, node, alpha, iterations=10000, e = 5e-4):
    W1, b1, W2, b2 = start_weights_biases(13,2,node)
    Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
    old = A2[0]

    for i in range(iterations):
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)

        if np.linalg.norm(A2[0] - old) / np.linalg.norm(A2[0]) < e:
            break
        old = A2[0]

    return W1, b1, W2, b2

# neural network model testing with simulate data
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim =
    train_test_split(X_sim, Y_sim, test_size=0.2, random_state=i)
    X_train_sim = X_train_sim.T
    X_test_sim = X_test_sim.T
    Y_pred_sim = neural_networks_prediction(X_train_sim,Y_train_sim,X_test_sim,10,0.1)
    accuracy_list.append(sum(Y_test_sim == Y_pred_sim)/X_test_sim.shape[1])

print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim =
    train_test_split(X_sim1, Y_sim1, test_size=0.007, random_state=i)
    X_train_sim = X_train_sim.T
    X_test_sim = X_test_sim.T
    Y_pred_sim = neural_networks_prediction(X_train_sim,Y_train_sim,X_test_sim,10,0.1)
    accuracy_list.append(sum(Y_test_sim == Y_pred_sim)/X_test_sim.shape[1])
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

# model selection for best number of nodes by k-flod cross validation
# The following code use k fold cross-validation k = 20 to find the best
# number of nodes in the hidden layer for the method of the neural network.
# We will find the best model considering the highest accuracy and f score.

X = data_clean[:,:13].T
Y = data_clean[:,13]
```

```
kf = KFold(n_splits = 20, shuffle = True,random_state = 0)
k_flod_lst = list(kf.split(data_clean))
max_accuracy = 0
max_accuracy_n = 0
max_f = 0
max_f_n = 0
for i in range(10,26):
    accuracy_list = []
    f_list = []
    for j in range(20):
        X_train = X[:,k_flod_lst[j][0]]
        Y_train = Y[k_flod_lst[j][0]]
        X_test = X[:,k_flod_lst[j][1]]
        Y_test = Y[k_flod_lst[j][1]]

        Y_pred = neural_networks_prediction(X_train, Y_train, X_test, i, 0.01)
        accuracy_list.append(sum(Y_test == Y_pred)/X_test.shape[1])
        f_list.append(f1_score(Y_test, Y_pred))

    mean_accuracy = np.mean(accuracy_list)
    if mean_accuracy > max_accuracy:
        max_accuracy = mean_accuracy
        max_accuracy_n = i

    mean_f = np.mean(f_list)
    if mean_f > max_f:
        max_f = mean_f
        max_f_n = i

#print(str(i) + " hidden layer nodes accuracy: " + str(mean_accuracy) +
'F score: ' + str(mean_f))
print("Best nodes by accuaracy is " + str(max_accuracy_n) + ', max accuracy: '
+ str((max_accuracy*100).round(2)) + "%")
print("Best nodes by f score is " + str(max_f_n) + ', max f score: '
+ str(max_f.round(4)))

# Neural network modeling performance
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train, X_test, Y_train, Y_test =
    train_test_split(X.T, Y, test_size=0.2, random_state=i)
    X_train = X_train.T
    X_test = X_test.T
    Y_pred = neural_networks_prediction(X_train, Y_train, X_test, 12, 0.01)
    accuracy_list.append(sum(Y_test == Y_pred)/X_test.shape[1])
print("Neural network modeling performance")
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))
```

[23]

(5) Support Vector Machine algorithm

```
"""
*     Title: ML from Scratch
*     Author: Sunrit Jana
*     Date: Jun 12, 2021
*     Code version: 1.0
*     Availability: https://github.com/python-engineer/MLfromscratch/
        blob/master/mlfromscratch/svm.py
"""

def svm_fit(X, y, l_r = 1e-3, Lambda = 1e-2, num_iter = 1000):
    """
    w*x-b     1 if yi = 1
    w*x-b < 1 otherwise
    keep updating w and b based on condition yi*(xi*w-b)
    """
    m,n = X.shape
    y2 = [-1 if yi==0 else 1 for yi in y]
    w = np.zeros(n)
    b = 0
    for num in range(num_iter):
        for i, xi in enumerate(X):
            c = y2[i]* (float(np.dot(xi, np.array(w)))-b)
            if c >= 1:
                w = w - l_r * (2 * Lambda * w)
            else:
                w = w - l_r * (2 * Lambda * w - np.dot(xi,y2[i]))
                b = b - l_r * y2[i]
    return w,b

def svm_predict(X,w,b):
    """
    predict the class of y given w and b
    """
    ans = np.dot(X, w) - b
    ans = np.sign(ans)
    ans = np.where(ans==-1,0,1)
    return ans

