Indian Institute of Technology Kanpur

MTH443A: Statistical & AI Techniques in Data Mining

Report On

Predictive Modeling for Parkinson's Disease Detection

Submitted by: Sarbojit Das (231080075)
Sayan Mukherjee (231080079)
Krishanu Mukherjee (231080051)
Soumo Biswas (231080092)

Course Instructor: Prof. Dr. Amit Mitra

Date: November 14, 2024

Abstract

Parkinson's disease is a brain disorder that causes movement problems and can also impact mental health and sleep. Early diagnosis is challenging due to the absence of definitive tests. However, distinctive vocal characteristics present a good diagnostic opportunity. In this project, we have utilized machine learning algorithms to analyze voice recordings for early detection of PD. We have developed classifiers to distinguish healthy individuals from those with PD, applied Principal Component Analysis for feature selection, clustering of patients via PCA to understand disease progression. And based on our findings, we have recommended the better classifier that could significantly improve early diagnosis of Parkinson's Disease with less error rate.

Contents

List of Tables

List of Figures

1 Introduction

Parkinson's Disease (PD) is a progressive neurodegenerative disorder characterized by loss of dopamine-producing neurons in the brain. This depletion of dopamine leads to motor impairments, including tremors and slowness of movement. As it progresses, individuals may also experience a variety of other symptoms, notably affecting speech with manifestations such as difficulty in articulating words, a reduced volume of speech and a monotonic voice that lacks the usual pitch variations. This disease can also impact mood leading to the risk of depression and dementia among affected individuals.

Early diagnosis is crucial, yet challenging, due to absence of definitive laboratory tests for PD. Given that the individuals with PD often exhibit distinctive vocal characteristics, their voice recordings may present a promising, non-invasive tool for diagnosis. Leveraging machine learning algorithms to analyze voice recordings could provide an accurate screening mechanism, potentially facilitating early detection of PD before a patient visits a clinician.

2 Objectives

The goals of this project are outlined as follows:

1. **Early Detection of Parkinson's Disease:** To build classifiers that distinguish between the healthy individuals and those afflicted with Parkinson's disease using ML algorithms such as Logistic Regression, Support Vector Machines, Neural Networks etc.

2. **Feature Selection and Dimensionality Reduction:** To apply techniques such as Principal Component Analysis (PCA) to reduce the feature space and select the most relevant biomarkers.

3. **Clustering of Patients:** To find subgroups of patients who may have different manifestations of the disease to help in understanding disease progression.

4. **Cross-Validation and Model Comparison:** To compare classifiers (e.g., SVM, KNN etc.) in terms of accuracy using techniques like cross-validation.

3 Dataset

- **Data Source:** Parkinson's Disease dataset.
- **Observations:** 195 total observations.
- **Variables:** 22 features, 1 categorical and 1 response variables respectively.

Codes for this project are written in the `R` and `Python` programming language.

4 Exploratory Data Analysis

4.1 Renamed Columns

For better clarity and ease of analysis, the columns are renamed to more intuitive identifiers. Following are the renamed columns associated with this dataset:

Original column	**Renamed column**	**Description**
name	name	ASCII subject name and recording number
MDVP.Fo.Hz.	avg_fre	Average vocal fundamental frequency
MDVP.Fhi.Hz.	max_fre	Maximum vocal fundamental frequency
MDVP.Flo.Hz.	min_fre	Minimum vocal fundamental frequency
MDVP.Jitter...	var_fre1	Measures of variation in fundamental frequency
MDVP.Jitter.Abs.	var_fre2	
MDVP.RAP	var_fre3	
MDVP.PPQ	var_fre4	
Jitter.DDP	var_fre5	
MDVP.Shimmer	var_amp1	Measures of variation in amplitude
MDVP.Shimmer.dB.	var_amp2	
Shimmer.APQ3	var_amp3	
Shimmer.APQ5	var_amp4	
MDVP.APQ	var_amp5	
Shimmer.DDA	var_amp6	
NHR	NHR	Measures of ratio of noise to tonal components in voice
HNR	HNR	
status	status	1 - Parkinson's, 0 - Healthy
RPDE	RPDE	Nonlinear dynamical complexity measures
DFA	DFA	
spread1	spread1	Nonlinear measures of fundamental frequency variation
spread2	spread2	
PPE	PPE	

Table 1: Table of Original and Renamed Columns with description

4.2 Data Structure

The structure of the dataset is summarized using the `str()` function. This provides insight into the data types of each column.

- `name` column as a factor.
- `status` column as a indicator of Parkinson's (1 for Parkinson's & 0 for healthy).
- All other columns are numeric.

4.3 Data Quality Checks

- **Missing Values:** The presence of missing values is assessed using the `any()` function combined with `is.na()` in `R`. The result indicates that there are no missing values.
- **Duplicates:** The dataset is checked for duplicate entries using the `duplicated()` function, which returns zero duplicates, confirming the integrity of the dataset.

4.4 Data Imbalance Checks

From the below pie chart, we see that there is unequal representation of people afflicted with Parkinson (75.4%) and those who are healthy (24.6%).

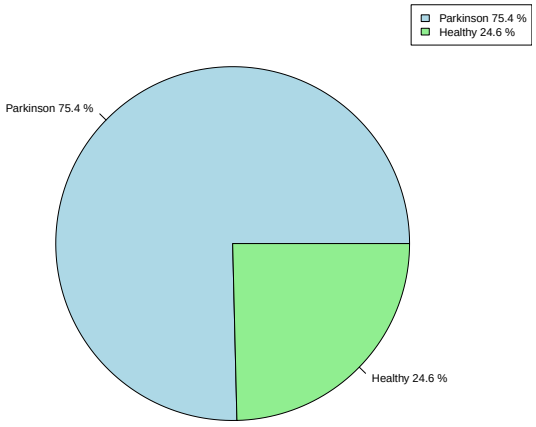


Figure 1: Pie Chart of class imbalance in dataset

4.5 Statistical Summary

A statistical summary is generated using the `summary()` function for all numeric variables (excluding `name`).

Variable	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
avg_fre	88.330	117.570	148.790	154.230	182.770	260.110
max_fre	102.100	134.900	175.800	197.100	224.200	592.000
min_fre	65.480	84.290	104.310	116.320	140.020	239.170
var_fre1	0.002	0.003	0.005	0.006	0.007	0.033
var_fre2	0.000	0.000	0.000	0.000	0.000	0.000
var_fre3	0.001	0.002	0.003	0.003	0.004	0.021
var_fre4	0.001	0.002	0.003	0.003	0.004	0.020
var_fre5	0.002	0.005	0.007	0.010	0.012	0.064
var_amp1	0.010	0.017	0.023	0.030	0.038	0.119
var_amp2	0.085	0.149	0.221	0.282	0.350	1.302
var_amp3	0.005	0.008	0.013	0.016	0.020	0.056
var_amp4	0.006	0.010	0.013	0.018	0.022	0.079
var_amp5	0.007	0.013	0.018	0.024	0.029	0.138
var_amp6	0.014	0.025	0.038	0.047	0.061	0.169
NHR	0.001	0.006	0.012	0.025	0.026	0.315
HNR	8.441	19.198	22.085	21.886	25.076	33.047
status	0.000	1.000	1.000	0.754	1.000	1.000
RPDE	0.257	0.421	0.496	0.499	0.588	0.685
DFA	0.574	0.675	0.722	0.718	0.762	0.825
spread1	-7.965	-6.450	-5.721	-5.684	-5.046	-2.434
spread2	0.006	0.174	0.219	0.227	0.279	0.450
D2	1.423	2.099	2.362	2.382	2.636	3.671
PPE	0.045	0.137	0.194	0.207	0.253	0.527

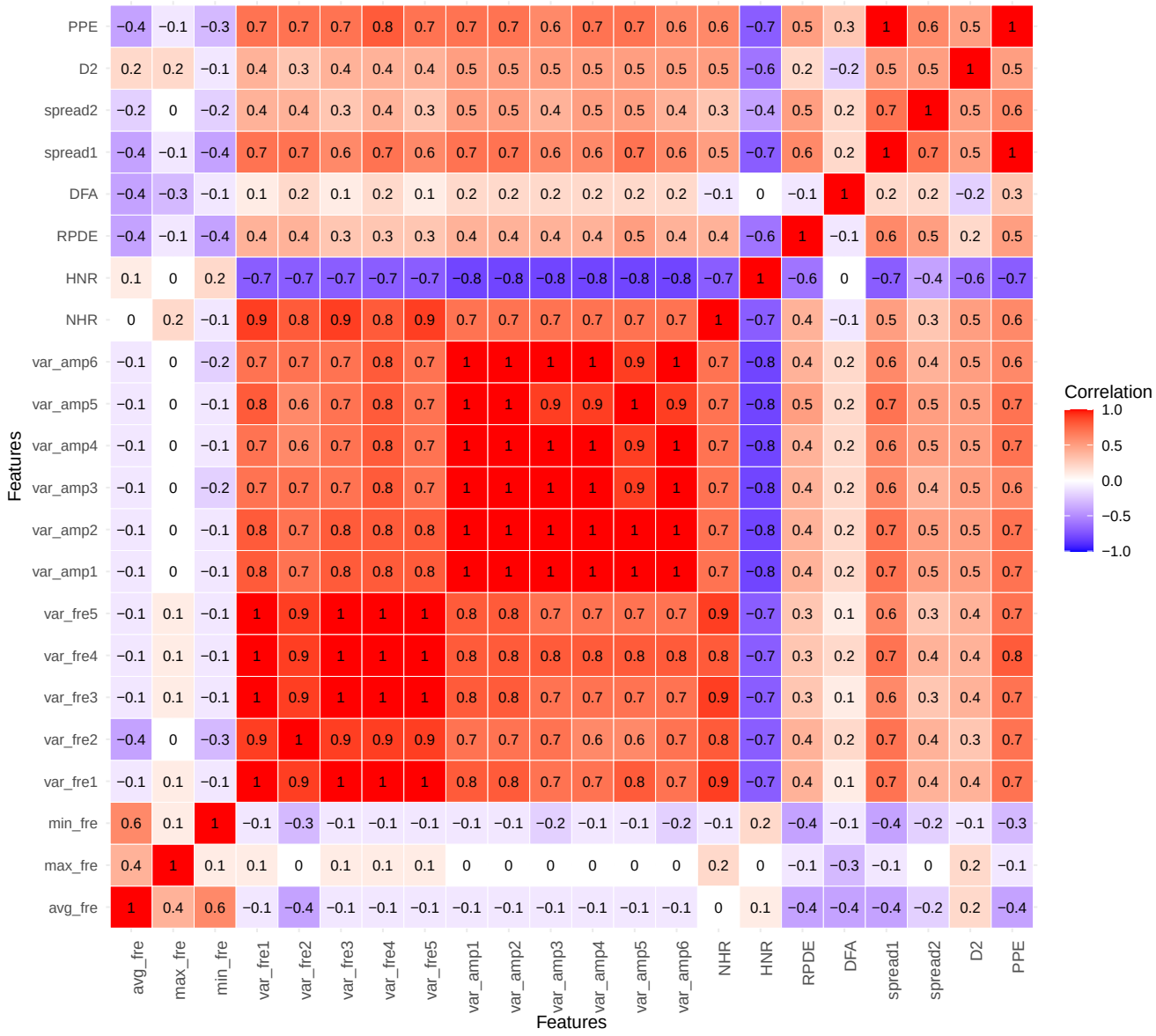
Table 2: Statistical Summary of Variables

- Frequency and amplitude-related features show significant variability, suggesting they are sensitive to differences in Parkinson's disease progression.
- The imbalance in the dataset (majority class being Parkinson's disease) suggests the need for careful consideration during model training to avoid biased predictions.

4.6 Visualisation of Correlation

The correlation matrix for the numeric features is calculated using the `cor()` function in `R`. This matrix assesses the linear relationship between pairs of variables.

