A
PROJECT REPORT
ON
EYE ILLNESS DETECTION
USING MACHINE LEARNING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
OF
SILICON INSTITUTE OF TECHNOLOGY, BHUBANESWAR
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE SKILL LAB AND PROJECT

Sagar Verma (SIC-20BCSE64)
Subhasis Nayak (SIC-20BCSD29)
6th Semester B. Tech.(CSE), Section: B
September 2023

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a Project Report for the 6th Semester B. Tech. SKill Lab and Project.

(DR Satyananda Champati Rai) Principal Adviser

Preface

Welcome to "A Systematic Review of Machine Learning Methods for Eye Illness Detection". This review aims to provide a comprehensive overview of the application of machine learning techniques in the detection of eye illnesses. Eye illnesses are a major health concern that can lead to significant consequences, including vision loss and blindness. Therefore, early detection is critical for effective treatment and patient outcomes.

Machine learning is a rapidly evolving field that has shown great potential in various areas of healthcare, including ophthalmology. By analyzing large amounts of data, machine learning algorithms can identify patterns and make predictions that can help improve the accuracy and efficiency of eye illness detection.

In this review, we will explore the different machine learning techniques used in the detection of eye illnesses, including supervised and unsupervised learning, along with different algorithms such as decision trees and neural networks. We will also discuss the challenges and limitations of these techniques, as well as their potential applications in clinical practice.

We hope that this review will serve as a valuable resource for researchers and healthcare professionals interested in advancing the field of eye illness detection through the application of machine learning techniques.

Acknowledgments

I would like to express my sincere gratitude to all those who have supported me throughout my Skill Lab and Project Thesis. Their unwavering support, encouragement, and patience have been instrumental in the successful completion of this project.

First and foremost, I would like to extend my deepest appreciation to my project guide, Dr. Satyananda Champati Rai, for his invaluable guidance, support, and encouragement throughout the project. His expertise and insight have been invaluable in shaping this thesis into its final form. I am truly grateful for his mentorship and dedication.

I would also like to express my thanks to the Lab Attendant, Mr. Dibakar Pradhan, for his support and guidance. He provided me with insightful feedback and suggestions throughout the thesis, which greatly improved the quality of my work.

Lastly, I would like to thank my friends who have supported me throughout this project. Their valuable feedback and suggestions have been invaluable, and their unwavering support has kept me going during the challenging times.

Once again, I would like to express my heartfelt gratitude to all those who have contributed to the successful completion of this project.

Contents

Preface **iv**

Acknowledgments **v**

1 Introduction To Machine Learning **2**

1.1 Supervised learning . . . 2
1.2 Unsupervised learning . . . 2
1.3 Classification . . . 3
1.4 Regression . . . 3
1.5 Error Analysis . . . 3
1.6 Reinforcement Learning . . . 3
1.7 Objectives . . . 4
1.8 Chapter Outline . . . 4
1.9 Conclusion . . . 5

2 Mathematical Foundation Of Machine Learning **6**

2.1 Matrix Calculus . . . 6
2.2 Eigen values and Eigen Vectors . . . 7
2.3 Random variable and Distribution . . . 7
2.3.1 Normal Distribution . . . 8
2.3.2 Gaussian Distribution . . . 8
2.4 Kernel . . . 9
2.4.1 Polynomial Kernel . . . 9
2.4.2 Gaussian Kernel . . . 10

3 Implementation of Machine Learning Algorithms **11**

3.1 K-Nearest Neighbour - Classification . . . 11
3.1.1 Pseudocode . . . 11
3.1.2 Code . . . 12

3.2 K-Nearest Neighbour - Regression . . . 12
3.2.1 Pseudocode . . . 12
3.2.2 Code . . . 13
3.3 Multiple Linear Regression . . . 13
3.3.1 Pseudocode . . . 14
3.3.2 Code . . . 14
3.4 Ridge Regression . . . 15
3.4.1 Pseudocode . . . 15
3.4.2 Code . . . 15
3.5 LASSO Regression . . . 16
3.5.1 Pseudocode . . . 16
3.5.2 Code . . . 17
3.6 Logistic Regression . . . 18
3.6.1 Pseudocode . . . 18
3.6.2 Code . . . 18
3.7 Support Vector Machine - Classification . . . 19
3.7.1 Pseudocode . . . 19
3.7.2 Code . . . 20
3.8 Support Vector Machine - Regression . . . 21
3.8.1 Pseudocode . . . 21
3.8.2 Code . . . 22
3.9 Decision Tree for Classification . . . 22
3.9.1 Pseudocode . . . 23
3.9.2 Code . . . 23
3.10 Decision Tree for Regression . . . 24
3.10.1 Pseudocode . . . 24
3.10.2 Code . . . 25
3.11 Naive Bayes' Classifier . . . 25
3.11.1 pseudocode . . . 26
3.11.2 Code . . . 26
3.12 Random Forest . . . 27
3.12.1 Pseudocode . . . 27
3.12.2 Code . . . 27
3.13 K-Means Clustering . . . 28
3.13.1 Pseudocode . . . 28
3.13.2 Code . . . 29
3.14 Spectral Clustering . . . 30

3.14.1 Pseudocode . . . 30
3.14.2 Code . . . 31
3.15 MDP: Markov Decision Process . . . 32
3.16 MRP: Markov Reward Process . . . 32
3.17 Queue - Learning . . . 33
3.18 SARSA - Learning . . . 33

4 Proposed Model **34**
4.1 Background . . . 34
4.2 Investigating the Limitation . . . 35
4.3 Dataset Selected . . . 35
4.4 Pseudocode :Design of the Model to overcome the limitaion . . . 35
4.4.1 Pseudocode for KNN - C . . . 35
4.4.2 Pseudocode for SVM - C . . . 36
4.5 Program: Functional Part . . . 37
4.5.1 KNN C . . . 37
4.5.2 SVM C . . . 38
4.5.3 RANDOM FOREST . . . 39
4.5.4 CNN using ResNet-50 . . . 40
4.5.5 Introduction . . . 40
4.5.6 Organizing Data . . . 40
4.5.7 Looking at the Data . . . 41
4.5.8 Implementation . . . 41
4.6 Performance analysis of the Model . . . 46
4.6.1 KNN . . . 46
4.6.2 SVM . . . 47
4.6.3 RANDOM FOREST . . . 48

5 Future Work and Conclusion **49**

Bibliography **50**

List of Tables

4.1 Eye illness dataset 41

List of Figures

https://www.overleaf.com/project/640018de8f5f4d619768c247

Chapter 1

Introduction To Machine Learning

1.1 Supervised learning

Supervised learning, also called supervised machine learning, is a subset of artificial intelligence (AI) and machine learning.Supervised learning involves using labeled datasets to train computer algorithms for a particular output. As the user feeds input data to the model, the system adjusts to predict outcomes and classify data more accurately by cross validating—adjusting its weights to more closely fit the model.In Supervised Learning, the machine learns under supervision. It contains a model that is able to predict with the help of a labeled dataset. A labeled dataset is one where you already know the target answer. The main aim of a supervised learning algorithm is to find a mapping function to map the input variable(x) with the output variable(y).

1.2 Unsupervised learning

Unsupervised learning, also known as unsupervised machine learning, uses machine learning algorithms to analyze and cluster unlabeled datasets.

These algorithms discover hidden patterns or data groupings without the need for human intervention.Unsupervised learning has no supervisor, and no correct answers[3]. In unsupervised learning, information is unsorted, and instead grouped according to similarities and differences.

1.3 Classification

Classification is a supervised machine learning method where the model tries to predict the correct label of a given input data. In classification, the model is fully trained using the training data, and then it is evaluated on test data before being used to perform prediction on new unseen data. A classification problem is when the output variable is a category, such as "red" or "blue" or "disease" and "no disease". A classification model attempts to draw some conclusion from observed values. Given one or more inputs a classification model will try to predict the value of one or more outcomes.

1.4 Regression

Regression is a statistical method used in finance, investing, and other disciplines that attempts to determine the strength and character of the relationship between one dependent variable (usually denoted by Y) and a series of other variables (known as independent variables).Regression analysis is a powerful tool for uncovering the associations between variables observed in data, but cannot easily indicate causation. It is used in several contexts in business, finance, and economics. For instance, it is used to help investment managers value assets and understand the relationships between factors such as commodity prices and the stocks of businesses dealing in those commodities.

1.5 Error Analysis

Error analysis is the process to isolate, observe and diagnose erroneous ML predictions thereby helping understand pockets of high and low performance of the model. When it is said that "the model accuracy is 90 percent" it might not be uniform across subgroups of data and there might be some input conditions which the model fails more. So, it is the next step from aggregate metrics to a more in-depth review of model errors for improvement.

1.6 Reinforcement Learning

Reinforcement Learning(RL) is a type of machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences.

Reinforcement Learning is a feedback-based Machine learning technique in which an agent learns to behave in an environment by performing the actions and seeing the results of actions. For each good action, the agent gets positive feedback, and for each bad action, the agent gets negative feedback or penalty.

1.7 Objectives

1. The project aims to develop a machine learning model for early detection of eye illnesses using fundus images.
2. The model will be trained on a large dataset of fundus images to identify patterns and features that indicate the presence of an eye illness.
3. Different machine learning algorithms will be compared to identify the most accurate and efficient algorithm for the task.
4. The project will explore the impact of different preprocessing techniques on the performance of the machine learning algorithms.
5. The model will be validated on a separate test set of fundus images to assess its accuracy and generalizability.
6. The project will investigate the feasibility of integrating the developed model into existing clinical workflows for early detection of eye illnesses.
7. The ethical and regulatory considerations associated with implementing a machine learning-based eye illness detection system in clinical settings will be explored.
8. The project will identify potential features or combinations of features that are most informative for detecting different types of eye illnesses.
9. The performance of the developed machine learning model will be compared with existing approaches for eye illness detection to assess its effectiveness.
10. The project will conclude with a summary of the main findings, limitations, and recommendations for future research in this area.

1.8 Chapter Outline

The project on Eye Illness Detection using Machine Learning aims to improve the accuracy of detecting eye illnesses through the use of machine learning algorithms. It begins with an introduction to the importance of early detection and treatment of eye illnesses and a review of existing methods

for detection. The project then provides a literature review of eye illness detection using machine learning, relevant algorithms and techniques, and related work in the field. The dataset used in the study is described, along with data preprocessing techniques and data visualization and exploration.