X = data_clean[:,:13]
Y = np.matrix(data_clean[:,13]).T

# SVM model testing with simulate data
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim =
    train_test_split(X_sim, Y_sim, test_size=0.2, random_state=i)
    w,b = svm_fit(X_train_sim,Y_train_sim)
    Y_pred_sim = svm_predict(X_test_sim,w,b)
    a_s = accuracy_score(Y_test_sim, Y_pred_sim)
    accuracy_list.append(a_s)
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))
```

```
start_time = time.time()
accuracy_list = []
for i in range(10):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim =
    train_test_split(X_sim1, Y_sim1, test_size=0.007, random_state=i)
    w,b = svm_fit(X_train_sim,Y_train_sim)
    Y_pred_sim = svm_predict(X_test_sim,w,b)
    a_s = accuracy_score(Y_test_sim, Y_pred_sim)
    accuracy_list.append(a_s)
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

# SVM modeling performance
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train, X_test, Y_train, Y_test =
    train_test_split(X, Y, test_size=0.2, random_state=i)
    w,b = svm_fit(X_train,Y_train)
    Y_pred = svm_predict(X_test,w,b)
    a_s = accuracy_score(Y_test, Y_pred)
    accuracy_list.append(a_s)
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))
```

[11]

(6) Naive Bayes algorithm

```
"""
*    Title: ML from Scratch
*    Author: Sunrit Jana
*    Date: Jun 12, 2021
*    Code version: 1.0
*    Availability: https://github.com/python-engineer/MLfromscratch/
        blob/master/mlfromscratch/naivebayes.py
"""

def Naivebayes(X_train,y_train,X_test):
    """
    calculate posterior probability based on the Bayes' theorem and
    select the class with the highest probability
    """
    m,n = X_train.shape
    class_y = np.unique(y_train)
    n_class_y = len(class_y)
    mean = np.zeros((n_class_y,n))
    var = np.zeros((n_class_y,n))
    prior = np.zeros(n_class_y)
    posteriors = []
    argmax_list = []
```

```
    for idx, classes in enumerate(class_y):
        X_c = X_train[y_train == classes]
        mean[idx,:] = X_c.mean(axis = 0)
        var[idx,:] = X_c.var(axis = 0)
        prior[idx] = X_c.shape[0] / float(m)
    for x in X_test:
        for idx, classes in enumerate(class_y):
            log_prior = np.log(prior[idx])
            posterior = np.sum(np.log((1/np.sqrt(2*np.pi*var[idx]))*
            (np.exp(-((x- mean[idx])**2)/(2*var[idx])))))
            posterior = log_prior + posterior
            posteriors.append(posterior)
    posteriors = [posteriors[i:i + 2] for i in range(0,len(posteriors),2)]
    for i in range(0,X_test.shape[0]):
        argmax_list.append(np.argmax(posteriors[i]))
    return np.array(argmax_list)

# Naive Bayes model testing with simulate data
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim =
    train_test_split(X_sim, Y_sim, test_size=0.2, random_state=i)
    Y_pred_sim = Naivebayes(X_train_sim,Y_train_sim,X_test_sim)
    a_s = accuracy_score(Y_test_sim, Y_pred_sim)
    accuracy_list.append(a_s)
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim =
    train_test_split(X_sim1, Y_sim1, test_size=0.007, random_state=i)
    Y_pred_sim = Naivebayes(X_train_sim,Y_train_sim,X_test_sim)
    a_s = accuracy_score(Y_test_sim, Y_pred_sim)
    accuracy_list.append(a_s)
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

X = data_clean[:,:13]
Y = data_clean[:,13]
# Naive Bayes modeling performance
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train, X_test, Y_train, Y_test =
    train_test_split(X, Y, test_size=0.2, random_state=i)
    Y_pred = Naivebayes(X_train,Y_train,X_test)
    a_s = accuracy_score(Y_test, Y_pred)
    accuracy_list.append(a_s)
print("run time: " + "%s seconds" % (time.time() - start_time))
```