Figure 2: Correlation Heatmap

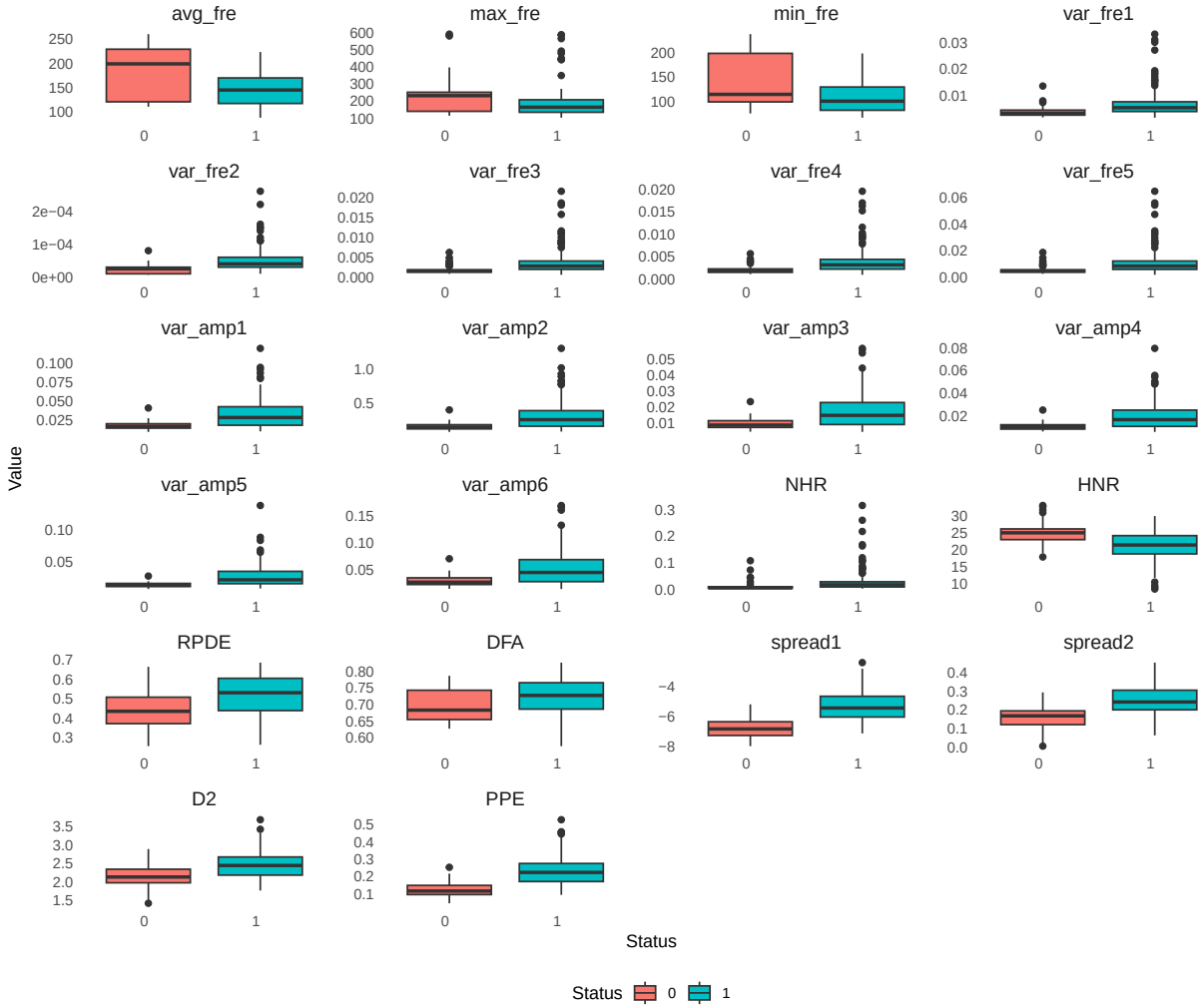


- In this correlation heatmap, we can see that many independent features are highly correlated with each other.

4.7 Visualisation of features via Boxplot

From the boxplot shown below, it is evident that if a patient has lower rates of `HNR`, `avg_fre`, `max_fre` and `min_fre`, then they are more likely to be affected by Parkinson's disease.

Figure 3: Boxplot of features based on patients' status



Remarks:

- The **2.5 times interquartile range (IQR) rule** is typically used to identify outliers. However, outliers detected by this rule may not always be true outliers in a clinical or medical context.
- These so-called "outliers" may represent important cases with different characteristics.

4.8 Pair Plot

From the above pair plot, we see that all these fundamental frequencies are highly correlated with each other.

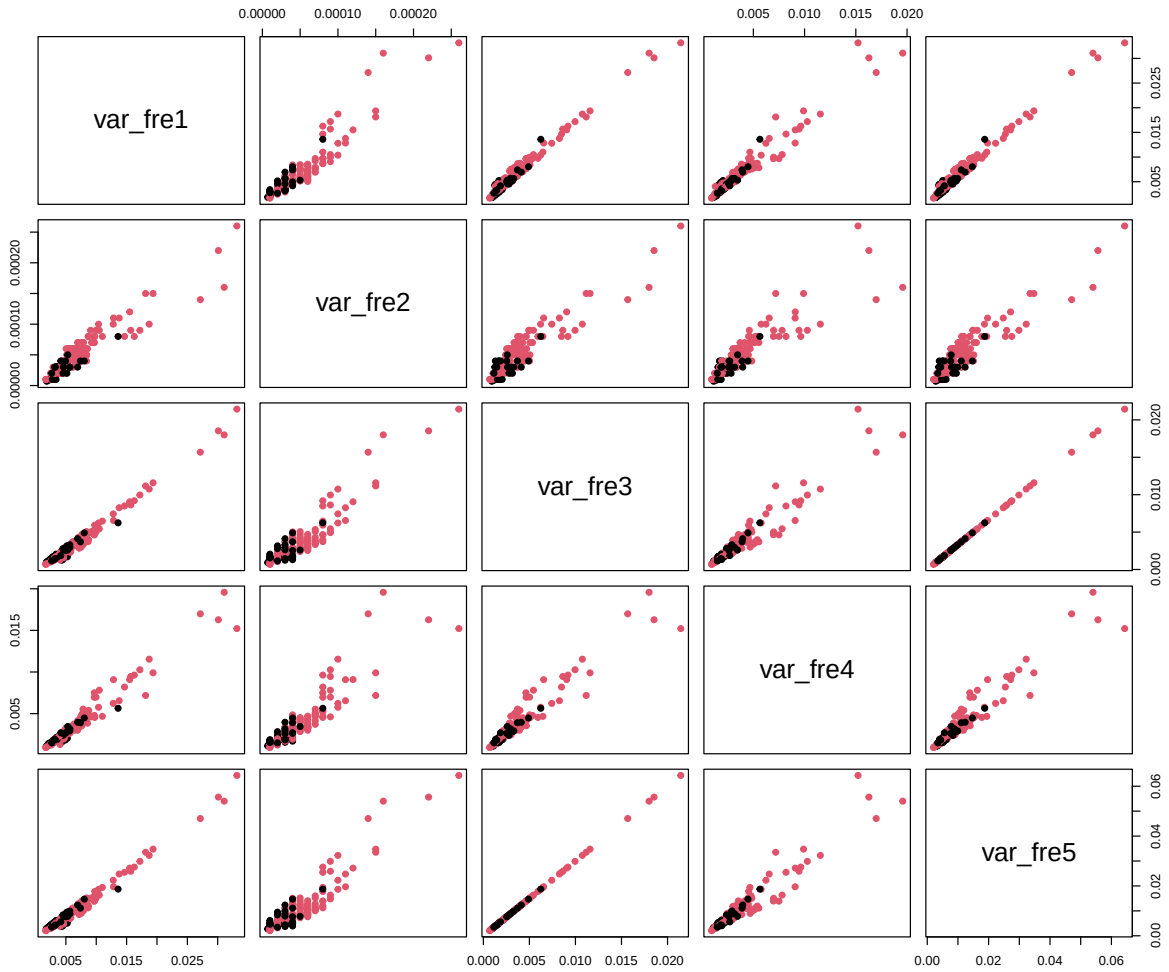


Figure 4: Pairplot of voice frequency by patient status

5 Principal Component Analysis

Our primary goal of carrying out PCA was to effectively reduce the dimensionality of the dataset, i.e. reduce the feature space, while preserving variabilty as much as possible. It also reduces multicollinearity, a common issue when the features are highly correlated, which can cause instability in regression coefficients.

- **Data Preparation:** We selected only the relevant features by excluding the `name` column (categorical variable) and the `status` column (response variable).
- **Standardization:** The numerical features were made unitless by using the `scale()` function in `R`. This step is crucial as it centers the data around zero and scales it to have unit variance, thus preventing features with larger ranges from disproportionately influencing the results.
- **Covariance Matrix and Eigen Decomposition:** We calculated the covariance matrix of the standardized data and performed eigen decomposition to extract the eigenvalues and eigenvectors.
- **Data Dimensionality Reduction:** Using the selected components, we reduced the dataset's dimensionality. The standardized data was multiplied by the selected eigenvectors to obtain the reduced dataset.
- **Data Cleaning:** By applying PCA for visualization and subsequently identifying outliers, we have enhanced the dataset's quality.

  The cleaned dataset (`parkinson_no_outlier.csv`) contains the data of subjects without the identified outliers. This ensures that the subsequent analyses, such as clustering or classification, are not adversely affected by extreme values that could skew results.

The reader may refer to sub-appendix A.1 for more details.

5.1 Selecting Optimal Principal Components

We wanted to determine the optimal number of principal components to retain for our analysis. The decision was based on examining the scree plot, the proportion of variance explained, and the cumulative variance explained by each component.

- **Scree Plot:** It represents the eigenvalues (Y-axis) associated with each principal component (X-axis). By inspecting this plot, we identified a clear inflection point at the 8^{th} principal component beyond which the eigenvalues showed a diminishing return in variance explanation.

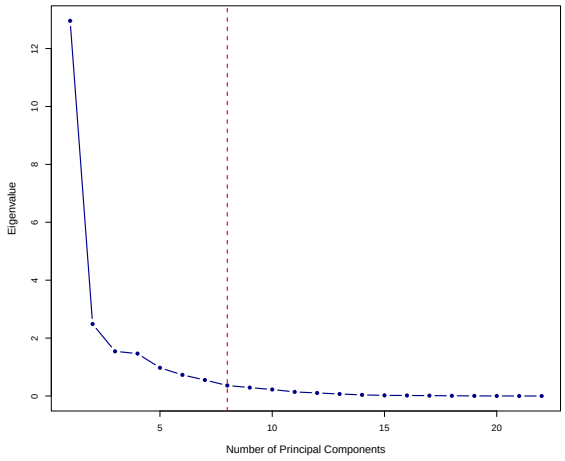


Figure 5: Scree Plot

- **Cumulative Variance Explained Plot:** The cumulative variance plot confirmed that retaining the first eight principal components explained approximately 95.76% of the total variance in the data.

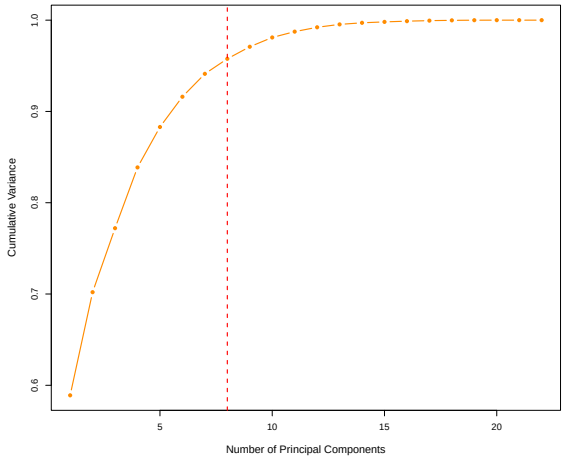


Figure 6: Cumulative Variance Explained by principal components

5.2 Importance of Components

We calculated the principal components using the `prcomp()` function, which provided a summary of the importance of each component, confirming the variance explained by each retained component.

Table 3: PCA Component Table

Component	S.D.	Prop. of Var.	Cumulative Prop.
PC1	3.600	0.589	0.589
PC2	1.577	0.113	0.702
PC3	1.242	0.070	0.772
PC4	1.210	0.067	0.839
PC5	0.987	0.044	0.883
PC6	0.854	0.033	0.916
PC7	0.743	0.025	0.941
PC8	0.602	0.016	0.958
PC9	0.538	0.013	0.971
PC10	0.473	0.010	0.981
PC11	0.375	0.006	0.987
PC12	0.324	0.005	0.992
PC13	0.264	0.003	0.995
PC14	0.195	0.002	0.997
PC15	0.148	0.001	0.998
PC16	0.133	0.001	0.999
PC17	0.112	0.001	0.999
PC18	0.085	0.000	1.000
PC19	0.059	0.000	1.000
PC20	0.033	0.000	1.000
PC21	0.001	0.000	1.000
PC22	0.000	0.000	1.000

5.3 Outlier Detection

In this analysis, we focus on identifying and removing outliers from this dataset concerning Parkinson's disease detection, specifically using the principal component analysis (PCA).