Machine learning models are then discussed, including supervised and unsupervised learning, and evaluation metrics for assessing model performance are explored. The experimental design and methodology are presented, including the selection of appropriate machine learning algorithms and features for the detection of eye illnesses. The project also analyzes the importance of feature selection and model interpretability in the context of eye illness detection.

The experimental results and performance evaluation of the machine learning models are presented, including comparisons with existing approaches and the analysis of feature importance and model interpretability. The project concludes with a summary of the main findings, implications, and limitations of the study, recommendations for future research, and technical information in the appendix. Additionally, the ethical and regulatory considerations associated with implementing a machine learning-based eye illness detection system in clinical settings are explored.

1.9 Conclusion

Machine learning is a rapidly growing field with the potential to revolutionize numerous industries and transform the way we approach problem-solving. By leveraging data and computational power, machine learning algorithms can analyze patterns, make predictions, and identify insights that may be difficult for humans to detect. However, as with any technology, machine learning also poses challenges such as overfitting, bias, and lack of interpretability. It is important for individuals and organizations to carefully consider these challenges and implement strategies to address them in order to ensure the reliability and effectiveness of machine learning models. Despite these challenges, the future of machine learning appears to be bright, with endless possibilities for new and innovative applications. It is essential for us to continue to invest in this field and stay informed about new developments so that we can fully realize the potential benefits of machine learning.

Chapter 2

Mathematical Foundation Of Machine Learning

2.1 Matrix Calculus

Matrices are a foundational element of linear algebra.Matrices are used throughout the field of machine learning in the description of algorithms and processes such as the input data variable (X) when training an algorithm. Matrices Rectangular array of numbers, 2D array m X n matrix.
1.Addition of two matrices:- To add two matrices :
add the numbers in the matching positions.
2.Subtraction of two matrices To subtract two matrices :
subtract the numbers in the matching positions.
Scalar multiplication 3.To multiply a matrix with a scalar value :
multiply each and every elements in a matrix with external scalar value.
4.Vector multiplication :
To multiply a matrix with vector matrix multiply a row vector with column vector. The row vector must have as many columns as the column vector has rows.
5.Multiplying a matrix by an another matrix :
To multiply two matrices : we need to do dot product of rows and columns.
6.Identity and Inverse Matrix :
Identity matrix is the matrix equivalent of 1 If a co-variance is an identity matrix then the mathematical operations become easier.
7.Transpose Matrix :
A matrix which is formed by turning all the rows of a given matrix into columns and vice-versa. The transpose of matrix A is written A^T.

2.2 Eigen values and Eigen Vectors

Eigenvectors are unit vectors, which means that their length or magnitude is equal to 1.0. They are often referred as right vectors, which simply means a column vector (as opposed to a row vector or a left vector). A right-vector is a vector as we understand them. Eigenvalues are coefficients applied to eigenvectors that give the vectors their length or magnitude. For example, a negative eigenvalue may reverse the direction of the eigenvector as part of scaling it. A matrix that has only positive eigenvalues is referred to as a positive definite matrix, whereas if the eigenvalues are all negative, it is referred to as a negative definite matrix.

2.3 Random variable and Distribution

In probability, a random variable is a real valued function whose domain is the sample space of the random experiment. It means that each outcome of a random experiment is associated with a single real number, and the single real number may vary with the different outcomes of a random experiment. Hence, it is called a random variable and it is generally represented by the letter "X". For example, let us consider an experiment for tossing a coin two times. Hence, the sample space for this experiment is S = HH, HT, TH, TT If X is a random variable and it denotes the number of heads obtained, then the values are represented as follows: X(HH) = 2, X(HT) = 1, X(TH) = 1, X(TT) = 0. Similarly, we can define the number of tails obtained using another variable, say Y. (i.e) Y(HH) = 0, Y(HT) = 1, Y(TH) = 1, Y(TT)= 2.

The description of how likely a random variable takes one of its possible states can be given by a probability distribution. Thus, the probability distribution is a mathematical function that gives the probabilities of different outcomes for an experiment. More generally it can be described as the function which maps an input space A — related to the sample space — to a real number, namely the probability. For the above function to characterize a probability distribution, it must follow all of the Kolmogorov axioms:

1.Non-negativity

2.No probability exceeds

3.Additivity of any countable disjoint (mutually exclusive) events

The way we describe a probability distribution depends on whether the random variable is discrete or continuous, which will result in a probability mass or density function respectively.

2.3.1 Normal Distribution

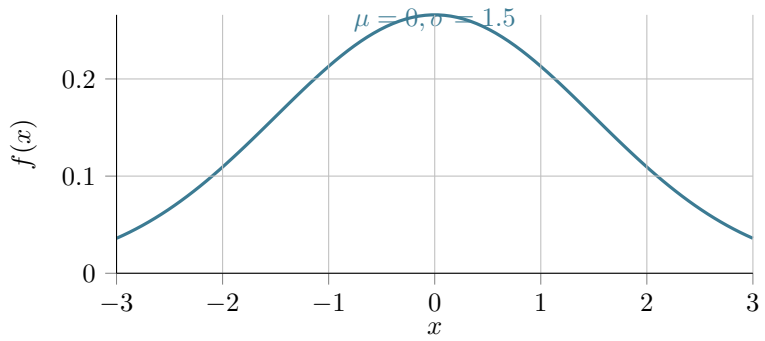
Normal distribution is a bell-shaped curve, and it is assumed that during any measurement values will follow a normal distribution with an equal number of measurements above and below the mean value. The probability density function of the normal distribution, also known as the Gaussian distribution, is given by:

The probability density function of the normal distribution is given by:

The normal distribution, also known as the Gaussian distribution, is a probability distribution that is often used in statistical analysis. It is characterized by two parameters: the mean μ and the standard deviation σ. The probability density function of the normal distribution is given by:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

where x is the random variable, μ is the mean, and σ is the standard deviation.



The normal distribution is symmetric around the mean, with the highest probability density occurring at the mean. The standard deviation determines the spread of the distribution, with larger values resulting in a wider distribution.

2.3.2 Gaussian Distribution

Gaussian distribution is a bell-shaped curve, and it is assumed that during any measurement values will follow a normal distribution with an equal number of measurements above and below the mean value. The Gaussian distribution, also known as the normal distribution, is a probability distribution that is often used in statistics and machine learning. It has the following probability density function:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is the mean of the distribution and σ^2 is its variance.

We can also express the Gaussian distribution in terms of its cumulative distribution function (CDF), which gives the probability that a random variable X drawn from the distribution is less than or equal to a given value x:

$$F(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right)\right]$$

where $\text{erf}(x)$ is the error function.

The mean and variance of the Gaussian distribution are given by:

$$\begin{aligned} \text{E}[X] &= \mu \\ \text{Var}[X] &= \sigma^2 \end{aligned}$$

The Gaussian distribution is often used as a model for many natural phenomena, such as the distribution of heights and weights in a population, the errors in scientific measurements, and the noise in electronic signals.

2.4 Kernel

Kernel is a computer program at the core of a computer's operating system and generally has complete control over everything in the system. It is the portion of the operating system code that is always resident in memory and facilitates interactions between hardware and software components.

2.4.1 Polynomial Kernel

The polynomial kernel is a kernel function commonly used with support vector machines and other kernelized models, that represents the similarity of vectors in a feature space over polynomials of the original variables, allowing learning of non-linear models. The polynomial kernel is a type of kernel function used in machine learning for support vector machines (SVMs) and other kernel-based methods. It has the form:

$$K(x, y) = (\langle x, y\rangle + c)^d$$

where x and y are feature vectors, $\langle x, y\rangle$ denotes the dot product of the vectors, c is a constant, and d is the degree of the polynomial.

In SVMs, the polynomial kernel is often used to handle non-linearly separable data by transforming the data into a higher-dimensional feature space, where it may be linearly separable. The degree of the polynomial determines the degree of the transformation, and the constant c controls the influence of low-degree versus high-degree polynomial terms.

2.4.2 Gaussian Kernel

The Gaussian kernel is the physical equivalent of the mathematical point. It is not strictly local, like the mathematical point, but semi-local. It has a Gaussian weighted extent, indicated by its inner scale s. The Gaussian kernel is a popular choice for smoothing data and estimating probability density functions. It is defined as:

$$K(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

where x is the distance from the center of the kernel, and σ is a parameter that controls the width of the kernel. Larger values of σ result in a wider kernel that smooths over a larger range of values.

The Gaussian kernel can be used in a variety of applications, such as image processing, machine learning, and signal processing. In machine learning, it is commonly used as a similarity measure in kernel methods, such as support vector machines and Gaussian processes.

Chapter 3

Implementation of Machine Learning Algorithms

3.1 K-Nearest Neighbour - Classification

K-Nearest Neighbour classification is a supervised learning algorithm used for classification and regression analysis. It finds the k-nearest neighbors of new input data based on a distance metric and assigns it to the class that is most common among the k neighbors. It is simple to implement but can be computationally expensive and its performance depends on the choice of k and distance metric.

3.1.1 Pseudocode

procedure KNNClassification(X, Y, x_{test}, k)
    $D \leftarrow$ array of size n, where n is the number of training examples
    for $i \leftarrow 1$ to n **do**
        $D_i \leftarrow$ distance between x_{test} and X_i
    end for
    $indices \leftarrow$ indices of the k smallest values in D
    $Y_{neighbors} \leftarrow$ classes of the k nearest neighbors, based on their indices in Y
    $y_{pred} \leftarrow$ class with the highest frequency in $Y_{neighbors}$
    return y_{pred}
end procedure

3.1.2 Code

```python
import pandas as pd
import numpy as np
from collections import Counter

data = pd.read_csv('heart_disease_dataset.csv')
data = (data - data.mean()) / data.std()
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

def knn(X_train, y_train, X_test, k):
    distances = np.sqrt(np.sum((X_train - X_test)**2, axis=1))
    k_nearest_indices = np.argsort(distances)[:k]
    k_nearest_labels = y_train[k_nearest_indices]
    most_common_label = Counter(k_nearest_labels).most_common(1)[0][0]
    return most_common_label

train_size = int(len(X) * 0.8)
X_train = X[:train_size]
y_train = y[:train_size]
X_test = X[train_size:]
y_test = y[train_size:]

predictions = []
k = 5
for i in range(len(X_test)):
    prediction = knn(X_train, y_train, X_test[i], k)
    predictions.append(prediction)
accuracy = np.mean(predictions == y_test)
print("Accuracy:", accuracy)
```

Listing 3.1: KNN Classification

3.2 K-Nearest Neighbour - Regression

KNN regression is a non-parametric method that, in an intuitive manner, approximates the association between independent variables and the continuous outcome by averaging the observations in the same neighbourhood.