```
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))
```

[11]

(7) Logistic Regression

```
"""
*    Title: Gradient Descent for Logistics Regression in Python
*    Author: Hoang Phong
*    Date: Jul 30, 2021
*    Code version: 1.0
*    Availability: https://medium.com/@IwriteDSblog/gradient-descent-
                    for-logistics-regression-in-python-18e033775082
"""

def generateXvector(X):
    """
    Add the X0 column that is full of 1s to the dataset X array
    """
    vectorX = np.c_[np.ones((len(X), 1)), X]
    return vectorX

def theta_init(X):
    """
    Initializing a theta vector for all X's variables
    Each theta contains a random number
    This is the initial guess of theta
    """
    theta = np.random.randn(len(X[0])+1, 1)
    return theta

def sigmoid_function(X):
    """
    Calculate the sigmoid value of the inputs
    """
    return 1/(1+math.e**(-X))

def fit(X,y,alpha, iterations, e = 0.001):
    """
    Optimize the logistic regression using Gradient Descent method
    X -> nd-array for independent variables
    y -> 1d-array for response variable
    alpha -> learning rate or step size (float)
    iterations -> numbers of iterations (int)
    """
    y_new = np.reshape(y, (len(y), 1))
    cost_lst = []
    vectorX = generateXvector(X)
    theta = theta_init(X)
    m = len(X)
    #iters = 0
    for i in range(iterations):
        gradients = 2/m * vectorX.T.dot(sigmoid_function(vectorX.dot(theta))-y_new)
```

```
        theta_old = theta.copy()
        theta = theta_old - alpha * gradients
        y_pred = sigmoid_function(vectorX.dot(theta))
        cost_value = - np.sum(np.dot(y_new.T,np.log(y_pred)+ np.dot((1-y_new).T,np.log(1-y
        cost_lst.append(cost_value)
        if np.linalg.norm(theta-theta_old,ord=np.inf)
                /np.linalg.norm(theta,ord=np.inf)<e:
            break
    return theta, cost_lst

def column(matrix, i):
    """
    Returning the values to the specific variable column
    """
    return [row[i] for row in matrix]

def predict(X,y,alpha, iteration,X_test, y_test):
    """
    Predict the y-values based on the trained logistic regression model
    This function also fit the model as well

    """
    ideal = fit(X,y,alpha, iteration)[0]
    hypo_line = ideal[0]
    for i in range(1,len(ideal)):
        hypo_line = hypo_line + ideal[i]*column(X_test,i-1)
    logistic_function = sigmoid_function(hypo_line)
    for i in range(len(logistic_function)):
        if logistic_function[i] >= 0.5:
            logistic_function[i] = 1
        else:
            logistic_function[i] = 0
    return logistic_function

def multi_run(x,y, alpha, run, max_iter):
    '''
    Check if the inputted variables are numpy array or not
    run n times according to user specification
    Split the data into training and testing dataset
    return accuracy, precisions, and recall list
    '''
    ac = []
    f_score = []


    if type(x) != "numpy.ndarray" and type(y) != "numpy.ndarray":
        x = np.array(x)
        y = np.array(y)
    else:
        pass

    for i in range(run):
        X_train, X_test, Y_train, Y_test =
        train_test_split(x, y, test_size = 0.2)
```

```
        Y_pred = predict(X_train, Y_train, alpha, max_iter, X_test, Y_test)
        ac.append(accuracy_score(Y_test, Y_pred))
        f_score.append(f1_score(Y_test, Y_pred))
    return [ac, f_score]

def report(x):
    print("accuracy: {}, F score: {}".format(np.mean(x[0]),np.mean(x[1])))

# Logistic regression model testing with simulate data
start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim =
    train_test_split(X_sim, Y_sim, test_size=0.2, random_state=i)
    Y_pred_sim = predict(X_train_sim, Y_train_sim, 0.1,1000,X_test_sim,Y_test_sim)
    a_s = accuracy_score(Y_test_sim, Y_pred_sim)
    accuracy_list.append(a_s)
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