5.3.1 2-Dimensional Visualisation

In this analysis, we utilized a two-dimensional plot based on the first two principal components (PC1 and PC2) derived from PCA. This approach allows us to visually inspect the distribution of the data and identify potential outliers.

1. **Plotting:** A scatter plot was created to visualize the first two principal components.
2. **Outliers:** After plotting the data, specific points were identified as outliers based on their visual positioning in the plot. The outliers detected in this analysis were:
   - `phon_R01_S35_4`
   - `phon_R01_S24_4`
   - `phon_R01_S24_6`
   - `phon_R01_S35_6`
   - `phon_R01_S35_7`

   Red labels were added only to those points detected as outliers using the corresponding subject names from the original dataset.

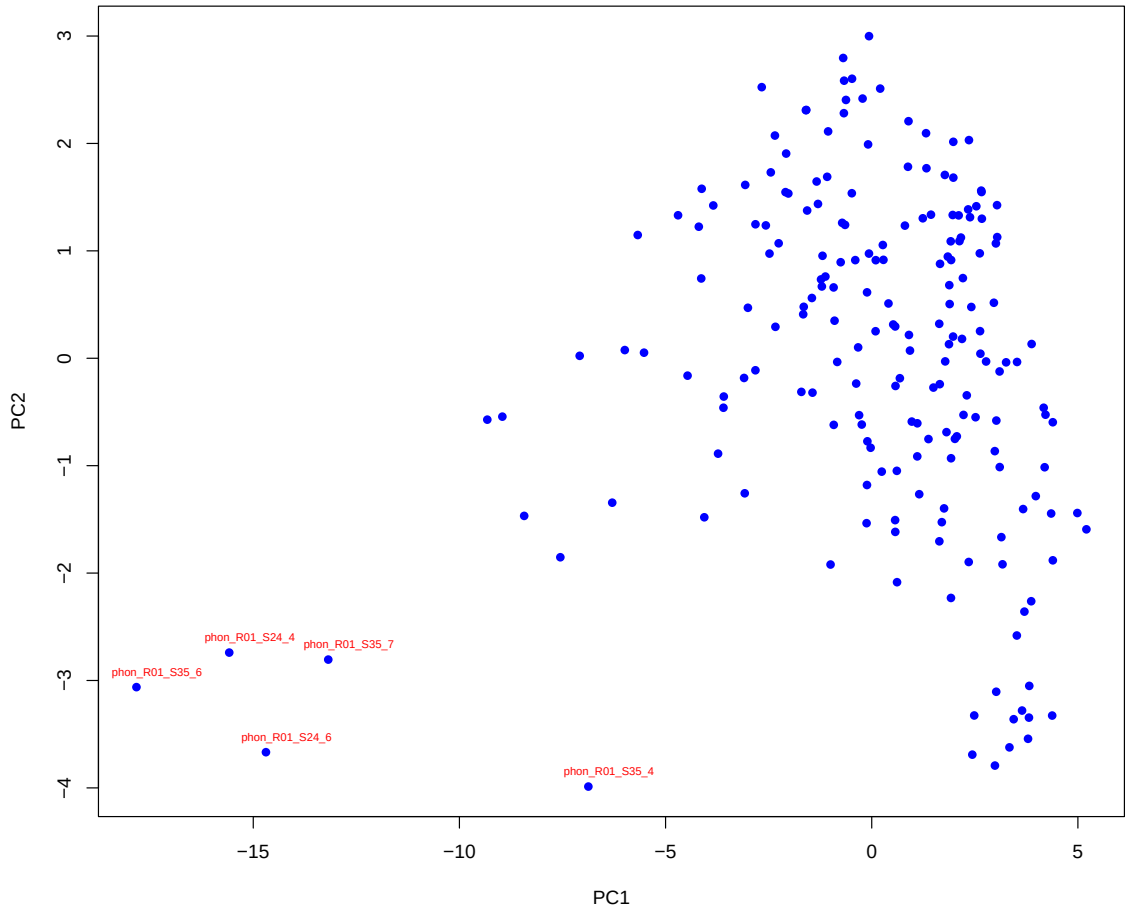


Figure 7: 2D scatter plot for outlier detection

5.3.2 Chernoff faces

Chernoff faces are a way of representing multivariate data where each feature influences different facial characteristics like the size of eyes, shape of face, etc.

Table 4: Variable Mapping for Chernoff Faces

Modified Item	**Variable**	**Principal Component**
Height of Face	Var1	PC1
Width of Face	Var2	PC2
Structure of Face	Var3	PC3
Height of Mouth	Var4	PC4
Width of Mouth	Var5	PC5
Smiling	Var6	PC6
Height of Eyes	Var7	PC7
Width of Eyes	Var8	PC8

Spotting Outliers: 5 subjects with facial features stand out significantly from the rest might be the potential outliers. They are as follows:

- `phon_R01_S35_4`
- `phon_R01_S24_4`
- `phon_R01_S24_6`
- `phon_R01_S35_6`
- `phon_R01_S35_7`

1. Observation: The potential outliers identified from the Chernoff faces match with those detected from the 2-Dimensional visualisation of the first 2 PCs.
2. Interpretation: Unusual faces hint at subjects with particularly extreme values in one or more features. These could be outliers that might need special attention in our analysis, such as checking for errors or investigating why these subjects are different.

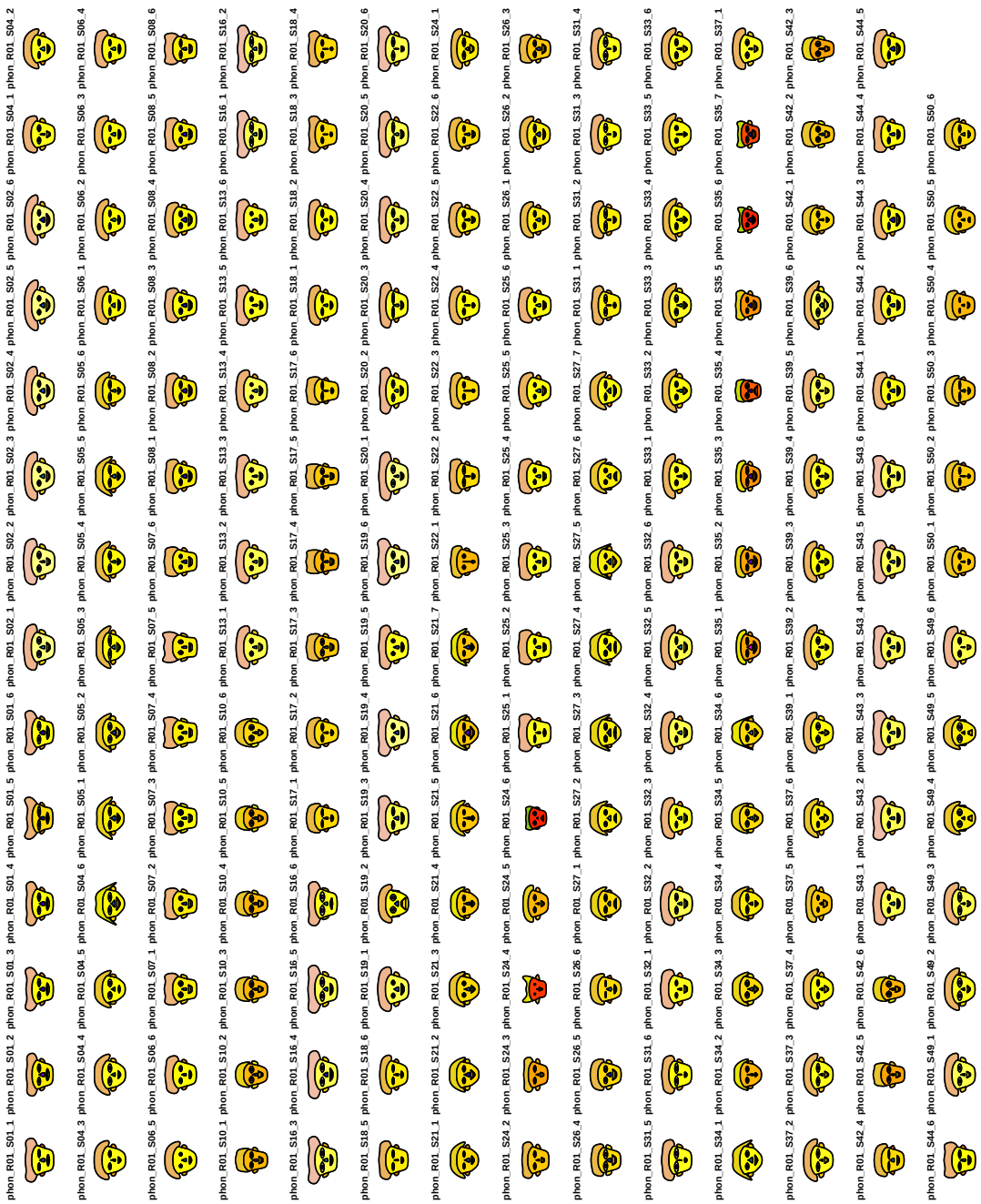


Figure 8: Chernoff faces representation for 195 observations

6 Clustering via PCA

3-Dimensional Visualisation:

We plotted the PCA-transformed feature space for the first three principal components that explains the most variabilty and color-coded based on the response variable i.e., the `status` (0 for non-Parkinson and 1 for Parkinson's).

Observations:

- **Clusters:** The presence of two distinct blue clusters (Healthy) within the red points (Parkinson) suggests that some healthy individuals may share almost same characteristics with individuals having Parkinson's. This overlap could indicate that certain features used in the analysis are not effective in distinguishing between the two groups, or that there is a spectrum of overlapping features. Ideally, we should have observed a clear demarcation between the healthy and Parkinson's cluster groups, with a boundary separating them. Therefore, the PCA clustering model is inconclusive.
- **Further Analysis:** To deepen our understanding, we can consider applying clustering algorithms like K-means to the PC scores to see if we can replicate the observed clusters.

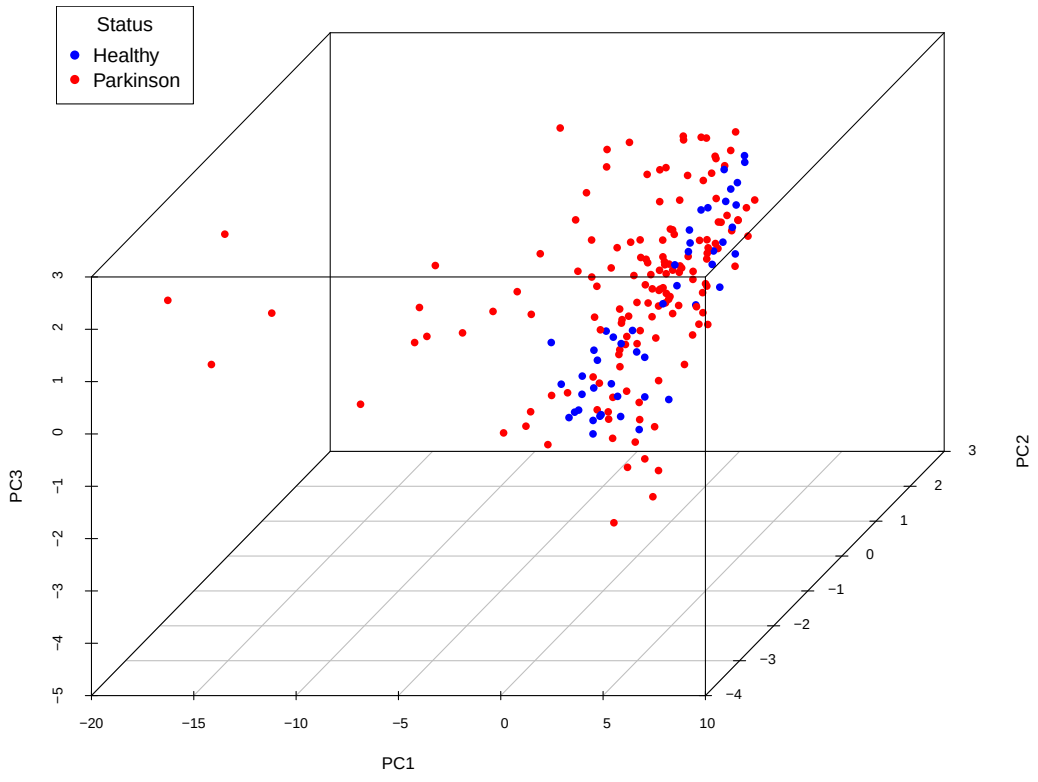


Figure 9: 3D visualization of first 3 principal components by response variable

7 Cross Validation

Since, the dataset is large we have resorted to a 5-fold Cross Validation. The steps involved are as follows:

- **Load the Data:** The data is read into a DataFrame format.
- **K-Fold Cross Validation:**
  1. A random permutation of the indices is generated, and the data is split into $K = 5$ groups or folds.
  2. One group (the 5-th group) is used as the test set, while the remaining groups are used to create the training set.
- **Calculate accuracy for each fold**: For each fold i, we calculated the accuracy A_i of the model on the test set D_i for $i = 1, \ldots, 5$.
- **Average the accuracies**: After completing all 5 folds, final cross-validation accuracy is average of the accuracies from each fold:

$$A_{\text{cv}} = \frac{1}{5} \sum_{i=1}^{5} A_i$$

The A_{cv} is considered as the model's expected performance on unseen data. Hence, there is no need for an additional test set if the goal is to estimate the generalization error.