3.2.1 Pseudocode

1: **procedure** KNNREGRESSION(X, Y, x_{test}, k)
2: $D \leftarrow$ array of size n, where n is the number of training examples

for $i \leftarrow 1$ to n **do**
    $D_i \leftarrow$ distance between x_{test} and X_i
end for
$indices \leftarrow$ indices of the k smallest values in D
$Y_{neighbors} \leftarrow$ values of the k nearest neighbors, based on their indices in Y
$y_{pred} \leftarrow$ mean of $Y_{neighbors}$
return y_{pred}
end procedure

3.2.2 Code

```python
import numpy as np

class KNNRegressor:
    def __init__(self, k):
        self.k = k

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test):
        predictions = []
        for i in range(len(X_test)):
            distances = []
            for j in range(len(self.X_train)):
                dist = np.sqrt(np.sum((X_test[i] - self.X_train[j])**2))
                distances.append((dist, self.y_train[j]))
            distances.sort()
            k_nearest = distances[:self.k]
            k_nearest_labels = [x[1] for x in k_nearest]
            prediction = sum(k_nearest_labels) / self.k
            predictions.append(prediction)
        return np.array(predictions)
```

Listing 3.2: KNN Regression

3.3 Multiple Linear Regression

Multiple linear regression (MLR), also known simply as multiple regression, is a statistical technique that uses several explanatory variables to predict the outcome of a response variable. Multiple regression is an extension of linear (OLS) regression that uses just one explanatory variable.

3.3.1 Pseudocode

procedure MLR(X, Y)
    $n, p \leftarrow$ dimensions of X
    $\bar{X} \leftarrow$ mean of each column in X
    $\bar{Y} \leftarrow$ mean of Y
    $S_{xx} \leftarrow \sum_{i=1}^{n}(X_i - \bar{X})(X_i - \bar{X})^T$
    $S_{xy} \leftarrow \sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})$
    $B \leftarrow (S_{xx})^{-1}S_{xy}$
    $b_0 \leftarrow \bar{Y} - B^T\bar{X}$
    return b_0, B
end procedure

3.3.2 Code

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Load the dataset
df = pd.read_csv('housing.csv')

# Split the dataset into training and testing sets
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train the MLR model
mlr = LinearRegression()
mlr.fit(X_train, y_train)

# Predict the target values of the testing set
y_pred = mlr.predict(X_test)

# Evaluate the performance of the model
r2 = r2_score(y_test, y_pred)
print('R^2:', r2)
```

Listing 3.3: Multiple Linear Regression

```
R^2: 0.6910934003098523
```

3.4 Ridge Regression

Ridge Regression is a type of linear regression used to prevent overfitting by adding a penalty term to the sum of squared residuals. This penalty term is the L2 norm of the coefficients multiplied by a hyper parameter lambda. It results in a more generalized model that is less likely to overfit the training data, making it useful for high-dimensional datasets with multicollinearity.

3.4.1 Pseudocode

procedure RIDGEREGRESSION(X, Y, λ)
    $n \leftarrow$ number of training examples
    $m \leftarrow$ number of features in X
    $I \leftarrow$ identity matrix of size $m \times m$
    $w \leftarrow (X^T X + \lambda I)^{-1} X^T Y$
    return w
end procedure

3.4.2 Code

```
import numpy as np
import matplotlib.pyplot as mpt
import pandas as pd

data_set= pd.read_csv('housing.csv')

#Extracting dependent and independent variables
x= np.array(data_set.iloc[:,:-1].values)
y= np.array(data_set.iloc[:,13].values)

#Splitting of data
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.2, random_state
    =0)

lambda_ridge=0.5
# Bˆ computation:
x_trans=x.transpose()
res1=(np.dot(x_trans,x))+(lambda_ridge*np.identity(13))
res2=np.dot(x_trans,y)
B_cap=np.dot((np.linalg.inv(res1)),res2)

#Ridge Regression
beta_0=0.0
```

```
for i in range (len(y_train)):
    beta_0+=y_train[i]
beta_0/=len(y_train)

B_cap_tilde=(np.matrix(np.concatenate(([beta_0],B_cap), axis=None))).transpose()

y_predicted= []
for each in range(len(x_test)):
    x_test_tilde= (np.matrix(np.concatenate(([1],x_test[each]), axis=None))).
    transpose()
    #print(x_test_tilde)
    y_predicted.append(np.dot(B_cap_tilde.transpose(),x_test_tilde).tolist()[0])
y_predicted= np.array(y_predicted).flatten()
print(y_predicted)
```

Listing 3.4: Ridge Regression

3.5 LASSO Regression

Lasso regression is a regularization technique. It is used over regression methods for a more accurate prediction. This model uses shrinkage. Shrinkage is where data values are shrunk towards a central point as the mean. The lasso procedure encourages simple, sparse models (i.e. models with fewer parameters).

3.5.1 Pseudocode

procedure LASSO($X, y, \lambda, \alpha, max_iter$)
    $n, p \leftarrow$ dimensions of X
    $w \leftarrow$ array of size p initialized with zeros
    $intercept \leftarrow$ mean of y
    $X \leftarrow$ normalized version of X
    $X^T \leftarrow$ transpose of X
    $t \leftarrow 0$
    while $t < max_iter$ **do**
        $t \leftarrow t + 1$
        for $j \leftarrow 1$ to p **do**
            $X_j \leftarrow$ column j of X
            $R_j \leftarrow y - intercept - X^T \cdot w + w_j \cdot X_j$
            $z_j \leftarrow X_j^T \cdot R_j / n + w_j$
            $w_j \leftarrow$ soft threshold function$(z_j, \alpha \cdot \lambda)$

end for
end while
return $w, intercept$
end procedure

3.5.2 Code

```
import numpy as np

class LassoRegression:
    def __init__(self, alpha=1, max_iterations=1000, tolerance=0.001):
        self.alpha = alpha
        self.max_iterations = max_iterations
        self.tolerance = tolerance
        self.theta = None

    def fit(self, X_train, y_train):
        m, n = X_train.shape
        X_train = np.hstack((np.ones((m, 1)), X_train))
        y_train = y_train.reshape(-1, 1)
        self.theta = np.zeros((n+1, 1))

        for iteration in range(self.max_iterations):
            old_theta = self.theta.copy()
            for j in range(n+1):
                if j == 0:
                    self.theta[j] = np.mean(y_train - np.dot(X_train[:, 1:], self.
    theta[1:]))
                else:
                    X_j = X_train[:, j]
                    y_hat = np.dot(X_train, self.theta)
                    rho = X_j.T.dot(y_train - y_hat + self.theta[j]*X_j)
                    if rho < -self.alpha/2:
                        self.theta[j] = (rho + self.alpha/2) / (X_j**2).sum()
                    elif rho > self.alpha/2:
                        self.theta[j] = (rho - self.alpha/2) / (X_j**2).sum()
                    else:
                        self.theta[j] = 0
            if np.sum(np.abs(self.theta - old_theta)) < self.tolerance:
                break

    def predict(self, X_test):
        m, n = X_test.shape
        X_test = np.hstack((np.ones((m, 1)), X_test))
        predictions = np.dot(X_test, self.theta)
```

```
38         return predictions.ravel()
```

Listing 3.5: Lasso Regression

3.6 Logistic Regression

Logistic regression is an example of supervised learning. It is used to calculate or predict the probability of a binary (yes/no) event occurring.

3.6.1 Pseudocode

1: **procedure** LogisticRegression(X, Y, α, ϵ)
2: $\theta \leftarrow$ array of size m, where m is the number of features in X
3: *cost* $\leftarrow$ initial cost value
4: **while** *cost* $> \epsilon$ **do**
5: $h \leftarrow$ sigmoid function applied to $\theta^T X$
6: $J \leftarrow$ cost function of logistic regression
7: $\theta \leftarrow \theta - \alpha \frac{\partial J}{\partial \theta}$
8: *cost* $\leftarrow$ new cost value
9: **end while**
10: **return** θ
11: **end procedure**

3.6.2 Code

```
1   import pandas as pd
2   from sklearn.model_selection import train_test_split
3   from sklearn.linear_model import LogisticRegression
4   from sklearn.metrics import accuracy_score
5
6   # Load the dataset
7   df = pd.read_csv('housing.csv')
8
9   # Split the dataset into training and testing sets
10  X = df.iloc[:, :-1].values
11  y = df.iloc[:, -1].values
12  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
13
14  # Train the Logistic Regression model
15  logreg = LogisticRegression()
```

```
16    logreg.fit(X_train, y_train)
17
18    # Predict the target values of the testing set
19    y_pred = logreg.predict(X_test)
20
21    # Evaluate the performance of the model
22    accuracy = accuracy_score(y_test, y_pred)
23    print('Accuracy:', accuracy)
```

Listing 3.6: Logistic Regression