start_time = time.time()
accuracy_list = []
for i in range(100):
    X_train_sim, X_test_sim, Y_train_sim, Y_test_sim =
    train_test_split(X_sim1, Y_sim1, test_size=0.007, random_state=i)
    Y_pred_sim = predict(X_train_sim,Y_train_sim,0.1,1000,X_test_sim,Y_test_sim)
    a_s = accuracy_score(Y_test_sim, Y_pred_sim)
    accuracy_list.append(a_s)
print("run time: " + "%s seconds" % (time.time() - start_time))
print("Average accuracy: " + str(np.mean(accuracy_list).round(4)))
print("Accuracy variance: " + str(np.var(accuracy_list).round(4)))

def cost_plot(X,y,alpha, iterations, e = 0.001):
    """
    Cost function graph
    The graph shows the change in difference between the actual outcome
    and the predicted outcomes over the span of n iterations.
    """
    y_new = np.reshape(y, (len(y), 1))
    cost_lst = []
    vectorX = generateXvector(X)
    theta = theta_init(X)
    m = len(X)

    for i in range(iterations):
        gradients = 2/m * vectorX.T.dot(sigmoid_function(vectorX.dot(theta))-y_new)
        theta_old = theta.copy()
        theta = theta_old - alpha * gradients
        y_pred = sigmoid_function(vectorX.dot(theta))
        cost_value = - np.sum(np.dot(y_new.T,np.log(y_pred)+ np.dot((1-y_new).T,np.log(1-y
        cost_lst.append(cost_value)
    plt.plot(np.arange(1,len(cost_lst)),cost_lst[1:])
    plt.title('Cost function Graph: lr = {}'.format(alpha))
```

```
    plt.xlabel('Number of iterations')
    plt.ylabel('Cost')

'''
Cost function graph for each learning rate for the simulated data
'''
for j in [0.1,0.25,0.5,0.75,1]:
    for i in range(100):
        x = np.array(X_sim1)
        y = np.array(Y_sim1)
        X_train1, X_test1, Y_train1, Y_test1 =
        train_test_split(x, y, test_size = 0.2)
        cost_plot(X_test, Y_test, j, 1000)
    plt.show()

X = data_clean[:,:13]
Y = np.matrix(data_clean[:,13]).T
for i in [0.1,0.25,0.5,0.75,1]:
    for j in [1000,10000]:
        start_time = time.time()
        run = multi_run(X,Y, i, 100, j)
        print('step size = ' + str(i) + " , iteration = " + str(j))
        print("run time: " + "%s seconds" % (time.time() - start_time))
        report(run)
        print('\n')
'''
Cost function graph for each learning rate for the real data
'''
for j in [0.1,0.25,0.5,0.75,1]:
    for i in range(100):
        x = np.array(X_normal)
        y = np.array(Y)
        X_train, X_test, Y_train, Y_test =
        train_test_split(x, y, test_size = 0.2)
        cost_plot(X_test, Y_test, j, 1000)
    plt.show()

#Sklearn logistic function
def np_logit(x,y, run, max_iter):
    ac = []
    f_score= []
    if type(x) != "numpy.ndarray" and type(y) != "numpy.ndarray":
        x = np.array(x)
        y = np.array(y)
    else:
        pass

    for i in range(run):
        X_train,X_test,Y_train,Y_test=
        train_test_split(x,y,test_size=0.2)
        classifier = LogisticRegression(random_state=0,penalty='none'
         ,max_iter=max_iter)
        classifier.fit(X_train, Y_train)
        classifier.intercept_, classifier.coef_
```

```
        y_pred = classifier.predict(X_test)
        ac.append(accuracy_score(Y_test, y_pred))
        f_score.append(f1_score(Y_test, y_pred))
    return [ac, f_score]

logit1 = np_logit(X_normal,Y, 100, 1000)[0:3]

start = time.time()
logit2 = np_logit(X_normal,Y, 100, 10000)[0:3]
end = time.time()
print(end - start)