For more details, the reader may refer to sub-appendix A.2.

8 Data Balancing

Since, the dataset is imbalanced we have employed the **Random Oversampling technique** to oversample the minority class and build various models over it. For more details, the reader may refer to sub-appendix A.6.

Dataset	**Class Label: 0**	**Class Label: 1**
Original	48	142
Balanced	142	142

Table 5: Distribution of the `status` variable in the original and balanced data sets.

9 Parametric Model Building

Here, we assume that the underlying distribution of the data is multivariate normal. Thus, for each class π_k:

$$\mathbf{x}|\pi_k \sim \mathcal{N}(\mu_k, \Sigma_k)$$

Hence,

$$p(\mathbf{x} \mid \pi_k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu_k)^T\Sigma_k^{-1}(\mathbf{x}-\mu_k)\right)$$

Where:

- μ_k is the mean vector of class π_k,
- $\mathbf{x}$ is the feature vector (e.g., frequency and amplitude features),
- μ_k is the mean vector for class π_k,
- Σ_k is the covariance matrix for class π_k.

Remark: In reality, we don't know whether our assumption actually holds or not (which is beyond the scope of this project!). We are just interested in seeing its performance over the non-parametric models. Violation of assumptions will likely lead to poor classification predictions.

9.1 Optimal Prior Choosing

Our goal is to choose prior for parametric models that yield the lowest misclassification error rate and for this we generated a sequence of prior probabilities.

- For the LDA model, the misclassification error rate decreases for prior values less than 0.15, becomes nearly constant and again increases for prior probability values greater than 0.85. Hence, we are not able to get a unique value.
- For the QDA and Naive Bayes models, the misclassification error rate curve remains almost constant for all prior probability values.
- Thus, a distinct "elbow" was not observed in all the 3 models indicating that choosing an optimal prior value is quite challenging given that the data is imbalanced.

Instead, we set $p(\pi_1) = 0.5$ and $p(\pi_2) = 0.5$.

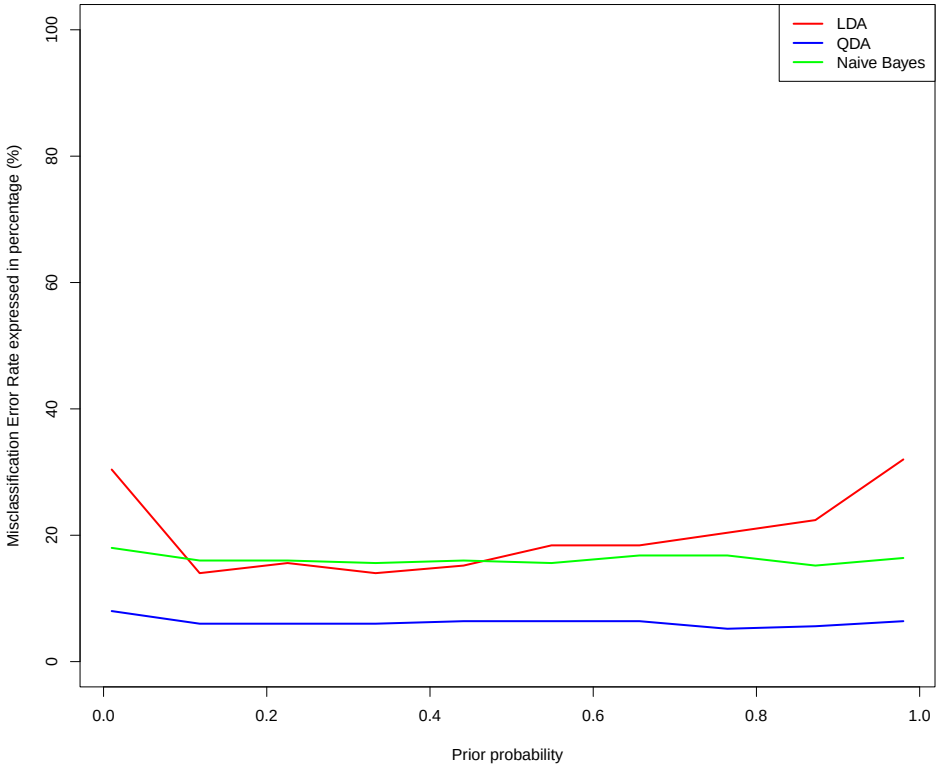


Figure 10: Choosing optimal prior probability value by grid search

9.2 Linear Discriminant Classifier

LDA tries to find a linear decision boundary that maximizes the separation between the classes. We assume equal misclassification costs. For more details, the reader may refer to sub-appendix A.3. The cross-validation scores for each fold are as follows:

9.2.1 Cross validation Fold Scores

$$\text{Cross-validation scores} = [0.84, 0.80, 0.84, 0.84, 0.78]$$

9.2.2 Average Accuracy

The average accuracy across all folds is:

$$\text{Average accuracy} = 0.820 \times 100 = 82\%$$

9.2.3 Confusion Matrices

Following are the confusion matries obtained for $K = 1$ in the 5-fold Cross Validation.

Actual	**Predicted**	
	1	0
1	19	2
0	6	23

Actual	**Predicted**	
	1	0
1	20	5
0	5	20

Actual	**Predicted**	
	1	0
1	21	4
0	4	21

Actual	**Predicted**	
	1	0
1	21	4
0	4	21

Actual	**Predicted**	
	1	0
1	18	4
0	7	21

Table 6: LDA Confusion Matrices for Folds 1 to 5

9.3 Quadratic Discriminant Classifier

We assume that classes have the unequal covariance matrix. Decision boundary is quadratic instead of linear. We assume equal misclassification costs. For more details, the reader may refer to sub-appendix A.4.

9.3.1 Cross validation Fold Scores

$$\text{Cross-validation scores} = [0.98, 0.94, 0.94, 0.94, 0.92]$$

9.3.2 Average Accuracy

The average accuracy across all folds is:

$$\text{Average accuracy} = 0.944 \times 100 = 94.4\%$$

9.3.3 Confusion Matrices

Following are the confusion matries obtained for $K = 1$ in the 5-fold Cross Validation.

Actual	**Predicted**	
	1	0
1	24	0
0	1	25

Actual	**Predicted**	
	1	0
1	23	1
0	2	24

Actual	**Predicted**	
	1	0
1	22	0
0	3	25

Actual	**Predicted**	
	1	0
1	24	2
0	1	23

Actual	**Predicted**	
	1	0
1	23	2
0	2	23

Table 7: QDA Confusion Matrices for Folds 1 to 5

9.4 Naive Bayes Classifier

This classifier is based on Bayes' Theorem, which describes the probability of a class given some observed features. For more details, the reader may refer to sub-appendix A.5.

9.4.1 Cross validation Fold Scores

$$\text{Cross-validation scores} = [0.82, 0.84, 0.76, 0.78, 0.76]$$

9.4.2 Average Accuracy

The average accuracy across all folds is:

$$\text{Average accuracy} = 0.792 \times 100 = 79.2\%$$

9.4.3 Confusion Matrices

Actual	Predicted	
	1	0
0	7	23
1	18	2

Actual	Predicted	
	1	0
0	6	23
1	19	2

Actual	Predicted	
	1	0
0	8	21
1	17	4

Actual	Predicted	
	1	0
0	7	21
1	18	4

Actual	Predicted	
	1	0
0	10	23
1	15	2

Table 8: Bayes Confusion Matrices for Folds 1 to 5

10 Non-Parametric Model Building

Non-Parametric models are data-driven and do not rely on assumptions about the underlying data distribution (e.g., normal, exponential). This makes them well-suited for real-world datasets that may not fit traditional distributional forms.

10.1 K-Nearest Neighbor Classifier

Our aim is to find the optimal K-nearest data point to a given query point and using majority voting to assign class labels to the feature vectors. For more details, the reader may refer to sub-appendix A.8.

10.1.1 Optimal Value of K for KNN Classifier

To determine the optimal value of K for the K-Nearest Neighbors (KNN) classifier, we have performed the following steps:

- We have plotted the misclassification error rate for values of K ranging from 1 to 20.
- The plot revealed a discrepancy in the pattern of errors. Error rate increases as K increases.
- Among the various values of K, we observed that for $K = 1$, the misclassification error rate was the lowest, indicating the best performance on the dataset.
- Based on the observed error rates, we have chosen $K = 1$ as the optimal value for the KNN classifier.

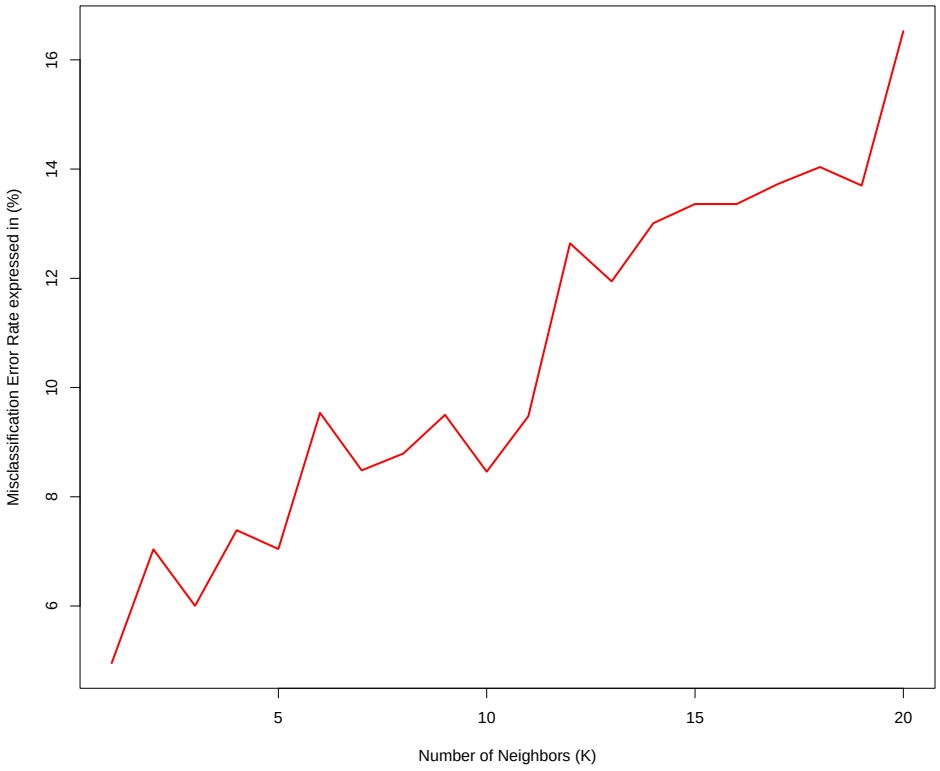


Figure 11: Choosing the optimal value of nearest neighbor for KNN

10.1.2 KNN Classifier Performance with $K = 1$

We evaluated the performance of the K-Nearest Neighbors (KNN) classifier with $K = 1$ using 5-fold cross-validation. The cross-validation scores for each fold are as follows:

$$\text{Cross-validation scores} = [0.9473684, 0.9642857, 0.9473684, 0.9482759, 0.9285714]$$

The average accuracy across all folds is:

$$\text{Average accuracy} = 0.947174 \times 100 = 94.7174\%$$

10.1.3 Confusion Matrices

Following are the confusion matries obtained for $K = 1$ in the 5-fold Cross Validation.