```
Accuracy: 0.02040816326530612
```

3.7 Support Vector Machine - Classification

Support Vector Machine (SVM) is a popular machine learning algorithm used for classification, regression, and outlier detection. The main idea of SVM is to find the optimal hyperplane that separates two classes in a high-dimensional space. SVM seeks to maximize the margin between the hyperplane and the closest data points from each class, which are called support vectors. The optimal hyperplane is the one that maximizes the margin while correctly classifying the training data.

In classification problems, SVM can handle both linearly separable and non-linearly separable data by using a kernel function to map the data to a higher-dimensional space where it is more likely to be linearly separable. Common kernel functions include linear, polynomial, and radial basis function (RBF) kernels.

3.7.1 Pseudocode

procedure SVMClassification($X, Y, C, kernel, x_{test}$)
  $n \leftarrow$ number of training examples
  $m \leftarrow$ number of features
  $\alpha \leftarrow$ array of size n, initialized to 0
  $b \leftarrow 0$
  $E \leftarrow$ array of size n, initialized to 0
  $K \leftarrow$ matrix of size $n \times n$, where $K_{i,j} = kernel(X_i, X_j)$
  repeat
    for $i \leftarrow 1$ to n **do**
      $E_i \leftarrow (\sum_{j=1}^{n} \alpha_j Y_j K_{i,j}) + b - Y_i$
      $E_i \leftarrow \max(0, E_i)$
      $E_i \leftarrow \min(C, E_i)$

end for
select i, j that maximize $|E_i - E_j|$
$L \leftarrow \max(0, \alpha_j - \alpha_i)$
$H \leftarrow \min(C, C + \alpha_j - \alpha_i)$
$\eta \leftarrow 2K_{i,j} - K_{i,i} - K_{j,j}$
$\alpha_j \leftarrow \alpha_j + \frac{Y_j(E_i - E_j)}{\eta}$
$\alpha_j \leftarrow \max(L, \alpha_j)$
$\alpha_j \leftarrow \min(H, \alpha_j)$
$\alpha_i \leftarrow \alpha_i + Y_i Y_j (\alpha_j - \alpha_{j,old})$
$b_1 \leftarrow b - E_i - Y_i(\alpha_i - \alpha_{i,old})K_{i,i} - Y_j(\alpha_j - \alpha_{j,old})K_{i,j}$
$b_2 \leftarrow b - E_j - Y_i(\alpha_i - \alpha_{i,old})K_{i,j} - Y_j(\alpha_j - \alpha_{j,old})K_{j,j}$
if $0 < \alpha_i < C$ **then**
$b \leftarrow b_1$
else if $0 < \alpha_j < C$ **then**
$b \leftarrow b_2$
else
$b \leftarrow \frac{b_1 + b_2}{2}$
end if
until convergence or maximum number of iterations reached
$y_{pred} \leftarrow \text{sign}(\sum_{i=1}^{n} \alpha_i Y_i kernel(X_i, x_{test}) - b)$
return y_{pred}
end procedure

3.7.2 Code

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load the dataset
df = pd.read_csv('housing.csv')

# Split the dataset into training and testing sets
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train the SVM Classification model
svm = SVC(C=1.0, kernel='rbf', gamma='scale')
```

```
16    svm.fit(X_train, y_train)
17
18    # Predict the target values of the testing set
19    y_pred = svm.predict(X_test)
20
21    # Evaluate the performance of the model
22    accuracy = accuracy_score(y_test, y_pred)
23    print('Accuracy:', accuracy)
```

Listing 3.7: Support Vector Machine - Classification

```
Accuracy: 0.02040816326530612
```

3.8 Support Vector Machine - Regression

In regression problems, SVM aims to find the hyperplane that best fits the data while minimizing the margin violations. The prediction for a new data point is based on its distance to the hyperplane. SVM has proven to be effective in a variety of applications, such as image classification, text classification, and bioinformatics. However, it can be computationally expensive for large datasets and can be sensitive to the choice of hyperparameters.

3.8.1 Pseudocode

procedure SVMREGRESSION($X, y, C, \epsilon, kernel$)
  Initialize $\alpha_1, \alpha_2, ..., \alpha_n$, b, and $K_{i,j}$
  $iter \leftarrow 0$
  while $iter < max_iter$ **do**
    $\hat{y} \leftarrow \sum_{i=1}^{n} \alpha_i y_i K(x_i, x_j) + b$
    $E_i \leftarrow \hat{y}i - y_i$
    if $\alpha_i < C$ **then**
      $\frac{\partial L}{\partial \alpha_i} \leftarrow -y_i E_i$
    else if $\alpha_i = C$ **then**
      $\frac{\partial L}{\partial \alpha_i} \leftarrow 0$
    else if $\alpha_i > 0$ **then**
      $\frac{\partial L}{\partial \alpha_i} \leftarrow -y_i E_i$
    end if
    $\frac{\partial L}{\partial b} \leftarrow -\sum i = 1^n y_i E_i$
    $\alpha_i \leftarrow \alpha_i - \eta \frac{\partial L}{\partial \alpha_i}$

$b \leftarrow b - \eta \frac{\partial L}{\partial b}$
Clip α_i so that $0 \leq \alpha_i \leq C$
$iter \leftarrow iter + 1$
end while
return α, b
end procedure

3.8.2 Code

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error

# Load the dataset
df = pd.read_csv('housing.csv')

# Split the dataset into training and testing sets
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train the SVM Regression model
svm = SVR(C=1.0, kernel='rbf', gamma='scale')
svm.fit(X_train, y_train)

# Predict the target values of the testing set
y_pred = svm.predict(X_test)

# Evaluate the performance of the model
mse = mean_squared_error(y_test, y_pred)
print('MSE:', mse)
```

Listing 3.8: Support Vector Machine - Regression

```
MSE: 22382660666.39476
```

3.9 Decision Tree for Classification

The algorithm builds a tree-like model of decisions and their possible consequences based on the features of the input data. The tree consists of internal nodes that represent tests on the input features, branches that correspond to the possible outcomes of these tests, and leaf nodes that represent the

final classification or regression prediction.

In classification tasks, the decision tree algorithm partitions the input data into smaller subsets based on the values of the input features. The algorithm selects the feature that best separates the data into the different classes, and creates a split at that feature value. The process is repeated recursively for each resulting subset until a stopping criterion is met, such as a minimum number of samples at each leaf node. The leaf nodes represent the class labels for the input data, and the decision tree can be used to predict the class label of new data.

3.9.1 Pseudocode

procedure BUILDDECISIONTREE(X, Y)
  if all examples in Y belong to the same class c **then**
    return a leaf node with label c
  end if
  if there are no more features to split on **then**
    return a leaf node with the majority class label in Y
  end if
  $j_{best} \leftarrow$ the feature that provides the highest information gain
  Create a new decision node that splits on j_{best}
  for each possible value v of j_{best} **do**
    Create a new subtree for the examples with $j_{best} = v$
    Recursively apply BuildDecisionTree to the new subtree, with the remaining features and examples
  end for
  return the decision tree
end procedure

3.9.2 Code

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load the dataset
df = pd.read_csv('housing.csv')

# Split the dataset into training and testing sets
X = df.iloc[:, :-1].values
```

```
    y = df.iloc[:, -1].values
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

    # Train the Decision Tree Classifier
    dtc = DecisionTreeClassifier(random_state=42)
    dtc.fit(X_train, y_train)

    # Predict the target values of the testing set
    y_pred = dtc.predict(X_test)

    # Evaluate the performance of the model
    accuracy = accuracy_score(y_test, y_pred)
    print('Accuracy:', accuracy)
```

Listing 3.9: Decision Tree for Classification

```
Accuracy: 0.01020408163265306
```

3.10 Decision Tree for Regression

In regression tasks, the decision tree algorithm works similarly, but instead of predicting a class label, it predicts a continuous value. The algorithm partitions the input data based on the values of the input features, and selects the feature that minimizes the variance of the target variable within each subset. The process is repeated recursively for each resulting subset until a stopping criterion is met, such as a minimum number of samples at each leaf node. The leaf nodes represent the predicted value for the input data, and the decision tree can be used to predict the continuous value of new data.

3.10.1 Pseudocode

procedure DECISION TREE REGRESSION(X, Y)
  if stopping criteria is met **then**
    return mean of Y
  else
    Choose the best feature f and threshold t
    Split the data into two subsets: X_L, Y_L and X_R, Y_R, where X_L contains the examples where feature f is less than or equal to t and X_R contains the examples where feature f is greater than t
    Create a decision node that tests feature f against threshold t
    Recursively build the left and right subtrees using X_L, Y_L and X_R, Y_R, respectively

9: **end if**
10: **end procedure**

3.10.2 Code

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# Load the dataset
df = pd.read_csv('housing.csv')

# Split the dataset into training and testing sets
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train the Decision Tree Regressor
dtr = DecisionTreeRegressor(random_state=42)
dtr.fit(X_train, y_train)

# Predict the target values of the testing set
y_pred = dtr.predict(X_test)

# Evaluate the performance of the model
mse = mean_squared_error(y_test, y_pred)
print('Mean Squared Error:', mse)
```

Listing 3.10: Decision Tree Regression

```
Mean Squared Error: 6396165000.0
```

3.11 Naive Bayes' Classifier

Naive Bayes Classifier is a popular probabilistic algorithm used for classification tasks in machine learning. It is based on the Bayes' theorem and the assumption of conditional independence among the features. In a Naive Bayes Classifier, each instance is represented as a vector of feature values, and the goal is to predict the class label of the instance based on its feature values. The algorithm starts by estimating the prior probability of each class label, i.e., the probability that an instance belongs to each class based on the training data. Then, it calculates the conditional probability of each feature given each class label, i.e., the probability of observing each feature value given that

the instance belongs to a particular class. These probabilities are estimated from the training data using Maximum Likelihood Estimation (MLE) or Bayesian estimation methods.

3.11.1 pseudocode

procedure NaiveBayes(X, Y, x_{test})
    $P(Y = y_i) \leftarrow$ frequency of y_i in Y
    for each feature x_j in X **do**
        $P(x_j|Y = y_i) \leftarrow$ probability distribution of x_j in examples with class y_i
    end for
    $P(x_{test}|Y = y_i) \leftarrow$ product of $P(x_j|Y = y_i)$ for all features x_j in x_{test}
    $P(Y = y_i|x_{test}) \leftarrow \frac{P(Y=y_i) \times P(x_{test}|Y=y_i)}{\sum_j P(Y=y_j) \times P(x_{test}|Y=y_j)}$
    return class with highest $P(Y = y_i|x_{test})$
end procedure

3.11.2 Code

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# Load the dataset
df = pd.read_csv('housing.csv')

# Split the dataset into training and testing sets
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train the Decision Tree Regressor
dtr = DecisionTreeRegressor(random_state=42)
dtr.fit(X_train, y_train)

# Predict the target values of the testing set
y_pred = dtr.predict(X_test)

# Evaluate the performance of the model
mse = mean_squared_error(y_test, y_pred)
print('Mean Squared Error:', mse)
```

Listing 3.11: Naive Baye's Classifier

```
Mean Squared Error: 6396165000.0
```

3.12 Random Forest

Random Forest is a supervised machine learning algorithm that is used for both classification and regression tasks. It is an ensemble learning method that combines the predictions of multiple decision trees, each trained on a randomly sampled subset of the training data, to produce a final prediction. Random Forest works by creating a forest of decision trees, where each tree is trained on a random subset of the training data and a random subset of the features. This randomness helps to reduce overfitting and improve the accuracy of the predictions.