report(logit1)
report(logit2)
np.var(logit2)
```

[16]

7.2 Mathematical Proof

7.2.1 Derivative of Gradient Descent for Logistic Regression

Cost Function =

$$(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

re-written as:

$$= \frac{-1}{n} \sum_{i=1}^{n} [ylog(\hat{y}) + (1 - y)log(1 - \hat{y})]$$

Properties:

L = loss function is the entire term inside the bracket
n = number of observations
y = class label of response variable
$\hat{y} = \sigma(z)$ where z is represented by: $z = \theta^T x + b$
b = single parameter
$\sigma(x) = \frac{1}{1+e^x}$

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

(i) Find the derivative of the cost function with respect to θ $[\frac{dCost}{d\sigma}]$

For simplicity in derivation, a = $\sigma(z)$
$\frac{dCost}{da} = \frac{dL}{da} * \frac{da}{dz} * \frac{dz}{dw}$

Using chain rule:
First, find the result of each derivative on left hand side:

Term 1:

$$\frac{dL}{da} = \frac{-y}{a} + \frac{(1-y)}{(1-a)}$$

Term2:

$$\frac{da}{dz} = (1+e^{-z})^{-1} = \frac{e^{-z}}{(1+e^{-z})^2}$$

$$a^2 = \frac{1}{(1+e^{-z})^2}; e^{-z} = \frac{(1-a)}{a}$$

$$\frac{(1-a)}{a} * a^2 = a(1-a)$$

Term 3:

$$\frac{dz}{dw} = x + 0 = x$$

Second, plug in results to find the final answer of one observation from the entire dataset.

$\frac{dL}{d\theta_1} = [\frac{-y}{a} + \frac{1-y}{1-a} * a(1-a) * (x_1)$

$= [\frac{-y+ay+a-(ay)}{a(1-a)} * a(1-a) * (x_1)$

$= [\frac{a-y}{a*(1-a)} * a * (1-a) * (x_1)$

$= (a-y) * (x_1)$

The same computation goes for $theta_2$: $[\frac{dL}{d\theta_2}]$

The same computation goes for $theta_n$ n times: $[\frac{dL}{d\theta_n} = (a-y) * X_n]$

Finally, the main result for the entire matrix of n x p will be: $[\frac{dL}{d\theta} = (A-Y) * X]$

Property of the final result are:
A = matrix representation of all the predictor variable
Y = matrix representation of all output values
X = all the input for all the observation
θ = matrix of all the θ parameter

(ii) Find the derivative of the cost function with respect to b. where $z = \theta_x + b$.

$$\frac{dL}{db_1} = \frac{dL}{da} * \frac{da}{dz} * \frac{dz}{db}$$

Where $\frac{dz}{db} = 1$,

$$\frac{dL}{db_1} = \frac{(a-y}{a(1-a)} * a(1-a) * 1$$

$$\frac{dL}{db_1} = (a-y)$$

Then we get:

$$\frac{dL}{db_n} = (A-Y)$$

Q.E.D

[19]

7.2.2 Neural Network Method

Properties of Neural Network Structure:
(I) Input Layer = vector representation of input data
(II) Weight 1, weight 2 = weights of each node between the input layer and hidden layer and weights of each node between the hidden layer and output layer.
(III) Bias 1, bias 2 = biases of each node of the hidden layer and biases of the output layer.
(IV) Output Layer = vector representation of the probability of each class

(1) Forward Propagation
This step calculates the cost function and the output data (y)

m = number of rows from input data
A = hidden layer, matrix that contains data from input layer [n, m]
$(Z) = \frac{1}{1+exp(-z)}$
$Z = Wx + b$
W = Weight/loss
X = Input data
b = Bias constant

Step One:
$A^{[0]} = X_[n * m]$
$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$
Step Two:
Activation Function for 1st node:
$A^{[1]} = g(Z^{[1]}) = ReLU(Z^{[1]})$
ReLU is rectified linear unit:

$$(ReLU(x)) = \begin{cases} X & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
Step Three:
Activation Function for 2nd node $A^{[2]} = softmax(Z^{[2]}$ softmax activation function returns probability over the predicted classes.
It divides each element over the total sum of it.

Properties of Softmax Function:
$\sigma(\overrightarrow{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$
σ = is the softmax function