Actual	**Predicted**	
	1	0
1	26	0
0	3	28

Actual	**Predicted**	
	1	0
1	27	1
0	1	27

Actual	**Predicted**	
	1	0
1	25	0
0	3	29

Actual	**Predicted**	
	1	0
1	26	0
0	3	29

Actual	**Predicted**	
	1	0
1	24	0
0	4	28

Table 9: KNN Confusion Matrices for Folds 1 to 5

10.2 Support Vector Machines

The goal of SVM is to find a hyperplane that best separates the data points of different classes. For more details, the reader may refer to sub-appendix A.9.

10.2.1 Cross validation Fold Scores

$$\text{Cross-validation scores} = [0.9703, 0.9776, 0.9326, 0.9388, 0.9102]$$

10.2.2 Average Accuracy

The average accuracy across all folds is:

$$\text{Average accuracy} = 0.9459 \times 100 = 94.59\%$$

10.2.3 Confusion Matrices

Actual	**Predicted**	
	1	0
0	7	0
1	2	28

Table 10: Confusion Matrix for SVM

10.3 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models inspired by biological neural networks, consisting of interconnected neurons arranged in layers. They process information through layer-by-layer computations, with the backpropagation algorithm used for training. For more details, the reader may refer to sub-appendix A.12.

10.3.1 Conclusion

From the results, we can conclude that we have performed 5 generarations under genetic algorithm and got the highest accuracy in 4th generation which is 93%.

The highest accuracy in the 4th generation of the genetic algorithm (GA) indicates that the algorithm has likely discovered a near-optimal solution for ANN. In the initial generations, the GA explores a wide range of potential solutions, and by the 4th generation, it has likely narrowed down to an effective set of parameters. After this point, additional generations may not yield much improvement in accuracy, possibly due to limited diversity in the solutions, overfitting, or the algorithm getting stuck in a local optimum.

10.4 Decision Trees

It builds a tree recursively by splitting the data at each node until one of the stopping criteria is met. For more details, the reader may refer to sub-appendix A.10.

10.4.1 Average Accuracy

The average accuracy across all folds is:

$$\text{Average accuracy} = 0.8649 \times 100 = 86.49\%$$

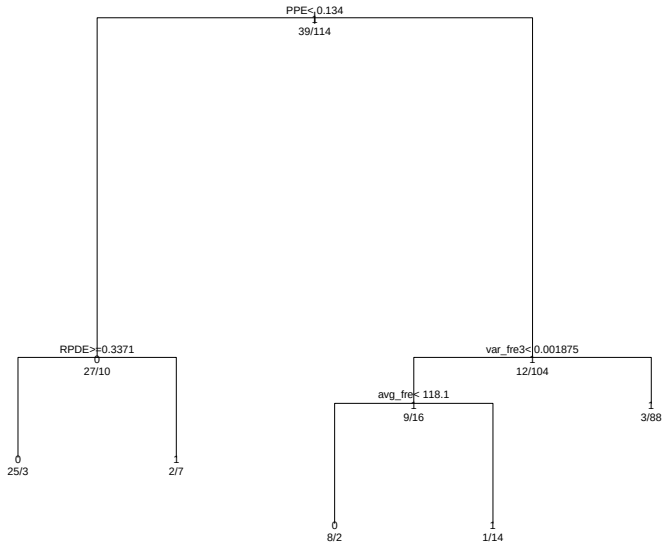


Figure 12: Decision Trees

10.4.2 Confusion Matrices

Actual	**Predicted**	
	1	0
0	7	0
1	2	28

Table 11: Confusion Matrix for Decision Trees

10.5 Random Forest

This alogrithm combines multiple decision trees to improve prediction accuracy and reduce overfitting. For more details, the reader may refer to sub-appendix A.11.

10.5.1 Average Accuracy

The average accuracy across all folds is:

$$\text{Average accuracy} = 0.973 \times 100 = 97.3\%$$

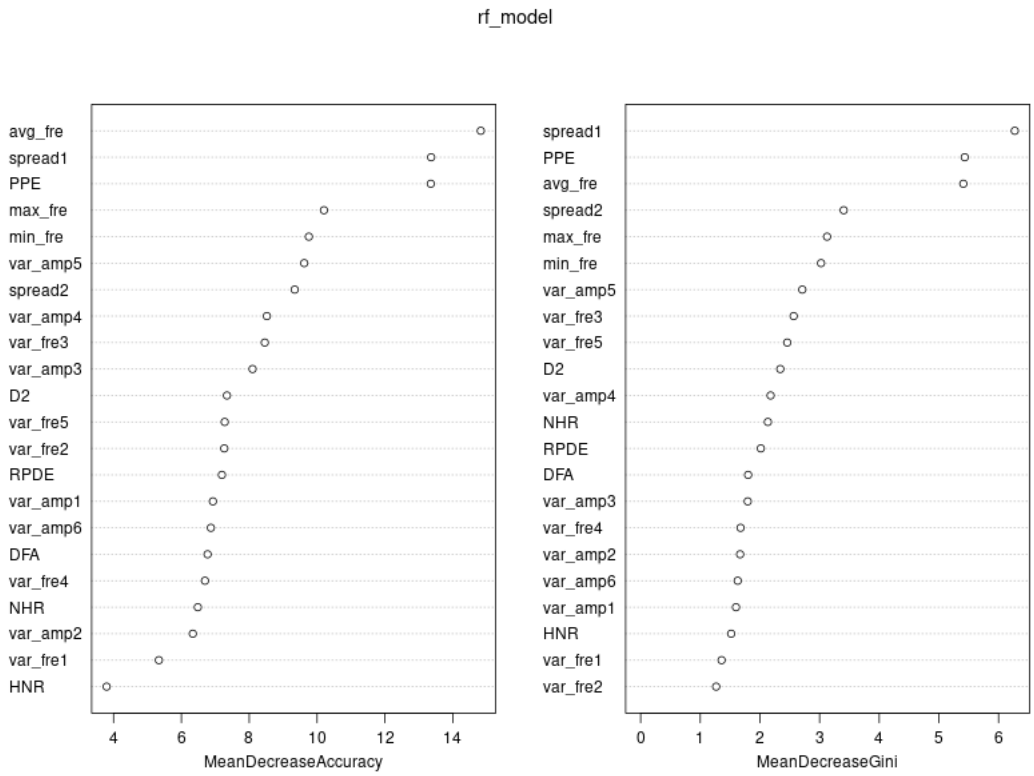


Figure 13: Variable Importance in Random Forest

10.5.2 Confusion Matrices

Actual	**Predicted**	
	1	0
0	8	0
1	1	28

Table 12: Confusion Matrix for Random Forest

11 Choosing the Better Model

- From the various models we have built, we observed that **non-parametric models** significantly outperform the parametric models in terms of accuracy.
- **Naive Bayes Classifier** yields the lowest accuarcy among all the models that we have tested.
- Among the non-parametric models, **Random Forest** achieves the highest accuracy, followed closely by **Decision Trees**.
- Given the high performance of Random Forest, with minimal chances of error, we suggest its use for the early and accurate identification of individuals with Parkinson's disease. This approach promises to enhance diagnostic precision and support timely interventions.

Model Type	**Average Accuracy (%)**
Parametric Models	
LDA	82.0
QDA	94.4
Naive Bayes	79.2
Non-Parametric Models	
KNN	94.714
SVM	94.59
Decision Trees	96.49
Random Forest	97.3
ANN	93.71

Table 13: Model Accuracy Comparison

12 Conclusion

After evaluating multiple algorithms, we found that non-parametric models, particularly Random Forest, significantly outperform parametric models in terms of accuracy. The Naive Bayes classifier, on the other hand, yielded the lowest accuracy across all tested models. Also, the genetic algorithm (GA) optimization for the artificial neural network (ANN) achieved its highest accuracy of 93% in the 4th generation. The lack of further accuracy improvements in subsequent generations likely indicates diminishing diversity among solutions, potential overfitting, or convergence to a local optimum, where additional iterations provided minimal benefit. Our findings from this project support the potential of using machine learning, for the early detection of Parkinson's disease based on vocal characteristics. This approach could lead to more accurate and timely diagnoses, facilitating earlier interventions and improving patient outcomes. Future research should focus on further refining these models and incorporating additional features to enhance their diagnostic precision.

13 Acknowledgement

We express our sincere gratitude to **Prof. Dr. Amit Mitra** for his continuous supervision and invaluable support throughout the completion of this project. His guidance, insights and encouragement have been fundamental in shaping our trajectory. We had the privilege of applying the knowledge and skills gained under this course to tackle a real-world problem. This opportunity allowed us to delve deep into the subject matter and hone our abilities through practical applications.

References

[1] T. Hastie, R. Tibshirani and J, Friedman. *"The elements of statistical learning: Data Mining, Inference and Prediction; Springer Series in Statistics"*

[2] R. A. Johnson and D.W. Wichern. *"Applied multivariate statistical analysis, Pearson"*

[3] Andrew Webb. *"Statistical Pattern Recognition, John Wiley & Sons."*

[4] S. S. Haykin. *"Neural Networks: A comprehensive foundation; Prentice Hall"*

A Appendix

A.1 Principal Component Analysis

Principal Component Analysis (PCA) is a widely used technique for dimensionality reduction and data visualization in machine learning and statistics. The goal of PCA is to reduce the number of features in a dataset while retaining the most significant information, typically measured as the variance in the data. PCA accomplishes this by transforming the data into a new set of variables known as *principal components*.

Mathematical Formulation

Given a dataset $\mathbf{X} \in \mathbb{R}^{n \times p}$, where n is the number of observations and p is the number of features, PCA involves the following steps:

1. Standardization

Before applying PCA, it is common to standardize the data so that each feature has a mean of 0 and a standard deviation of 1. This ensures that all features contribute equally to the analysis. The standardized data $\mathbf{Z}$ is obtained by:

$$Z_{ij} = \frac{X_{ij} - \mu_j}{\sigma_j},$$

where μ_j is the mean of feature j and σ_j is the standard deviation of feature j, defined as:

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} X_{ij}, \quad \sigma_j = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_{ij} - \mu_j)^2}.$$

Here, X_{ij} is the value of the j-th feature for the i-th sample.

2. Covariance Matrix

The covariance matrix $\mathbf{C} \in \mathbb{R}^{p \times p}$ of the standardized data $\mathbf{Z}$ is computed as:

$$\mathbf{C} = \frac{1}{n-1}\mathbf{Z}^\top \mathbf{Z}.$$

This matrix describes the variance and covariance between the features.

3. Eigenvalue Decomposition

Next, the covariance matrix $\mathbf{C}$ is decomposed into eigenvalues and eigenvectors:

$$\mathbf{C}\mathbf{v}_i = \lambda_i \mathbf{v}_i \quad \text{for} \quad i = 1, 2, \ldots, p,$$

where λ_i represents the eigenvalues and $\mathbf{v}_i$ are the corresponding eigenvectors. The eigenvectors represent the directions of maximum variance (principal components), and the eigenvalues represent the amount of variance explained by each component.

4. Selection of Principal Components

The eigenvectors (principal components) are sorted in descending order according to their eigenvalues. The first k eigenvectors corresponding to the largest k eigenvalues are selected, where k is the desired number of principal components.

5. Projection of Data

Finally, the original data is projected onto the selected principal components to form the reduced dataset:

$$\mathbf{X}_{\text{PCA}} = \mathbf{Z}\mathbf{V}_k,$$

where $\mathbf{V}_k$ is the matrix of the top k eigenvectors.

Explained Variance

The proportion of the total variance explained by the k principal components is calculated as:

$$\text{Explained Variance Ratio} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i}.$$

This value indicates how much information (variance) is retained by the selected components.

Applications of PCA

PCA is commonly applied in the following areas:

- **Dimensionality Reduction:** Reducing the number of features in large datasets to remove noise and improve computational efficiency.
- **Data Visualization:** Projecting high-dimensional data into two or three principal components for visualization.
- **Preprocessing:** Used as a preprocessing step in machine learning models to improve performance and reduce overfitting.

A.2 K-Fold Cross Validation

K-fold cross validation is a popular model evaluation technique in machine learning and statistics, used to assess the performance of a predictive model on a limited dataset. It involves partitioning the dataset into K distinct subsets, also known as *folds*. The main steps of K-fold cross validation are as follows:

1. **Data Splitting:** The dataset is randomly divided into K equal-sized (or nearly equal) folds. Each fold contains a portion of the data, ensuring that every sample is used for both training and validation during the process.

2. **Training and Testing:** For each iteration (total K iterations), one fold is used as the *test set*, and the remaining $K-1$ folds are combined to form the *training set*. The model is trained on the training set and then evaluated on the test set.

3. **Repeat Process:** The process is repeated K times, with each fold used exactly once as the test set. This ensures that the model's performance is tested on all the available data.