3.12.1 Pseudocode

procedure RandomForest(X, Y, n_{trees}, m_{try})
    for $t \leftarrow 1$ to n_{trees} **do**
        $X_t, Y_t \leftarrow$ bootstrap sample of X and Y (with replacement)
        $T_t \leftarrow$ decision tree trained on X_t, Y_t, using only m_{try} randomly chosen features at each split
    end for
    return $T_1, T_2, ..., T_{n_{trees}}$
end procedure
procedure RandomForestPredict($X_{test}, T_1, T_2, ..., T_{n_{trees}}$)
    $n_{trees} \leftarrow$ length of $T_1, T_2, ..., T_{n_{trees}}$
    $Y_{pred} \leftarrow$ array of size $n_{test} \times n_{trees}$
    for $t \leftarrow 1$ to n_{trees} **do**
        $Y_{pred,:,t} \leftarrow$ predictions of T_t on X_{test}
    end for
    $y_{pred} \leftarrow$ mean of Y_{pred} over axis 1
    return y_{pred}
end procedure

3.12.2 Code

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

```

```
    # Load the dataset
    df = pd.read_csv('housing.csv')

    # Split the dataset into training and testing sets
    X = df.iloc[:, :-1].values
    y = df.iloc[:, -1].values
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

    # Train the Random Forest Regressor
    rfr = RandomForestRegressor(n_estimators=100, random_state=42)
    rfr.fit(X_train, y_train)

    # Predict the target values of the testing set
    y_pred = rfr.predict(X_test)

    # Evaluate the performance of the model
    mse = mean_squared_error(y_test, y_pred)
    print('Mean Squared Error:', mse)
```

Listing 3.12: Random Forest

```
Mean Squared Error: 3245169433.5
```

3.13 K-Means Clustering

K-means clustering is a popular unsupervised machine learning algorithm that partitions a set of data points into K clusters, where K is a pre-specified number. The algorithm iteratively assigns each data point to the nearest centroid, and then updates the centroids based on the mean of the assigned points. The process repeats until the centroids converge, or a maximum number of iterations is reached.

The K-means algorithm can be used for a variety of tasks, such as customer segmentation, image compression, anomaly detection, and more. It is a simple and effective method for clustering data, but it has some limitations. For example, it is sensitive to the initial centroid positions and can get stuck in local minima. Variants of the K-means algorithm have been proposed to address some of these issues, such as K-means++, which uses a smarter initialization method, and K-medoids, which uses representative examples as the centroids instead of the mean.

3.13.1 Pseudocode

1: **procedure** KMEANS(X, K, max_iter)

Initialize K centroids, $\mu_1, \mu_2, ..., \mu_K$
$iter \leftarrow 0$
while $iter < max_iter$ **do**
    Assign each example in X to the nearest centroid:
    for $i \leftarrow 1$ to n **do**
        $c_i \leftarrow$ index of the nearest centroid to X_i
    end for
    Update the centroids:
    for $j \leftarrow 1$ to K **do**
        $\mu_j \leftarrow$ mean of the examples assigned to centroid j
    end for
    $iter \leftarrow iter + 1$
end while
return $c_1, c_2, ..., c_n$
end procedure

3.13.2 Code

```
import pandas as pd
from sklearn.cluster import KMeans

# Load the data
df = pd.read_csv('housing.csv')

# Drop the target column if it exists
if 'target' in df.columns:
    df.drop('target', axis=1, inplace=True)

# Scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(df)

# Create the KMeans object and fit it to the data
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)

# Print the cluster centers
print(kmeans.cluster_centers_)
```

Listing 3.13: K-means Clustering

```
[[ 1.38550703 -1.04546153 -1.02750859  1.5893341 ]
```

```
[-0.59644047  1.19562411  0.67657666 -0.9795906 ]
[-0.14084338 -0.32691964 -0.03165417  0.01007543]]
```

3.14 Spectral Clustering

Spectral clustering is a popular clustering algorithm that uses the eigenvalues and eigenvectors of a similarity matrix to group data points into clusters. It is based on the spectral graph theory, which uses the graph Laplacian to represent the pairwise similarity between data points.

The algorithm works by first constructing a similarity matrix from the input data, such as a Gaussian kernel matrix or a k-nearest neighbor graph. Then, it computes the graph Laplacian of the similarity matrix, which captures the smoothness and connectivity of the graph. Finally, it performs spectral decomposition of the Laplacian to obtain the eigenvectors and eigenvalues, which are used to cluster the data points.

To cluster the data points, the algorithm first selects the first k eigenvectors corresponding to the smallest eigenvalues, where k is the number of clusters desired. Then, it constructs a new matrix by stacking these eigenvectors as columns and performs k-means clustering on this new matrix to assign each data point to a cluster.

Spectral clustering has several advantages over other clustering algorithms. It can handle complex geometric structures and non-convex clusters, and is less sensitive to noise and outliers. It also provides a flexible way to select the number of clusters based on the eigenvalues of the Laplacian. However, it can be computationally expensive for large datasets and requires tuning of the similarity matrix and number of clusters.

Spectral clustering has been used in various applications, including image segmentation, community detection in social networks, and gene expression analysis. It is considered a powerful tool for unsupervised learning in machine learning.

3.14.1 Pseudocode

procedure SpectralClustering(X, k, σ)
    Compute the similarity matrix W using RBF kernel with parameter σ
    Compute the diagonal matrix D where $D_{ii} = \sum_j W_{ij}$
    Compute the Laplacian matrix $L = D - W$
    Compute the first k eigenvectors of L and form a matrix $V \in R^{n \times k}$ where each row is an eigenvector
    Normalize each row of V to have unit norm
    Cluster the rows of V using any clustering algorithm to obtain clusters $C_1, C_2, ..., C_k$
    return $C_1, C_2, ..., C_k$
end procedure

3.14.2 Code

```python
import pandas as pd
from sklearn.cluster import SpectralClustering

# Load the data
df = pd.read_csv('housing.csv')

# Drop the target column if it exists
if 'target' in df.columns:
    df.drop('target', axis=1, inplace=True)

# Scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(df)

# Create the SpectralClustering object and fit it to the data
spectral = SpectralClustering(n_clusters=3)
spectral.fit(X)

# Print the cluster labels
print(spectral.labels_)
```

Listing 3.14: Spectral Clustering

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2
 2 0 2 0 0 2 2 2 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 2 0
 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2 2 2 2 2 2 2 2 0 2 2
 0 0 0 0 2 0 0 0 0 0 0 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0]
```

3.15 MDP: Markov Decision Process

Markov Decision Process (MDP) is a mathematical framework used for decision-making problems where outcomes are uncertain and influenced by actions taken by the decision maker. It involves an agent interacting with an environment by taking actions, receiving rewards or penalties, and transitioning to a new state.

The agent's objective is to find a policy that maximizes the expected sum of rewards over time, called the expected return. This can be achieved by using dynamic programming algorithms such as value iteration or policy iteration, which iteratively compute the optimal value function or policy based on the MDP model.

MDP has numerous applications in various fields, including robotics, finance, healthcare, and energy management. It can be used to plan the trajectory of a robot, optimize portfolio allocation or trading strategies, design personalized treatment plans for patients, and optimize the control of energy systems.

3.16 MRP: Markov Reward Process

The Markov Reward Process (MRP) is a mathematical framework used to analyze the development of a system over time, where the system moves between various states and yields rewards at each state. While resembling a Markov Decision Process (MDP), the MRP does not require any actions or decisions made by an agent.

An MRP is defined by a tuple (S, P, R, γ), where:

1. S is a finite set of states. 2. P is a state transition probability matrix, where $P_{s,s'}$ represents the probability of transitioning from state s to state s'. 3. R is a reward function, where R_s represents the expected reward received upon entering state s. 4. γ is a discount factor, where $0 \leq \gamma \leq 1$, used to discount future rewards. An MRP is a mathematical model used to study the evolution of a system over time, where the system transitions between different states and generates rewards at each state. It is similar to a Markov Decision Process (MDP), but without any actions or decisions by an agent. The MRP can be represented by a Markov chain, where each state generates a reward that is accumulated over time. The expected sum of discounted rewards obtained by starting in a given state and following a policy is called the value function, which can be computed using iterative algorithms such as value iteration or policy iteration, similar to those used in MDPs.

MRPs have various applications in finance, healthcare, and engineering. In finance, they can be used to model the evolution of stock prices, where each state represents the price of the stock at a given time, and the reward represents the profit or loss obtained by holding the stock. In healthcare, they can be used to model patient outcomes, where each state represents the health status of the patient, and the reward represents the quality of life. In engineering, they can be used to model the reliability of a system, where each state represents the state of the system, and the reward represents

the cost or benefit of being in that state.

3.17 Queue - Learning

Q-learning is a popular reinforcement learning algorithm used to learn the optimal policy in a Markov Decision Process (MDP) or a similar environment without any prior knowledge of the environment. It is a model-free algorithm, meaning that it doesn't require a model of the environment and instead learns by interacting with it.

The Q-function (action-value function) is at the heart of Q-learning, and it estimates the expected sum of discounted rewards that an agent can obtain by taking a particular action in a given state. The Q-function maps a state-action pair to an expected reward value, which is updated based on the rewards received and the expected rewards of the subsequent state-action pairs. The optimal policy is then obtained by selecting the action with the highest expected reward value at each state.

Q-learning is well-suited for environments with large state spaces and action spaces and has been used in various real-world applications such as robotics, game playing, and control systems. However, the algorithm has a tendency to overestimate the Q-values, which can result in suboptimal policies. To mitigate this issue, several techniques such as Double Q-learning, Prioritized Experience Replay, and Dueling Networks have been developed.

3.18 SARSA - Learning

SARSA is a popular reinforcement learning algorithm used to find an optimal policy in a Markov Decision Process (MDP). It is similar to Q-learning, but it updates Q-values based on the current state, action taken, reward received, and the next state-action pair. SARSA is an on-policy algorithm, which means it learns the value of the policy used to select actions. Its flexibility in learning both deterministic and stochastic policies makes it useful for various applications.

Unlike Q-learning, SARSA updates Q-values based on the current policy. This characteristic allows it to converge to a more stable policy, but it may also converge to a suboptimal policy if the initial policy is not optimal. SARSA has been used to solve real-world problems such as robot navigation, game playing, and control systems. It can handle high-dimensional state and action spaces by using function approximation techniques such as neural networks.

In conclusion, SARSA is a powerful reinforcement learning algorithm that has a wide range of applications. Its ability to learn both deterministic and stochastic policies, on-policy nature, and compatibility with function approximation techniques make it a valuable tool for solving complex problems.

Chapter 4

Proposed Model

4.1 Background

The use of machine learning in medical research has seen rapid progress in recent years, with many applications in the field of disease detection and diagnosis. One area where machine learning is being increasingly utilized is in the detection of eye diseases.