$\overrightarrow{z}$ = probability of the input vector
k = number of class label for multi-class classifier
e^{z_i} = standard exponential function for input vector
e^{z_j} = standard exponential function for output vector

(2) Backward Propagation
This step calculates the gradient descent by updating parameter that checks how much the prediction deviated from the actual label.

Derivative of Cost Function $dZ^{[2]} = A^{[2]} - Y$
Derivative of Loss with respect to Weight $dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1][T]}$
Derivative of Loss with respect to Bias "average of absolute error" $db^{[2]} = \frac{1}{m} \sum_i dz$

$dZ^{[1]} = W^{[2][T]} dZ^{[2]} g^{[z^{[1]}]}$
$g^{[z^{[1]}]}$ is the derivation to undo the activation function
Repeats derivation calculation for 2nd and remaining nodes
$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{[T]}$
$db^{[2]} = \frac{1}{m} \sum_i dZ^{[1]}$

(3) Update Parameter using Gradient Descent

$\alpha = LearningRate$
$W^{[1]} = W^{[1]} - \alpha dW^{[1]}$
$b^{[1]} = b^{[1]} - \alpha db^{[1]}$
$W^{[2]} = W^{[2]} - \alpha dW^{[2]}$
$b^{[2]} = b^{[2]} - \alpha db^{[2]}$

(4) Iteration of Gradient Descent

repeat { $(W_n = W_n - \alpha(\frac{d-CostFunction}{dW_n})$,
$(b_n = b_n - \alpha(\frac{d-CostFunction}{db_n})$ }
This process states at iteration n,
we are at some point of X_n,
then pick a vector b_n and,
follow the process until we reach $X_n + 1$
[1]

7.2.3 Support Vector Machine:

SVM Model maximizes the boundary margin between data and hyperplane to minimize the loss.

Properties:
x = input data in vector form $X \in \mathbb{R}^{[d]}$
R^d = vector space with D = 13 dimension

W = $\mathbb{R}^{[d]} -> \mathbb{R}^{[m]}$ or W $\in \mathbb{R}^{[m]}$, transform feature space for each input transformed to basis vector
y = output with classified label in vector form

$$y = \begin{cases} (wx - b) \geq 1 & \text{if } y_i = 1 \\ (wx - b) \leq 1 & \text{if } y_i = -1 \end{cases}$$

w = Weight Vector
b = Bias Constant
λ = Regularization parameter
l_r = Learning Rate
$f(x) = wx - b$
(A): Hinge Loss function helps to maximize data points and hyperplane.

$$c(x, y, f(x)) = \begin{cases} cost = 0 & \text{if } y * f(x) \geq 1 \\ cost = (1 - y * f(x)) & \text{otherwise} \end{cases}$$

(B): Regularization parameter added to balance the boundary margin maximum and loss.
$min_w \lambda ||w||^2 + \sum_{i=1}^{n} (1 - y_i f(x_i))$

(C): Learn the parameter using Gradient Descent

(i): Derivative of Loss Function with respect to weight finds gradient
$\frac{d}{dW_k}\lambda||w||^2 = 2\lambda w_k$

$$\frac{d}{dW_k}(1 - y_i f(x_i)) = \begin{cases} 0 & \text{if } y_i f(x_i) \geq 1 \\ (1 - y_i f(x_i)) & \text{otherwise} \end{cases}$$

(D): Update weights

(i) If model predicts the label correctly, then we only update gradient
$H_i = \lambda||w||^2$
$w = w - \alpha * (2\lambda * w)$

(ii) If model predicts the label incorrectly, then we update both gradient and regularization parameter

$H_i = \lambda||w||^2 + y(wx - b)$
$w = w - \alpha * 2(\lambda * w - x_i y_i)$
$b = b - \alpha * y_i$

(E): SVM Prediction in Hyperplane Equation

$$y_n = \begin{cases} 1 & \text{if } (w_i x_i - b) > 0 \\ -1 & \text{if } (w_i x_i - b) < 0 \end{cases}$$

[8]

7.2.4 Multinomial naive Bayes

Probablistic Model

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)}$$

- p($C_k|x$) = posterior
- p(C_k) = prior
- p(x|C_k) = likelihood
- p(x) = evidence

Conditional Probability

$$\begin{aligned} p(C_k, x_1, ..., x_n) &= p(x_1, ..., x_n, C_k) \\ &= p(x_1|x_2, ..., x_n, C_k)p(x_2, ..., x_n, C_k) \\ &= p(x_1|x_2, ..., x_n, C_k)p(x_2|x_3, ..., x_n, C_k)p(x_3, ..., x_n, C_k) \\ &= ... \\ &= p(x_1|x_2, ..., x_n, C_k)p(x_2|x_3, ..., x_n, C_k)...p(x_{n-1}|x_n, C_k)p(x_n|C_k)p(C_k) \end{aligned}$$