4. **Averaging the Results:** After completing all K iterations, the performance metrics (e.g., accuracy, precision, recall) obtained from each test set are averaged to provide an overall evaluation of the model's performance.

Advantages

- **More Reliable Evaluation:** K-fold cross validation provides a more reliable estimate of model performance compared to a simple train-test split, as the model is evaluated on different subsets of the data.

- **Efficient Use of Data:** Since all data points are used for both training and testing across different iterations, K-fold cross validation maximizes the use of the available data.

- **Reduced Overfitting:** By validating the model on multiple different test sets, K-fold cross validation helps to reduce the risk of overfitting to a particular subset of the data.

Choosing the Value of K

The choice of K is important in K-fold cross validation. A common choice is $K = 10$, which provides a good balance between bias and variance. In general:

- **Smaller K values (e.g., $K = 5$):** These may result in a more biased estimate but are computationally less expensive.

- **Larger K values (e.g., $K = 10$ or higher):** These provide a less biased estimate but require more computational resources, as the model is trained and tested more times.

Step-by-Step Process

Let $\mathcal{D}$ represent the entire dataset, with n data points:

$$\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid i = 1, 2, \ldots, n\}$$

where $\mathbf{x}_i \in \mathbb{R}^p$ are the features of the i-th sample, and $y_i \in \mathbb{R}$ is the corresponding label.

1. **Partitioning the Dataset:** The dataset $\mathcal{D}$ is randomly divided into K equal-sized (or nearly equal-sized) folds:

   $$\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K$$

   where each $\mathcal{D}_k$ represents a subset of the data such that:

   $$\mathcal{D}_i \cap \mathcal{D}_j = \emptyset \quad \text{for} \quad i \neq j \quad \text{and} \quad \bigcup_{k=1}^{K} \mathcal{D}_k = \mathcal{D}.$$

2. **Training and Testing:** For each fold $k \in \{1, 2, \ldots, K\}$, use $\mathcal{D}_k$ as the test set, and the remaining data $\mathcal{D}_{\text{train}}^{(k)}$ as the training set:

   $$\mathcal{D}_{\text{train}}^{(k)} = \mathcal{D} \setminus \mathcal{D}_k.$$

   Train the model $f^{(k)}$ on $\mathcal{D}_{\text{train}}^{(k)}$ and evaluate its performance on $\mathcal{D}_k$ by calculating an error metric, such as accuracy, mean squared error, etc.

3. **Evaluation:** Let $E^{(k)}$ represent the error (or evaluation metric) on fold k. This can be expressed as:

   $$E^{(k)} = \frac{1}{|\mathcal{D}_k|} \sum_{i \in \mathcal{D}_k} \mathcal{L}(f^{(k)}(\mathbf{x}_i), y_i),$$

   where $\mathcal{L}(\cdot, \cdot)$ is the loss function (e.g., squared error or 0-1 loss for classification), and $|\mathcal{D}_k|$ is the number of samples in fold k.

4. **Averaging the Results:** After repeating the training and testing process for all K folds, the final performance is computed as the average error across all folds:

   $$E_{\text{avg}} = \frac{1}{K} \sum_{k=1}^{K} E^{(k)}.$$

   This provides an unbiased estimate of the model's performance by considering all data points as part of both the training and the test sets.

Bias-Variance Tradeoff

The choice of K affects the bias and variance of the model's performance estimate:

- When K is small (e.g., $K = 5$), the test sets are large, which can lead to a more biased estimate but reduced variance.
- When K is large (e.g., $K = 10$), the test sets are smaller, which provides a less biased estimate but higher variance.

In the extreme case of $K = n$ (where n is the total number of samples), this method is called *Leave-One-Out Cross Validation* (LOOCV).

A.3 Linear Discriminant Classifier

For a binary classification problem with classes π_1 and π_2, the goal of Linear Discriminant Analysis (LDA) is to find a linear decision boundary that maximizes the separation between the classes.

Decision Rule

For the two-class problem (π_1 and π_2), the decision rule is to assign $\mathbf{x}$ to class π_1 if:

$$p(\pi_1 \mid \mathbf{x}) > p(\pi_2 \mid \mathbf{x})$$

This is equivalent to:

$$\frac{p(\pi_1)p(\mathbf{x} \mid \pi_1)}{p(\mathbf{x})} > \frac{p(\pi_2)p(\mathbf{x} \mid \pi_2)}{p(\mathbf{x})}$$

Simplifying the terms, we obtain the discriminant function:

$$\ln p(\pi_1 \mid \mathbf{x}) - \ln p(\pi_2 \mid \mathbf{x}) = \ln \frac{p(\pi_1)}{p(\pi_2)} + (\mathbf{x} - \mu_1)^T \Sigma^{-1} (\mu_1 - \mu_2) - \frac{1}{2} (\mu_1 - \mu_2)^T \Sigma^{-1} (\mu_1 - \mu_2)$$

Where:

- Σ is the pooled covariance matrix (assuming equal covariance for both classes).

The decision rule is:

$$\hat{\pi} = \begin{cases} \pi_1 & \text{if } (\mathbf{x} - \mu_1)^T \Sigma^{-1} (\mu_1 - \mu_2) > \frac{1}{2} (\mu_1 - \mu_2)^T \Sigma^{-1} (\mu_1 - \mu_2) - \ln \frac{p(\pi_2)}{p(\pi_1)} \\ \pi_2 & \text{otherwise} \end{cases}$$

This represents the linear decision boundary between the two classes.

A.4 Quadratic Discriminant Classifier

In the Quadratic Discriminant Analysis (QDA), we do not assume that the classes have the same covariance matrix, unlike Linear Discriminant Analysis (LDA). This allows the decision boundary to be quadratic instead of linear. For a binary classification problem with classes π_1 and π_2, the goal is to find the decision rule that maximizes the posterior probability.

Decision Rule

For the two-class problem (π_1 and π_2), the decision rule is to assign $\mathbf{x}$ to class π_1 if:

$$p(\pi_1 \mid \mathbf{x}) > p(\pi_2 \mid \mathbf{x})$$

This is equivalent to:

$$\frac{p(\pi_1)p(\mathbf{x} \mid \pi_1)}{p(\mathbf{x})} > \frac{p(\pi_2)p(\mathbf{x} \mid \pi_2)}{p(\mathbf{x})}$$

Simplifying the terms, we obtain the discriminant function:

$$\ln p(\pi_1 \mid \mathbf{x}) - \ln p(\pi_2 \mid \mathbf{x}) = \ln \frac{p(\pi_1)}{p(\pi_2)} + (\mathbf{x} - \mu_1)^T \Sigma_1^{-1} (\mathbf{x} - \mu_1) - \frac{1}{2} (\mu_1 - \mu_2)^T \Sigma_1^{-1} (\mu_1 - \mu_2)$$

Where:

- Σ_1 and Σ_2 are the covariance matrices of class π_1 and π_2, respectively.

The decision rule is:

$$\hat{\pi} = \begin{cases} \pi_1 & \text{if } \ln \frac{p(\pi_1)}{p(\pi_2)} + (\mathbf{x} - \mu_1)^T \Sigma_1^{-1} (\mathbf{x} - \mu_1) - \frac{1}{2} (\mu_1 - \mu_2)^T \Sigma_1^{-1} (\mu_1 - \mu_2) \\ \pi_2 & \text{otherwise} \end{cases}$$

This represents the quadratic decision boundary between the two classes.

A.5 Naive Bayes Classifier

For a two-class classification problem, we denote the classes as π_1 and π_2, and the observed feature vector as $\underset{\sim}{x} = (x_1, x_2, \ldots, x_n)$, where x_i is the i-th feature of the observation.

Bayes' Theorem: The posterior probability of a class π_k given the feature vector $\underset{\sim}{x}$ is:

$$p(\pi_k|\underset{\sim}{x}) = \frac{p(\underset{\sim}{x}|\pi_k)p(\pi_k)}{p(\underset{\sim}{x})}$$

where:

- $p(\pi_k \mid \underset{\sim}{x})$ is the posterior probability of class π_k given $\underset{\sim}{x}$,
- $p(\underset{\sim}{x} \mid \pi_k)$ is the likelihood of $\underset{\sim}{x}$ given class π_k,
- $p(\pi_k)$ is the prior probability of class π_k,
- $p(\underset{\sim}{x})$ is the evidence, or the total probability of observing $\underset{\sim}{x}$, which is constant for all classes and can be ignored in the decision rule.

Naive Bayes Assumption:

In Naive Bayes, we assume that the features $X_1, X_2, \ldots, X_n$ are conditionally independent given the class label π_k. This simplifies the likelihood term as follows:

$$p(\underset{\sim}{x}|\pi_k) = \prod_{i=1}^{n} p(x_i|\pi_k)$$

Thus, the posterior probability becomes:

$$p(\pi_k|\underset{\sim}{x}) = \frac{\prod_{i=1}^{n} p(x_i|\pi_k)p(\pi_k)}{p(\underset{\sim}{x})}$$

Decision Rule:
In a two-class problem with classes π_1 and π_2, the decision rule assigns $\underset{\sim}{x}$ to class π_1 if:

$$p(\pi_1|\underset{\sim}{x}) > p(\pi_2|\underset{\sim}{x})$$

Using Bayes' Theorem, this simplifies to:

$$p(\underset{\sim}{x}|\pi_1) \cdot p(\pi_1) > p(\underset{\sim}{x}|\pi_2) \cdot p(\pi_2)$$

where:

- $p(\underset{\sim}{x} \mid \pi_1) = \prod_{i=1}^{n} p(x_i \mid \pi_1)$,
- $p(\underset{\sim}{x} \mid \pi_2) = \prod_{i=1}^{n} p(x_i \mid \pi_2)$.

This decision rule is based on the assumption of conditional independence of features and maximizes the posterior probability for each class.

A.6 Oversampling

Oversampling is a technique used to address *class imbalance* in datasets, where one class has significantly more samples than the other(s). This imbalance can lead to biased machine learning models that perform poorly on the minority class. Oversampling works by increasing the number of samples in the minority class, either by **replicating existing samples** or **generating new synthetic samples**. This technique helps the model to give equal importance to both classes and improve classification performance.

Random Oversampling (Replicating Samples)

Random oversampling involves duplicating samples from the minority class until the number of samples in the minority class is increased to match the majority class.

Example: If the majority class has 1000 samples and the minority class has 100 samples, random oversampling will randomly pick samples from the minority class and duplicate them until the minority class reaches 1000 samples.

Pros:

- Simple to implement.
- Fast.

Cons:

- May lead to overfitting, as the model sees the same minority class samples multiple times.
- Reduces diversity in the data.

Synthetic Minority Over-sampling Technique (SMOTE)

SMOTE generates *synthetic samples* rather than replicating existing ones. For each sample in the minority class, SMOTE selects its k nearest neighbors and creates new samples by interpolating between the original sample and its neighbors.

Pros:

- Creates more varied samples.
- Helps the model generalize better than simple replication.

Cons:

- May introduce noise if synthetic samples are generated improperly.
- May not work well for very sparse minority classes.

Process of Oversampling

1. **Identify the imbalance**: First, check the class distribution in the dataset. If the dataset is imbalanced, oversampling is applied to the minority class.
2. **Choose an oversampling method**: Depending on the dataset, choose whether to apply simple replication or more advanced methods like SMOTE etc.
3. **Apply oversampling**: Based on the chosen method, replicate or generate synthetic samples to increase the number of minority class instances.
4. **Train the model**: Train your machine learning model on the newly balanced dataset.
5. **Evaluate the model**: After training, evaluate the model to assess whether it performs better on the minority class, using metrics like precision, recall, F1-score and AUC.

A.7 Some Performance Metrics

Precision

Precision is the proportion of true positive predictions among all instances that were predicted as positive. It answers the question: "Of all the instances that the model predicted as positive, how many were actually positive?"

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

When to focus on Precision:

- Precision is useful when the cost of *false positives* is high. For example:
  - In medical diagnostics, a false positive might lead to unnecessary treatments.

  - In spam detection, marking a legitimate email as spam can cause important messages to be missed.

Pros:

- Focuses on reducing false positives.
- Important when positive predictions need to be highly reliable.

Cons:

- Does not account for false negatives.
- In imbalanced datasets, precision alone may not give a complete picture of model performance.

F1 Score

The F1 Score is the harmonic mean of **Precision** and **Recall**. It balances the two metrics, giving equal importance to both. The F1 Score is useful when we want to balance the trade-off between precision and recall, especially in imbalanced datasets.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Where **Recall** is the proportion of true positives among all actual positive instances:

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

When to focus on F1 Score:

- F1 score is useful when you care about both *Precision* (avoiding false positives) and *Recall* (capturing all positive instances).
- It's especially helpful when you have imbalanced classes.