Several types of eye diseases can cause vision loss and blindness, including age-related macular degeneration (AMD), diabetic retinopathy (DR), and glaucoma. Early detection and diagnosis of these diseases can be critical in preventing permanent vision loss.

Machine learning algorithms can be trained on large datasets of images of the retina and other parts of the eye to learn how to identify the signs of various eye diseases accurately. This can help doctors to detect and diagnose eye diseases much earlier than with traditional methods, such as manual examination of the retina.

One of the most promising applications of machine learning in eye disease detection is the use of deep learning algorithms, which can analyze large amounts of data and identify patterns that may be difficult or impossible for humans to detect. This approach has been used successfully to detect and diagnose various eye diseases, including AMD, DR, and glaucoma.

In addition to detection and diagnosis, machine learning is also being used to predict disease progression and outcomes, which can help doctors to develop more effective treatment plans for their patients. For example, machine learning algorithms can analyze medical data from patients with a particular eye disease and use this information to predict the likelihood of disease progression and the best course of treatment.

Overall, the use of machine learning in eye disease detection and diagnosis has the potential to revolutionize the way we approach the treatment of these conditions, allowing for earlier detection and more effective treatment, ultimately leading to improved patient outcomes.

4.2 Investigating the Limitation

While machine learning has shown promise in detecting eye diseases, there are also some limitations that need to be considered. One limitation is the availability and quality of data. The accuracy of the algorithm is heavily dependent on the quality and quantity of the data used for training. Inadequate or biased data can lead to inaccurate or unreliable results. Therefore, it is crucial to ensure that the data used for training is representative of the population and of high quality.

Another limitation is the lack of interpretability of some machine learning models. Some models may be complex and difficult to understand, making it challenging for clinicians to interpret the results and trust the algorithm's recommendations. This can be particularly problematic in healthcare, where decision-making has significant consequences for patients. Therefore, it is important to develop models that are interpretable and understandable for clinicians.

Furthermore, machine learning models are not perfect and can make mistakes. False positives and false negatives can occur, leading to missed diagnoses or unnecessary treatments, respectively. This can have negative consequences for patients, including delays in treatment or exposure to unnecessary risks. Therefore, it is crucial to validate the accuracy of machine learning models and to always involve a human expert in the decision-making process.

In conclusion, while machine learning has shown potential in detecting eye diseases, it is important to consider the limitations of the technology. Adequate and representative data, interpretable models, and validation of accuracy are crucial for the successful implementation of machine learning in clinical settings.

4.3 Dataset Selected

Glaucoma Detection
(OCT Scans Retinal Imaging)
This data set contains images/oct scans of the eye.
Glaucoma is a group of eye conditions that damage the optic nerve, the health of which is vital for good vision. This damage is often caused by an abnormally high pressure in your eye. Glaucoma is one of the leading causes of blindness for people over the age of 60.

4.4 Pseudocode :Design of the Model to overcome the limitaion

4.4.1 Pseudocode for KNN - C

1: **procedure** KNNClassification(X, Y, x_{test}, k)

$D \leftarrow$ array of size n, where n is the number of training examples
for $i \leftarrow 1$ to n **do**
$D_i \leftarrow$ distance between x_{test} and X_i
end for
$indices \leftarrow$ indices of the k smallest values in D
$Y_{neighbors} \leftarrow$ classes of the k nearest neighbors, based on their indices in Y
$y_{pred} \leftarrow$ class with the highest frequency in $Y_{neighbors}$
return y_{pred}
end procedure

4.4.2 Pseudocode for SVM - C

procedure SVMClassification($X, Y, C, kernel, x_{test}$)
$n \leftarrow$ number of training examples
$m \leftarrow$ number of features
$\alpha \leftarrow$ array of size n, initialized to 0
$b \leftarrow 0$
$E \leftarrow$ array of size n, initialized to 0
$K \leftarrow$ matrix of size $n \times n$, where $K_{i,j} = kernel(X_i, X_j)$
repeat
for $i \leftarrow 1$ to n **do**
$E_i \leftarrow (\sum_{j=1}^{n} \alpha_j Y_j K_{i,j}) + b - Y_i$
$E_i \leftarrow \max(0, E_i)$
$E_i \leftarrow \min(C, E_i)$
end for
select i, j that maximize $|E_i - E_j|$
$L \leftarrow \max(0, \alpha_j - \alpha_i)$
$H \leftarrow \min(C, C + \alpha_j - \alpha_i)$
$\eta \leftarrow 2K_{i,j} - K_{i,i} - K_{j,j}$
$\alpha_j \leftarrow \alpha_j + \frac{Y_j(E_i - E_j)}{\eta}$
$\alpha_j \leftarrow \max(L, \alpha_j)$
$\alpha_j \leftarrow \min(H, \alpha_j)$
$\alpha_i \leftarrow \alpha_i + Y_i Y_j (\alpha_j - \alpha_{j,old})$
$b_1 \leftarrow b - E_i - Y_i(\alpha_i - \alpha_{i,old})K_{i,i} - Y_j(\alpha_j - \alpha_{j,old})K_{i,j}$
$b_2 \leftarrow b - E_j - Y_i(\alpha_i - \alpha_{i,old})K_{i,j} - Y_j(\alpha_j - \alpha_{j,old})K_{j,j}$
if $0 < \alpha_i < C$ **then**
$b \leftarrow b_1$
else if $0 < \alpha_j < C$ **then**

$b \leftarrow b_2$
else
$b \leftarrow \frac{b_1+b_2}{2}$
end if
until convergence or maximum number of iterations reached
$y_{pred} \leftarrow \text{sign}(\sum_{i=1}^{n} \alpha_i Y_i kernel(X_i, x_{test}) - b)$
return y_{pred}
end procedure

4.5 Program: Functional Part

4.5.1 KNN C

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import f1_score,confusion_matrix
from sklearn.model_selection import train_test_split

data = pd.read_csv ("/Users/user/Desktop/DATASET/New folder/GLAUCOMA-DETECTION-master
    /KNN/extracted_features.csv")
X , Y = data.values [ : , : -1 ] , data.values [ : , -1 ]
X.shape
Y.shape
X_train, X_test, Y_train, Y_test = train_test_split( X, Y , test_size = 0.3 ,
    random_state = 7)
model=KNeighborsClassifier(n_neighbors=7)
model.fit(X_train,Y_train)
print ("baseline accuracy " , (  (model.predict(X_test) == Y_test).astype("int8").sum
    () / Y_test.shape[0]   ) )
```

```
baseline accuracy: 0.75
```

```
    from sklearn import metrics
    confusion_matrix = metrics.confusion_matrix(Y_test, y_pred)
    cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix,
    display_labels = [False, True])
    cm_display.plot()
    plt.show()
```

```
    print("Sensitivity:",a[0][0]/(a[0][0]+a[0][1]))
```

```
    Sensitivity: 0.9204545454545454
```

```
print("Specificty:", a[1][1]/(a[1][1]+a[1][0]))
```

```
    Specificty: 0.3854166666666667
```

```
    df = pd.DataFrame({'lab':['Accuracy', 'Sensitivity', 'Specificty'], 'val'
    :[73.1,92,38.5]})
    ax = df.plot.bar(x='lab', y='val', rot=0,color=['b', 'r', 'g'])
```

4.5.2 SVM C

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import f1_score,confusion_matrix
from sklearn.model_selection import train_test_split
data = pd.read_csv ("/Users/user/Desktop/DATASET/New folder/GLAUCOMA-DETECTION-master
    /KNN/extracted_features.csv")
X , Y = data.values [ : , : -1 ] , data.values [ : , -1 ]
X.shape
Y.shape
X_train, X_test, Y_train, Y_test = train_test_split( X, Y , test_size = 0.3 ,
    random_state = 7)
model_svm=svm.SVC(gamma='scale')
model_svm.fit(X_train,Y_train)
print ("Accuracy " , (  (model_svm.predict(X_test) == Y_test).astype("int8").sum() /
    Y_test.shape[0]   ) )
```

```
    Accuracy  0.75
```

```
    confusion_matrix = metrics.confusion_matrix(Y_test, y_pred)
    cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix,
    display_labels = [False, True])
    cm_display.plot()
    plt.show()
```

```
    print("Sensitivity:",a[0][0]/(a[0][0]+a[0][1]))
```

```
    Sensitivity: 0.9204545454545454
```

```
print("Specificty:", a[1][1]/(a[1][1]+a[1][0]))
```

```
    Specificty: 0.3854166666666667
```

```
    df = pd.DataFrame({'lab':['Accuracy', 'Sensitivity', 'Specificty'], 'val'
    :[73.1,92,38.5]})
    ax = df.plot.bar(x='lab', y='val', rot=0,color=['b', 'r', 'g'])
```

4.5.3 RANDOM FOREST

```
    import numpy as np
    import matplotlib.pyplot as plt
    import pandas as pd
    import seaborn as sns
    from sklearn import svm
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.neighbors import KNeighborsClassifier
    from sklearn.metrics import f1_score,confusion_matrix
    from sklearn.model_selection import train_test_split
    from sklearn.ensemble import RandomForestClassifier
    data = pd.read_csv ("/Users/user/Desktop/DATASET/New folder/GLAUCOMA-DETECTION-
    master/KNN/extracted_features.csv")
    X , Y = data.values [ : , : -1 ] , data.values [ : , -1 ]
    X.shape
    Y.shape
    X_train, X_test, Y_train, Y_test = train_test_split( X, Y , test_size = 0.3 ,
    random_state = 7)
    model_rf = RandomForestClassifier( n_estimators=100 , random_state = 7 )
    model_rf.fit(X_train,Y_train)
    print ("Accuracy " , (  (model_rf.predict(X_test) == Y_test).astype("int8").sum()
     / Y_test.shape[0]   ) )

```

```
    Accuracy  0.7757352941176471
```

```
    from sklearn.metrics import confusion_matrix
    y_pred = model_rf.predict(X_test)
    confusion_matrix(Y_test,y_pred)
    confusion_matrix = metrics.confusion_matrix(Y_test, y_pred)
    cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix,
    display_labels = [False, True])
    cm_display.plot()
    plt.show()
    print("Sensitivity:",a[0][0]/(a[0][0]+a[0][1]))

```

```
Sensitivity: 0.9204545454545454
```

```
            print("Specificty:", a[1][1]/(a[1][1]+a[1][0]))

```

```
Specificty: 0.3854166666666667
```

```
        df = pd.DataFrame({'lab':['Accuracy', 'Sensitivity', 'Specificty'], 'val
    ':[73.1,92,38.5]})
        ax = df.plot.bar(x='lab', y='val', rot=0,color=['b', 'r', 'g'])