naive Conditional Probability

$$\begin{aligned} p(C_k|x_1, ..., x_n) &\propto p(C_k, x_1, ..., x_n) \\ &\propto p(C_k)p(x_1|C_k)p(x_2|C_k)p(x_3|C_k)... \\ &\propto p(x_1|x_2, ..., x_n, C_k)p(x_2|x_3, ..., x_n, C_k)p(x_3, ..., x_n, C_k) \\ &\propto p(C_k)\prod_{i=1}^{n} p(x_i|C_k) \end{aligned}$$

[22]

Conditional distribution over the class variable C is:

$$p(C_k|x_1,..,x_n) = \frac{1}{Z}p(C_k)\prod_{i=1}^{n} p(x_i)|C_k$$

- Z is the evidence.
  - Z is a scaling factor dependent on $x_1,...,x_n$.

Bayes Classifier Constructed from the Probability Model

$$\hat{y} = \underset{k\in 1,...,k}{\operatorname{argmax}}\, p(C_k)\prod_{i=1}^{n} p(X_i|C_k)$$

Multinomial naive Bayes

- p_i = the probability that even i occurs
- $x = (x_1,...,x_3)$ is a vector containing the features
  - x_i = the number of times event i was observed in a particular interval

The likelihood observing X is given by:

$$p(x|C_k)|\frac{(\sum_{i=1}^{n} x_i)}{\prod_{i=1} nx_i}\prod_{i=1} np_{ki}^{x_i}$$

Taking the log of the multinomial naive Bayes classifier turns it into linear classifier:

$$\begin{aligned} log(p(C_k|x)) &\propto log(p(C_k)\prod_{i=1}^{n} p_{k_i}^{x_i} \\ &= log(p(C_k)) + \sum_{i=1}^{n} x_i log(p_{x_i})^{)} \\ &= b + W_k^T X \end{aligned}$$

[17]

7.2.5 K-Nearest Neighbor with Cross-Validation

K-nearest neighbor

- X_i = input sample
- p = number of features
- n = total number of input samples
- p = total number of features
- R_i = Voronoi Cell
- X = all points within Voronoi Cell

Euclidean Distance between $x_i and x_i$ $d(x_i, x_l) = \sqrt{(x_{i1}-x_{l1})^2 + (x_{i2}-x_{l2})^2 + ... + (x_{ip}-x_{lp})^2} R_i = x \in^p: d(x, x_i) \leq d($

- The K-Nearest-Neighbor assign the test samples the category label based on this characteristic

[15]

K-fold Cross-Validation

- K = Number of folds
- MSE_i = Test MSE on the i^{th} iteration

$$\frac{1}{k}\sum_{i=1}^{n} MSE_i$$

7.2.6 Gradient Descent

Gradient Descent method iteration formula

- w = regression coefficient
- Y = response variable

$$w = w - \alpha \sum_{1}^{m} (y'(1-y')x_j)$$

Proof to solving Gradient Descent

$$\frac{\partial Y}{\partial w_j} = \sum_{1}^{m}((y'-y)\frac{\partial y'}{\partial w_i})$$

$$\sigma^{'(z)} = n\sigma(z)(1-n\sigma^{('}z))$$

$$\frac{\partial y'}{\partial w_j} = ny'(1-y')\frac{w^T X_i}{\partial w_j}$$

$$\frac{w^T X_i}{\partial w_j} = \frac{\partial(w_0x_0 + w_1x_1 + w_2x_2 + ... + w_nx_n)}{\partial w_j}$$

- j = 0:
  - w_i is obtained through the partial derivative
  - x_i is obtained

$$\frac{\partial w^T x}{\partial w_j} = \chi_j$$

The combination of the previous five derivative formulas results in:

$$\frac{\partial Y}{\partial w_j} = \sigma_1^m(y'(1-y')x_j)$$

[24]

References

[1] Opetunde Adepoju. *A step-by-step tutorial on coding Neural Network Logistic*. [Domain Name: Medium, Retrieved:12 March 2022]. Aug. 2019. URL: https://theopetunde.medium.com/a-step-by-step-tutorial-on-coding-neural-network-logistic-regression-model-from-scratch-5f9025bd3d6.

[2] AskPython. *K-Nearest Neighbors from Scratch with Python*. [Domain Name: AskPython, Retrieved:12 March 2022]. Aug. 2022. URL: https://www.askpython.com/python/examples/k-nearest-neighbors-from-scratch.

[3] Peter M Atkinson and R Massari. "Generalised linear modelling of susceptibility to landsliding in the central Apennines, Italy". In: *Computers & Geosciences* 24.4 (1998), pp. 373–385.

[4] Robert Berwick. "An Idiot's guide to Support vector machines (SVMs)". In: *Retrieved on October* 21 (2003), p. 2011.