Pros:

- Provides a single metric to evaluate model performance.
- Balances the trade-off between precision and recall, making it useful in imbalanced datasets.

Cons:

- Does not give insight into false positives or false negatives separately.
- Less interpretable than precision or recall alone.

Example:

Consider the following confusion matrix for a binary classification task:

	Predicted Positive	Predicted Negative
Actual Positive	50 (True Positives)	10 (False Negatives)
Actual Negative	5 (False Positives)	100 (True Negatives)

Precision:

$$\text{Precision} = \frac{50}{50+5} = \frac{50}{55} = 0.909 \text{ (or } 90.9\%)$$

Recall:

$$\text{Recall} = \frac{50}{50+10} = \frac{50}{60} = 0.833 \text{ (or } 83.3\%)$$

F1 Score:

$$\text{F1 Score} = 2 \times \frac{0.909 \times 0.833}{0.909 + 0.833} = 2 \times \frac{0.758}{1.742} = 0.869 \text{ (or } 86.9\%)$$

A.8 K-Nearest Neighbors (K-NN) Classifier

The K-Nearest Neighbors (KNN) algorithm is a supervised learning algorithm used for both classification and regression tasks. The algorithm works by finding the K-nearest data points to a given query point and using them to predict the output.

Algorithm Overview

The K-NN algorithm can be summarized in the following steps:

1. **Choose the number of neighbors (K):** The first step in the K-NN algorithm is to choose the value of K, which represents the number of nearest neighbors to consider.

2. **Calculate distances:** For each query point, calculate the distance between the query point and all the points in the training dataset. Common distance metrics include:

   - **Euclidean distance:**
   $$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

   - **Manhattan distance:**
   $$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

   - **Minkowski distance:**
   $$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

3. **Select the K nearest neighbors:** Based on the calculated distances, identify the k nearest training data points to the query point.

4. **Assign the class:** For classification tasks, the majority class among the K-nearest neighbors is assigned to the query point. For regression tasks, the predicted value is the average of the k-nearest neighbors' values.

Choosing the Value of K

The choice of K plays a crucial role in the performance of the k-NN algorithm:

- If K is too small, the model may be overly sensitive to noise (high variance).
- If K is too large, the model may become overly simplistic (high bias).

Typically, the optimal value of K is determined by cross-validation.

Advantages and Disadvantages

Advantages:

- Simple and intuitive.
- No explicit training phase (instance-based learning).
- Effective for small to medium-sized datasets.

Disadvantages:

- Computationally expensive during prediction, as it requires calculating the distance between the query point and all training points.
- Sensitive to the choice of distance metric and the value of K.
- Not suitable for high-dimensional datasets (curse of dimensionality).

A.9 Support Vector Machine

A Support Vector Machine (SVM) is a supervised machine learning model used primarily for classification tasks. The main ideas behind SVM are as follows:

Mathematical Formulation

Given a set of training data points $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^d$ are the feature vectors, and $y_i \in \{-1, +1\}$ are the class labels, the objective is to find a hyperplane that separates the two classes with the maximum margin.

The equation of the hyperplane is:

$$\mathbf{w}^T\mathbf{x} + b = 0$$

where $\mathbf{w} \in \mathbb{R}^d$ is the normal vector to the hyperplane, and b is the bias term.

Optimization Problem

The objective is to maximize the margin between the two classes while ensuring correct classification of the data points. The margin is given by $\frac{1}{\|\mathbf{w}\|}$, so the optimization problem is:

$$\underset{\mathbf{w},b}{\text{maximize}} \quad \frac{1}{\|\mathbf{w}\|}$$

subject to the constraints:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad \forall i = 1, \ldots, n$$

This problem can be rewritten as a constrained optimization problem:

$$\text{minimize} \quad \frac{1}{2}\|\mathbf{w}\|^2$$

subject to:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad \forall i = 1, \ldots, n$$

Dual Formulation

The primal problem can be converted into a dual problem using Lagrange multipliers. The Lagrangian function is:

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i \left[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 \right]$$

where $\alpha_i \geq 0$ are the Lagrange multipliers. The dual formulation of the problem is obtained by minimizing the Lagrangian with respect to $\mathbf{w}$ and b and maximizing with respect to α_i. The dual optimization problem becomes:

$$\underset{\alpha_i}{\text{maximize}} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

subject to:

$$\sum_{i=1}^{n} \alpha_i y_i = 0, \quad \alpha_i \geq 0, \quad \forall i$$

The solution to this dual problem provides the optimal values of α_i, and the weight vector $\mathbf{w}$ can be computed as:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

Support Vectors

The data points that lie on the margin boundaries are called support vectors. These are the data points for which $\alpha_i > 0$. Only support vectors are needed to define the optimal hyperplane, making SVM a memory-efficient classifier.

A.10 Decision Trees

Gini Index

The Gini index is a measure of impurity or purity used in decision trees to select the best split. It is calculated as:

$$Gini(D) = 1 - \sum_{i=1}^{k} p_i^2$$

where:

- D is the dataset,
- k is the number of classes,
- p_i is the proportion of instances of class i in the dataset.

For a split with two subsets D_1 and D_2, the Gini index is calculated as the weighted average of the Gini indices of each subset:

$$Gini_{\text{split}}(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2)$$

Entropy

Entropy measures the uncertainty or impurity in a dataset and is used in the information gain calculation. The entropy of a dataset D is given by:

$$Entropy(D) = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

where p_i is the proportion of instances of class i in the dataset.
For a split into two subsets D_1 and D_2, the entropy of the split is:

$$Entropy_{\text{split}}(D) = \frac{|D_1|}{|D|} Entropy(D_1) + \frac{|D_2|}{|D|} Entropy(D_2)$$

CART Algorithm (Classification and Regression Trees)

CART is a decision tree algorithm that can be used for both classification and regression tasks. It uses Gini index for classification and mean squared error (MSE) for regression. The tree is built by recursively splitting the data at each node to minimize the impurity (Gini index) or variance.

For a classification problem:

$$\text{Best Split} = \arg\min_{A,v} Gini_{\text{split}}(D)$$

For a regression problem, the best split minimizes the variance within each subset:

$$\text{Best Split} = \arg\min_{A,v} \sum_{i \in D_1} (y_i - \bar{y}_1)^2 + \sum_{i \in D_2} (y_i - \bar{y}_2)^2$$

where:

- y_i is the target value of instance i,
- $\bar{y}_1$ and $\bar{y}_2$ are the mean target values in subsets D_1 and D_2, respectively.

Random Forest is an ensemble learning method that constructs a collection of decision trees at training time and outputs the class that is the majority vote of the classes (for classification) or average (for regression) of the individual trees.

Let $T_1, T_2, \ldots, T_k$ represent the k decision trees in the forest. For an input x, the output of the random forest is:

$$\hat{y}_{\text{RF}}(x) = \frac{1}{k} \sum_{i=1}^{k} \hat{y}_i(x)$$

where $\hat{y}_i(x)$ is the prediction of tree T_i for the input x.

A.11 Random Forest

Each tree in a Random Forest is typically trained using a subset of the training data selected via bootstrap sampling (i.e., random sampling with replacement). For each tree T_i, the model is trained on a random subset $D_i \subset D$, where D is the original training set.

Additionally, at each node of the decision tree, only a random subset of features is considered for splitting, which introduces further randomness. This reduces correlation among the trees in the forest and helps prevent overfitting.

Out-of-Bag Error Estimate

In Random Forests, one of the key advantages is the use of out-of-bag (OOB) samples for error estimation. Each tree is trained on a bootstrap sample, meaning about one-third of the data is not used to train each tree. These out-of-bag samples can be used to estimate the performance of the model. The OOB error rate is the proportion of times a prediction for an out-of-bag sample is incorrect:

$$\text{OOB Error Rate} = \frac{1}{N_{\text{OOB}}} \sum_{i=1}^{N_{\text{OOB}}} \mathbf{1}(\hat{y}_i \neq y_i)$$

where: - N_{OOB} is the number of out-of-bag samples, - $\mathbf{1}(\hat{y}_i \neq y_i)$ is an indicator function that is 1 if the predicted label $\hat{y}_i$ is different from the true label y_i, and 0 otherwise.

Feature Importance in Random Forests

Feature importance in Random Forests is often determined using the decrease in impurity, i.e., how much each feature contributes to reducing the Gini impurity or variance in the case of regression. A feature's importance is computed by averaging the decrease in impurity over all trees in which the feature is used for splitting.

$$\text{Importance}(f) = \frac{1}{k} \sum_{i=1}^{k} \Delta\text{Impurity}_i(f)$$

where $\Delta\text{Impurity}_i(f)$ represents the decrease in impurity when feature f is used in tree i.

Random Forest reduces the variance component of the error by introducing randomness into the model (bootstrap sampling and random feature selection), while maintaining relatively low bias.

A.12 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of biological neural networks, particularly the brain. They consist of interconnected nodes (neurons) organized into layers that process information collectively. Let's go through the theory and mathematical formulation of ANNs in detail.

Structure of Artificial Neural Networks

Neuron (Node): A neuron is the fundamental unit of an ANN. It takes inputs, applies weights to them, sums them, adds a bias term, and then passes this through an activation function to produce an output.

Layers: Neurons are arranged in layers:

- **Input Layer:** The first layer receives the input data.
- **Hidden Layers:** Intermediate layers that process the data from the input layer.
- **Output Layer:** The final layer that produces the output of the network.

Architecture: The arrangement of neurons and layers is termed the architecture of the network. Examples include feedforward networks, recurrent neural networks, and convolutional neural networks.

Mathematical Formulation of a Neuron

For a single neuron, the output y is computed as follows:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where:

- x_i: Input feature values
- w_i: Weights associated with each input
- b: Bias term
- $f(\cdot)$: Activation function

Activation Functions

Activation functions introduce non-linearity into the network, allowing it to learn complex patterns. Common activation functions include:

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Hyperbolic Tangent (Tanh):** $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- **ReLU (Rectified Linear Unit):** $\text{ReLU}(x) = \max(0, x)$
- **Leaky ReLU:** Leaky $\text{ReLU}(x) = \max(\alpha x, x)$, where α is a small constant
- **Softmax (used in output layer for classification):** $\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Feedforward Process

In a feedforward neural network, information flows from the input layer to the output layer through the hidden layers:

- **Weighted Sum Calculation:** For each neuron in a hidden or output layer, the weighted sum of inputs and bias is calculated.
- **Activation Function Application:** The weighted sum is passed through the activation function.
- **Layer-by-Layer Calculation:** This process is repeated for each layer until the output layer is reached.

Mathematically, for a layer l, if $a^{(l-1)}$ represents the activations from the previous layer, the output $a^{(l)}$ for the current layer l is:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = f(z^{(l)})$$

where:

- $W^{(l)}$: Weight matrix for layer l
- $b^{(l)}$: Bias vector for layer l
- $z^{(l)}$: Pre-activation output for layer l
- $a^{(l)}$: Post-activation output for layer l

Loss Function

The loss function measures the difference between the predicted output and the actual output, guiding the network's training process. Common loss functions include:

- **Mean Squared Error (MSE)** for regression:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- **Cross-Entropy Loss** for classification:

$$\text{Cross-Entropy} = -\sum_{i} y_i \log(\hat{y}_i)$$

where:

- y_i: True label for the i-th instance
- $\hat{y}_i$: Predicted output for the i-th instance
- N: Number of instances in the dataset

Backpropagation Algorithm

Backpropagation is an algorithm used to minimize the loss function by adjusting weights and biases in the network.

Steps of Backpropagation

1. **Forward Pass:** Compute the output and loss for a given input.
2. **Compute the Gradient:** Calculate the gradient of the loss with respect to each parameter (weights and biases).
3. **Weight Update:** Adjust the weights and biases in the direction that reduces the loss using an optimization algorithm, such as Gradient Descent.

For weight w in layer l, the update rule is:

$$w_{ij}^{(l)} := w_{ij}^{(l)} - \eta \frac{\partial L}{\partial w_{ij}^{(l)}}$$

where:

- L: Loss function
- η: Learning rate (a small constant that controls the step size)

Gradient Calculation: Using the chain rule, backpropagation calculates the partial derivatives for each weight w and bias b:

- **For the output layer:**
$$\delta^{(L)} = \nabla_a L \circ f'(z^{(L)})$$
- **For previous layers l (backpropagating the error):**
$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \circ f'(z^{(l)})$$

Here, $\delta^{(l)}$ represents the error term for layer l, and $\circ$ denotes the Hadamard (element-wise) product.