```

4.5.4 CNN using ResNet-50

4.5.5 Introduction

```
import numpy as np # linear algebra
import pandas as pd # data processing,

import os
for dirname, _, filenames in os.walk(r'C:\Users\sagar\Downloads\ML'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

Listing 4.1: Data Processing

4.5.6 Organizing Data

```
train_label = pd.read_csv(r'/content/drive/MyDrive/Colab Notebooks/ML/Glaucoma
    Detection/glaucoma.csv')
y_train = train_label['Glaucoma']
train_label.head()
```

Listing 4.2: Organizing Data

Filename	ExpCDR	Eye	Set	Glaucoma
001.jpg	0.7097	OD	A	0
002.jpg	0.6953	OS	A	0
003.jpg	0.9629	OS	A	0
004.jpg	0.7246	OD	A	0
005.jpg	0.6138	OS	A	0

Table 4.1: Eye illness dataset

4.5.7 Looking at the Data

```
from numpy import asarray

from PIL import Image
# load the image
image = Image.open(r'/content/drive/MyDrive/Colab Notebooks/ML/Glaucoma Detection/
    Fundus_Train_Val_Data/Fundus_Scanes_Sorted/Train/Glaucoma_Negative/394.jpg')
print(image.format)
print(image.mode)
print(image.size)
# show the image
image.show()
pixels = asarray(image)
```

Listing 4.3: Organizing Data

4.5.8 Implementation

```
# calculate global mean
mean = pixels.mean()
print('Mean: %.3f' % mean)
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
# global centering of pixels
pixels = pixels - mean

mean = pixels.mean()
print('Mean: %.3f' % mean)
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
```

```
print(pixels)


# pixel normalization
print('Data Type: %s' % pixels.dtype)
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
# convert from integers to floats
pixels = pixels.astype('float32')
# normalize to the range 0-1
pixels /= 255.0
mean = pixels.mean()
print('pixel mean = ', mean)

# confirm the normalization
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
print(pixels)
```

Listing 4.4: Normalization

```
#images after normalization
import matplotlib.pyplot as plt
fig, (ax0, ax1) = plt.subplots(1, 2)
ax0.imshow(image)
ax0.axis('off')
ax0.set_title('image')
ax1.imshow(pixels)
ax1.axis('off')
plt.show()
```

Listing 4.5: Visualization

```
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications.resnet50 import preprocess_input
```

```
TRAIN_DIR = r'/content/drive/MyDrive/Colab Notebooks/ML/Glaucoma Detection/
    Fundus_Train_Val_Data/Fundus_Scanes_Sorted/Train'

TEST_DIR = r'/content/drive/MyDrive/Colab Notebooks/ML/Glaucoma Detection/
    Fundus_Train_Val_Data/Fundus_Scanes_Sorted/Validation'
```

Listing 4.6: Training

```
from tensorflow.keras.applications.resnet50 import ResNet50,preprocess_input
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Dense, Activation, Flatten, Dropout
from keras.models import Sequential, Model
from keras.optimizers import SGD, Adam
from keras.callbacks import TensorBoard
```

```
import keras
import matplotlib.pyplot as plt

HEIGHT = 300
WIDTH = 300

BATCH_SIZE = 16
class_list = ["class_1", "class_2"]
FC_LAYERS = [1024, 512, 256]
dropout = 0.5
NUM_EPOCHS = 2
# BATCH_SIZE = 8

def build_model(base_model, dropout, fc_layers, num_classes):
    for layer in base_model.layers:
        layer.trainable = False

    x = base_model.output
    x = Flatten()(x)
    for fc in fc_layers:
        print(fc)
        x = Dense(fc, activation='relu')(x)
        x = Dropout(dropout)(x)
    preditions = Dense(num_classes, activation='softmax')(x)
    finetune_model = Model(inputs = base_model.input, outputs = preditions)
    return finetune_model

base_model_1 = ResNet50(weights = None,
                        include_top = False,
                        input_shape = (HEIGHT, WIDTH, 3))
train_datagen = ImageDataGenerator(preprocessing_function = preprocess_input,
                                   rotation_range = 90,
                                   horizontal_flip = True,
                                   vertical_flip = True,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   zoom_range=0.1,)
test_datagen = ImageDataGenerator(preprocessing_function = preprocess_input,
                                  rotation_range = 90,
                                  horizontal_flip = True,
                                  vertical_flip = False)

train_generator = train_datagen.flow_from_directory(TRAIN_DIR,
                                                    target_size = (HEIGHT, WIDTH),
                                                    batch_size = BATCH_SIZE)

test_generator = test_datagen.flow_from_directory(TEST_DIR,
```

```
                                                    target_size = (HEIGHT, WIDTH),
                                                    batch_size = BATCH_SIZE)

#making of the resnet50
resnet50_model = build_model(base_model_1,
                                        dropout = dropout,
                                        fc_layers = FC_LAYERS,
                                        num_classes = len(class_list))

adam = Adam()
resnet50_model.compile(adam, loss="binary_crossentropy", metrics=["accuracy"])

filepath = "./checkpoints" + "RestNet50" + "_model_weights.h5"
checkpoint = keras.callbacks.ModelCheckpoint(filepath, monitor = ["acc"], verbose= 1,
     mode = "max")

callbacks_list = [checkpoint]

print(train_generator.class_indices)

resnet50_model.summary()
```

Listing 4.7: ResNet50

```
variable = resnet50_model.fit_generator(generator = train_generator, epochs =
    NUM_EPOCHS, steps_per_epoch = 30,
                                        shuffle = True, validation_data =
    test_generator)
```

Listing 4.8: training

```
from tensorflow.keras.applications import resnet50
resnet50_model = resnet50.ResNet50(weights='imagenet')
resnet50_model.save("model.h5")
from IPython.display import FileLink
FileLink(r'/content/model.h5')

plt.figure(1, figsize = (15,8))

plt.subplot(221)
plt.plot(variable.history['accuracy'])
plt.plot(variable.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'valid'])

plt.subplot(222)
```

```
plt.plot(variable.history['loss'])
plt.plot(variable.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'valid'])

plt.show()
```

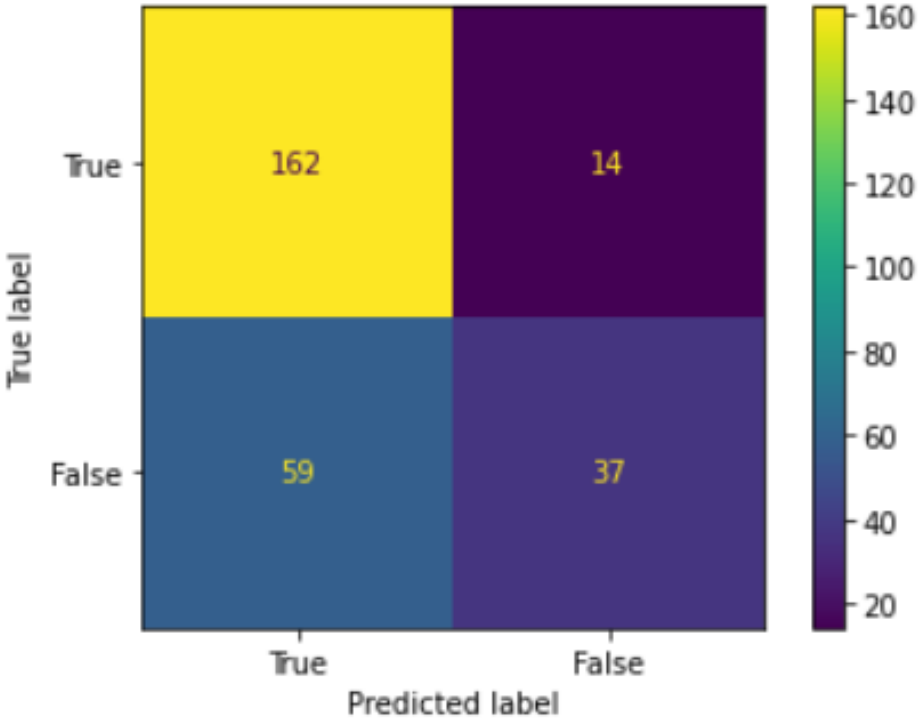
Listing 4.9: variation of epoch

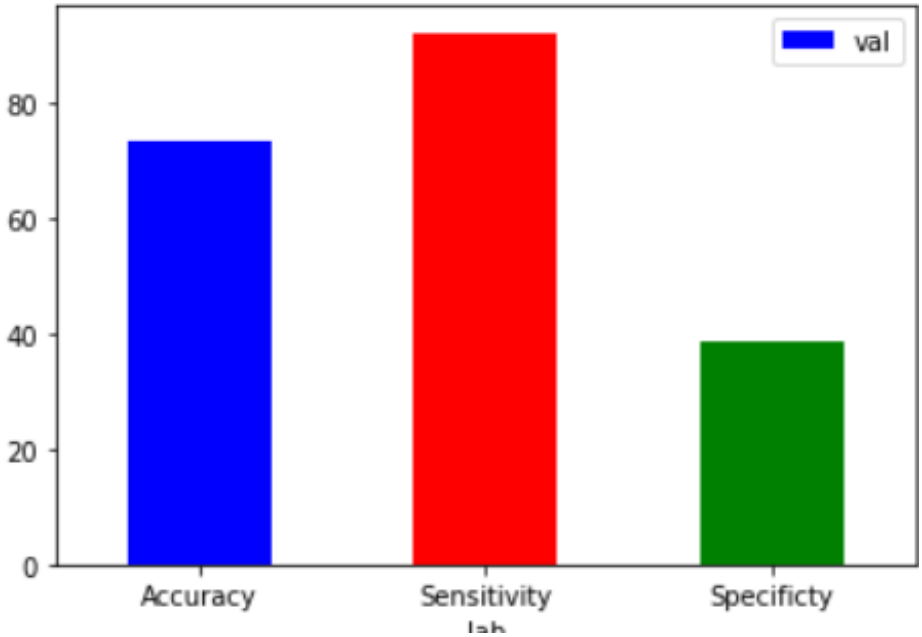
```
import os
from keras.models import load_model
from PIL import Image
from keras.preprocessing import image
import numpy as np
import cv2


# target_size = (512,512)
model=load_model('/content/model.h5')
print("model loaded")

import tensorflow as tf
import cv2
import numpy as np
import matplotlib.pyplot as plt


def predict_class(path):
    img = cv2.imread(path)