[5] Ekaba Bisong. *Building machine learning and deep learning models on Google cloud platform: A comprehensive guide for beginners*. Apress, 2019.

[6] Cherngs. *Heart Disease Cleveland UCI*. [Retrieved: 02/19/2022, Version: 1]. Mar. 2020. URL: https://www.kaggle.com/cherngs/heart-disease-cleveland-uci/metadata.

[7] Center of Disease Control. *Heart disease facts*. [Online; accessed 10-March-2022]. Feb. 2022. URL: https://www.cdc.gov/heartdisease/facts.htm.

[8] Rohit Gandhi. *Support Vector Machine — Introduction to Machine Learning*. [Domain Name: Medium, Retrieved: 11 March 2022]. July 2018. URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

[9] Akansh Gupta et al. "Heart disease prediction using classification (naive bayes)". In: *Proceedings of First International Conference on Computing, Communications, and Cyber-Security (IC4S 2019)*. Springer. 2020, pp. 561–573.

[10] Inc HeartFlow. *Misdiagnosis is a Top Concern of Cardiac Patients*. [accessed: March 10, 2022]. Feb. 2019. URL: https://www.heartflow.com/articles/MOH_misdiagnosis/.

[11] Sunrit Jana. *ML From Scratch*. 21 June 2021. URL: https://github.com/python-engineer/MLfromscratch/blob/master/mlfromscratch/svm.py.

[12] Andrey Kim et al. "Logistic regression model training based on the approximate homomorphic encryption". In: *BMC medical genomics* 11.4 (2018), pp. 23–31.

[13] Samaya Madhavan. *Supervised deep learning*. [access: March 16, 2022, authors: , Sidra Ahmed, Sandhya Nayak, Dhivya Lakshminarayanan, Mostafa Abdelaleem, M. Tim Jones, Casper Hansen]. July 2021. URL: https://developer.ibm.com/learningpaths/supervised-deep-learning/.

[14] WHO/Quinn Mattingly. *Cardiovascular diseases*. [Retrieved: 12-March-2022]. Jan. 2022. URL: https://www.who.int/health-topics/cardiovascular-diseases#tab=tab_1.

[15] L. E. Peterson. "K-nearest neighbor". In: *Scholarpedia* 4.2 (2009). revision #137311, p. 1883. DOI: 10.4249/scholarpedia.1883.

[16] Hoang Phong. *Gradient Descent for Logistics Regression in Python*. [$\text{code}_version : 1.0, type : opensource,$]. July 2021. URL: https://medium.com/@IwriteDSblog/gradient-descent-for-logistics-regression-in-python-18e033775082.

[17] Jason D Rennie et al. "Tackling the poor assumptions of naive bayes text classifiers". In: *Proceedings of the 20th international conference on machine learning (ICML-03)*. 2003, pp. 616–623.

[18] James M Rippe and Theodore J Angelopoulos. "Lifestyle strategies for risk factor reduction, prevention and treatment of cardiovascular disease". In: *Lifestyle Medicine* (2019), pp. 19–36.

[19] Neha Seth. *Is Gradient Descent sufficient for Neural Network?* [Domain Name: Analytics Vidhya, Retrieved: 11 March 2022. Apr. 2021. URL: https://www.analyticsvidhya.com/blog/2021/04/is-gradient-descent-sufficient-for-neural-network/.

[20] Jill C Stoltzfus. *Logistic regression: a brief primer*. 2011.

[21] Janice Hopkins Tanne. *Covid-19: Even mild infections can cause long term heart problems, large study finds*. 2022.

[22] Wikipedia contributors. *Naive Bayes classifier — Wikipedia, The Free Encyclopedia*. [Online; accessed 13-March-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Naive_Bayes_classifier&oldid=1075188874.

[23] SAMSON ZHANG. *Simple MNIST NN from scratch (numpy, no TF/Keras)*. [souce: Apache 2.0 open source license , code version: 1.0]. Nov. 2020. URL: https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras/notebook.

[24] Xiaonan Zou et al. "Logistic Regression Model Optimization and Case Analysis". In: *2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT)*. 2019, pp. 135–139. DOI: 10.1109/ICCSNT47585.2019.8962457.

Figure 2: Cost Function of Learning Rate(s) for Real Data