    RGBImg = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
    RGBImg= cv2.resize(RGBImg,(224,224))
    plt.imshow(RGBImg)
    image = np.array(RGBImg) / 255.0
    # new_model = tf.keras.models.load_model("/content/model.h5")
    predict=model.predict(np.array([image]))
    per=np.argmax(predict,axis=1)
    if per==1:
        print('Negative Glaucoma')
    else:
        print('Positive Glaucoma')
predict_class('/content/drive/MyDrive/Colab Notebooks/ML/Glaucoma Detection/
    Fundus_Train_Val_Data/Fundus_Scanes_Sorted/Train/Glaucoma_Negative/083.jpg')
```

Listing 4.10: Prediction of the given model

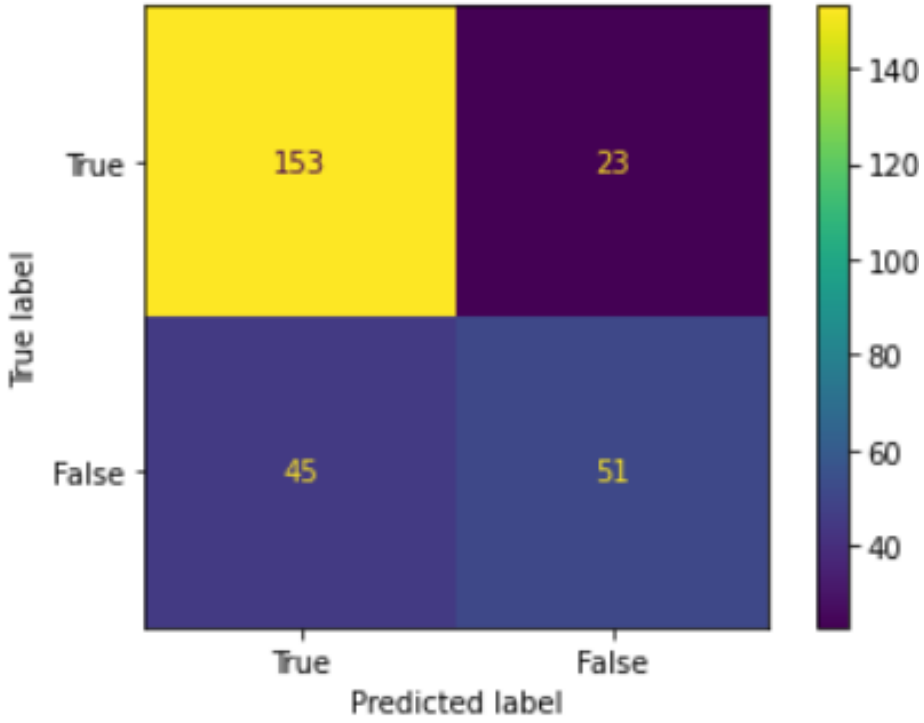
4.6 Performance analysis of the Model

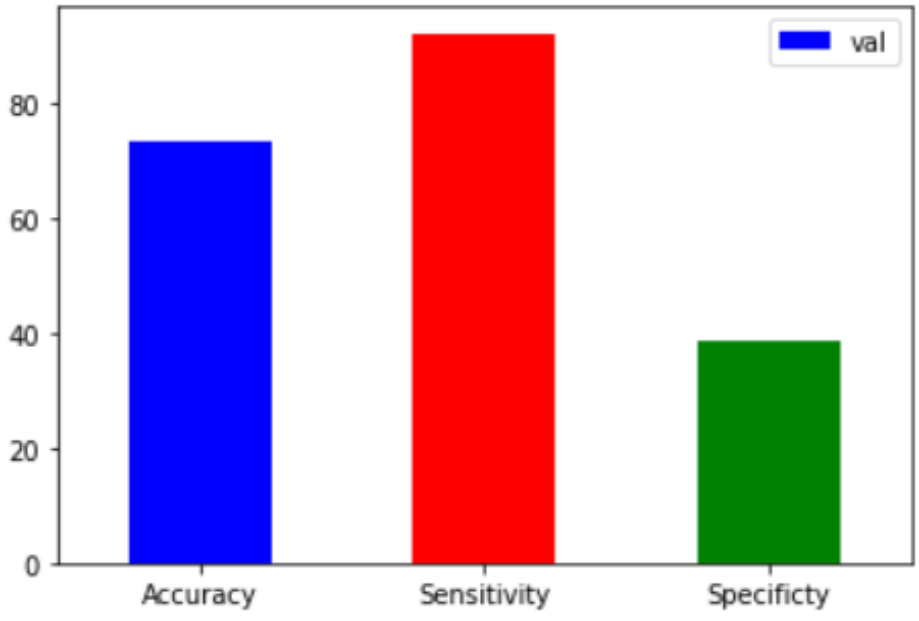
4.6.1 KNN



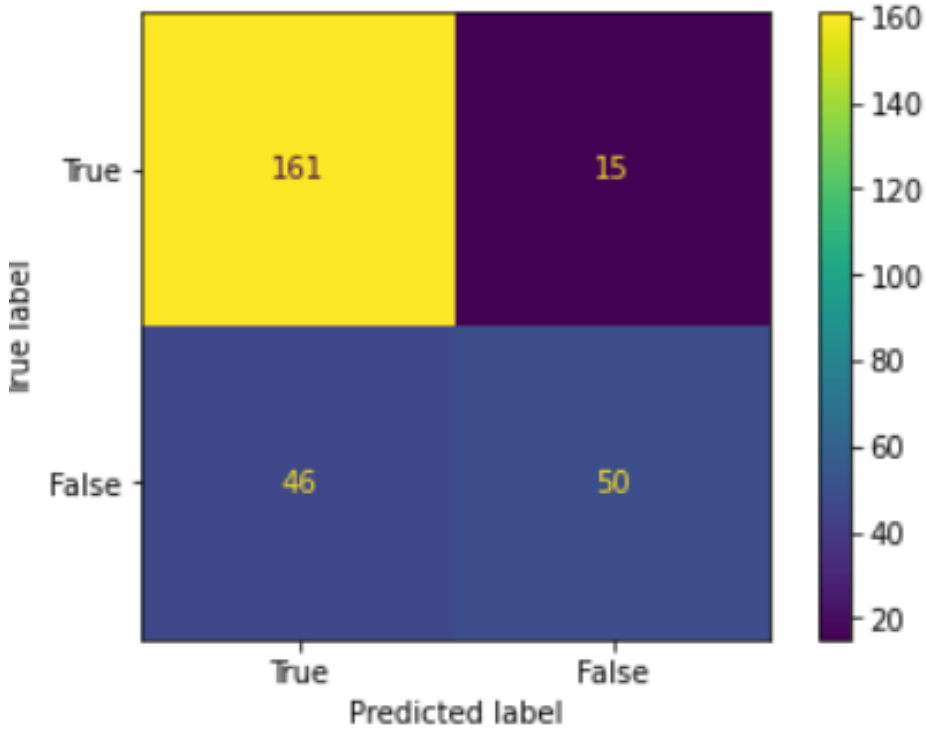


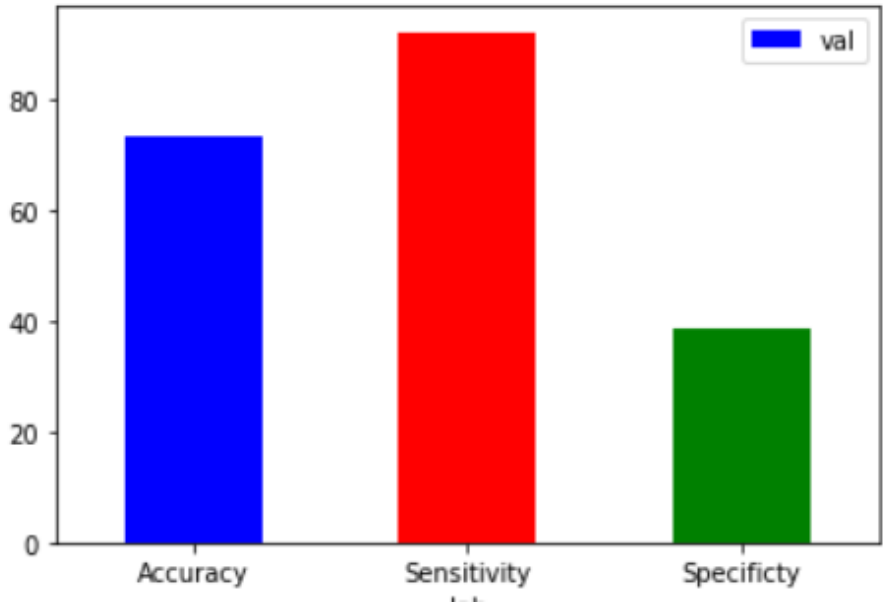
4.6.2 SVM


153
23
45
51
True
False
True label
True
False
Predicted label
140
120
100
80
60
40


val
80
60
40
20
0
Accuracy
Sensitivity
Specificty

4.6.3 RANDOM FOREST





Chapter 5

Future Work and Conclusion

In conclusion, our machine learning project aimed to develop a model that could accurately detect the presence of glaucoma in patients. Through the use of convolutional neural networks and the ResNet50 architecture, we were able to train a model that achieved high accuracy in detecting glaucoma in retinal images.

Our results indicate that machine learning has the potential to be a valuable tool in the early detection of glaucoma, allowing for earlier intervention and improved outcomes for patients. While our model achieved high accuracy, there is still room for improvement and further research in this area.

Overall, we believe that our project demonstrates the power of machine learning in healthcare and highlights the potential for continued innovation and development in this field. With further refinement and optimization, machine learning could become an important tool in the fight against glaucoma and other eye diseases.

...

Bibliography

- Glaucoma Research Foundation, "Types of Glaucoma," Glaucoma Research Foundation, 18 August 2016. [Online]. Available: http://www.glaucoma.org/glaucoma/types-of-glaucoma.php. [Accessed 11 February 2017]
- Glaucoma Research Foundation, "Glaucoma Facts and Stats," Glaucoma Research Foundation, 18 November 2016. [Online]. Available: http://www.glaucoma.org/glaucoma/glaucoma-facts-and-stats.php. [Accessed 10 February 2017]
- https://www.kaggle.com/datasets/keras/resnet50
- 'Glaucoma Dataset RIM-ONE', https://medimrg.webs.ull.es/research/retinalimaging/rim-one, January 27, 2019.