Gradient Descent and Optimization

Gradient Descent is a common optimization method for minimizing the loss function. Variants include:

- **Stochastic Gradient Descent (SGD):** Updates weights after each training example.
- **Mini-Batch Gradient Descent:** Divides data into small batches and updates weights for each batch.
- **Momentum, RMSprop, and Adam:** Advanced optimizers that adjust the learning rate and incorporate momentum to improve convergence speed and stability.

Adam Optimizer Update Rules:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta L(\theta)$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla_\theta L(\theta)^2$$
$$\theta = \theta - \eta \frac{m_t}{\sqrt{v_t} + \epsilon}$$

where:

- m_t and v_t: Moving averages of gradients and squared gradients
- β_1, β_2: Hyperparameters controlling decay rates
- ϵ: Small constant to prevent division by zero

Regularization Techniques

Regularization prevents overfitting by constraining the model's complexity:

- **L2 Regularization:** Adds a penalty proportional to the sum of squared weights:
$$L_{reg} = L + \frac{\lambda}{2} \sum_{i=1}^{n} w_i^2$$
- **Dropout:** Randomly drops neurons during training to prevent co-adaptation of neurons.

B Supplementary R and Python codes

Exploratory Data Analysis

```r
setwd("~/Desktop/Parkinson")
library(ggplot2)
library(reshape2)
dat <- read.csv("Parkinsson disease.csv")
# Renaming the columns for ease of
colnames(dat) <- c("name","avg_fre", "max_fre", "min_fre", "var_fre1",
 ↪  "var_fre2",
                   "var_fre3", "var_fre4", "var_fre5", "var_amp1", "var_amp2",
                   "var_amp3", "var_amp4", "var_amp5", "var_amp6", "NHR", "HNR",
                   "status", "RPDE", "DFA", "spread1", "spread2", "D2", "PPE")
# Pie Chart
class.proportion <- c(mean(dat$status) * 100, 100 - (mean(dat$status) * 100))
labels <- c("Parkinson", "Healthy")
# Custom colors
colors <- c("lightblue", "lightgreen")
# Creating labels with percentages
labels_with_percent <- paste(labels, round(class.proportion, 1), "%")
# Plotting pie chart with the updated labels
pie(class.proportion, labels = labels_with_percent, col = colors)
legend("topright", labels_with_percent,
       fill = colors)
dim(dat) # 195  X 24: 195 obs of 24 variables
head(dat, 3)
# Checking for missing values
any(is.na(dat)) # FALSE
# Finding numeric and non numeric columns
sapply(dat, is.numeric) # only name col is FALSE
# Information of the dataset
str(dat) # Status col is binary of 0 and 1 (hence integer)
# The rest col's are all numeric except name col which is character type
# Convert name column into categorical type
dat$name <- as.factor(dat$name)
# Statistical Summary
summary(dat[-1])
# Duplicate Entries
sum(duplicated(dat[-1])) # sum is 0
# Highly positively correlated columns
# criteria : abs(cor_matrix) >0.9
# (spread1, PPE), (var_amp5,var_amp1,var_amp3,var_amp6,var_amp2), (var_freq2,
 ↪  var_freq4, var_freq1, var_freq3, var_freq5) )
# (NHR,var_freq1, var_freq3, var_freq5)
cor_matrix <- round(cor(dat[,-c(1,18)]), 1)
# Melt the correlation matrix into long format for ggplot2
```

```r
melted_cor_matrix <- melt(cor_matrix)
# Heatmap
ggplot(data = melted_cor_matrix, aes(Var1, Var2, fill = value)) +
  geom_tile(color = "white") +
  scale_fill_gradient2(low = "blue", high = "red", mid = "white",
                       midpoint = 0, limit = c(-1,1), space = "Lab",
                       name="Correlation") + theme_minimal() +
  theme(axis.text.x = element_text(angle = 90, vjust = 1,
                                   size = 10, hjust = 1)) +
  coord_fixed() +
  geom_text(aes(Var1, Var2, label = value), color = "black", size = 3) +
  labs(x = "Features", y = "Features")
df_long <- melt(dat[, -1], id.vars = "status")  # Exclude the 'name' column
# Boxplot
ggplot(df_long, aes(x = as.factor(status), y = value, fill = as.factor(status))) +
  geom_boxplot() +
  facet_wrap(~ variable, scales = "free", ncol = 4) +  # Create a 5x5 grid
  theme_minimal() +
  labs(x = "Status", y = "Value", fill = "Status") +  # Adding fill legend title
  theme(axis.text.x = element_text(angle = 0, hjust = 1),
        strip.text = element_text(size = 12),  # Increase facet label size if needed
        legend.position = "bottom",
        panel.grid.major = element_blank(),  # Remove major grid lines
        panel.grid.minor = element_blank())  # Remove minor grid lines
scale_fill_brewer(palette = "Set2")  # Change color palette as needed
# Pairplot
dat$status <- as.factor(dat$status)
pairs(dat[, c('var_fre1', 'var_fre2', 'var_fre3', 'var_fre4', 'var_fre5')],
      col = dat$status,  # Color by 'status'
      pch = 19)
write.csv(dat,"parkinson.csv", row.names = FALSE)
```

Principal Component Analysis

```r
setwd("~/Desktop/Parkinson")
library(scatterplot3d)
library(reshape2)
library(aplpack)
dat <- read.csv("parkinson.csv")
df <- dat[, -c(1,18)]
for (j in 1:dim(df)[2])
{
  if(is.numeric(df[ ,j]) == FALSE) df[ ,j] <- as.numeric(df[ ,j])
}
std.df <- scale(df, center = TRUE, scale = TRUE)
```

```r
covariancematrix <- cov(std.df)
eigenvaluesvector <- eigen(covariancematrix)
sortindice <- order(eigenvaluesvector$values, decreasing = TRUE)
sorteigenvalues <- eigenvaluesvector$values[sortindice]
sorteigenvector <- eigenvaluesvector$vectors[ ,sortindice]
par(mfrow=c(1,1), mar = c(5, 4, 2, 1))
# First plot: Eigenvalues
plot(eigenvaluesvector$values, type = "b", xlab = "Number of Principal
↪  Components",
     ylab = "Eigenvalue", pch=16, col="darkblue", lwd=2, cex.lab =1.2)
abline(v = 8, col = "red", lty=2, lwd=2)
# Second plot: Cumulative Variance Explained
plot(cumsum(eigenvaluesvector$values) / sum(eigenvaluesvector$values), type =
↪  "b",
     xlab = "Number of Principal Components", ylab = "Cumulative Variance",
     pch=16, col="darkorange", lwd=2, cex.lab = 1.2)
abline(v = 8, col = "red", lty=2, lwd=2)
# From the plot, 8 PCs explains 95.76% of total variability.
n.pcomponent <- 8
selectedcomponents <-sorteigenvector[ ,1:n.pcomponent]
# Reduced data (Dimensionality Reduction)
reduceddata <- std.df %*% selectedcomponents # selected eigenvector
# Importance of Components
pca_result <- prcomp(df, scale. = TRUE)
summary(pca_result)
# Projection into visualisable plane
tags.reduceddata <- cbind(dat[18], reduceddata[,1], reduceddata[,2],
↪  reduceddata[,3] )
colnames(tags.reduceddata ) <- c("Status", "PC1", "PC2", "PC3")
colors <- ifelse(dat[,18] == 0, "blue", "red")
s3d <- scatterplot3d(tags.reduceddata$PC1,tags.reduceddata$PC2,
↪  tags.reduceddata$PC3,
                     xlab = "PC1", ylab = "PC2", zlab = "PC3",
                     color = colors,
                     pch = 16, angle=60)
# 45, 135, 225, and 315
legend("topleft",
       legend = c("Healthy", "Parkinson"),  # Labels for legend
       col = c("blue", "red"),
       pch = 16,
       title = "Status", cex=1.2)
# Outlier Detection
## 2-D Plane
par(mfrow = c(1,1))
par(mar =  c(5.1, 4.1, 4.1, 2.1))
plot(x = reduceddata[,1], y = reduceddata[,2], xlab = "PC1", ylab = "PC2",
     col="blue", pch = 16, cex.lab = 1)
```

```r
tags <- dat$name
outlier <- c('phon_R01_S35_4', 'phon_R01_S24_4', 'phon_R01_S24_6',
             'phon_R01_S35_6', 'phon_R01_S35_7')
outlier_indices <- which(tags %in% outlier)
text(x = reduceddata[outlier_indices, 1] + 0.5,
     y = reduceddata[outlier_indices, 2],
     labels = tags[outlier_indices],
     pos = 3, cex = 0.6, col = "red")
## Chernoff faces
faces(reduceddata, face.type = 1,
      labels = dat$name, cex = 0.6)
# Outliers removed after detection
outlier <- c('phon_R01_S35_4', 'phon_R01_S24_4', 'phon_R01_S24_6',
             'phon_R01_S35_6', 'phon_R01_S35_7')
df.outliers.removed <- dat[!dat$name %in% outlier, ]
write.csv(df.outliers.removed,"parkinson_no_outlier.csv", row.names = FALSE)
```

Parametric Models

1. LDA, QDA, Naive Bayes

```r
library(caret)
library(MASS)
library(tidyverse)
library(e1071)
# Load and prepare the data
parkinson <- read.csv("no_outlier_data.csv")[, -1]
parkinson$status <- as.factor(parkinson$status) # levels: 0, 1
# Separate features and target variable
X <- parkinson[, !(colnames(parkinson) %in% c("status"))]
y <- parkinson$status
data_balanced <- ovun.sample(status ~ ., data = parkinson[,-1],
                             method = "over", N = max(table(y)) * 2)$data
folds <- createFolds(data_balanced$status, k = 5) # Randomization Step
# Define Bayes Model functions
kde.prior.estimate <- function(x.vec) {
  group0.x.vec <- 1
  group1.x.vec <- 1
  for (i in 1:length(x.vec)) {
    group0.x.vec <- group0.x.vec * density(group0.train.scale[[i]], from =
    ↪  x.vec[i], to = x.vec[i], n = 1)$y
    group1.x.vec <- group1.x.vec * density(group1.train.scale[[i]], from =
    ↪  x.vec[i], to = x.vec[i], n = 1)$y
  }
  return(as.vector(prior * c(group0.x.vec, group1.x.vec))) # Apply priors
}
predict.fun <- function(feature.mat) {
```

```r
  n <- nrow(feature.mat)
  y.predict <- c()
  for (i in 1:n) {
    x.vec <- unlist(unname(feature.mat[i, ]))
    y.predict <- c(y.predict, which.max(kde.prior.estimate(x.vec)) - 1)
  }
  return(y.predict)
}
# Initialize matrix to store accuracy for each fold
accuracy <- matrix(0, nrow = 3, ncol = 5)
# Estimate prior from training data
prior <- as.vector(prop.table(table(data_balanced$status)))
# Loop over models
model <- 1:3
for (m in model) {
  for (i in 1:5) {
    # Split data into training and testing sets for the i-th fold
    test_indices <- folds[[i]]
    test <- data_balanced[test_indices, ]
    train <- data_balanced[-test_indices, ]
    # Split features and response
    feature.train <- train[, colnames(train) != "status"]
    response.train <- train[, colnames(train) == "status"]
    feature.test <- test[, colnames(test) != "status"]
    response.test <- test[, colnames(test) == "status"]
    # Scale the features
    train.scale <- scale(feature.train)
    means <- attr(train.scale, "scaled:center")
    sds <- attr(train.scale, "scaled:scale")
    train.scale <- as.data.frame(train.scale)
    test.scale <- scale(feature.test, center = means, scale = sds)
    test.scale <- as.data.frame(test.scale)
    # Define new variables for Bayes Model
    group0.train <- train[train$status == 0, !(colnames(train) == "status")]
    group1.train <- train[train$status == 1, !(colnames(train) == "status")]
    group0.train.scale <- as.data.frame(scale(group0.train, center = means, scale
    ↪  = sds))
    group1.train.scale <- as.data.frame(scale(group1.train, center = means, scale
    ↪  = sds))
    # Train LDA and QDA models on training data
    lda.model <- lda(status ~ ., data = cbind(train.scale, status =
    ↪  response.train),
                     prior = prior)
    qda.model <- qda(status ~ ., data = cbind(train.scale, status =
    ↪  response.train),
                     prior = prior)
    # Make predictions for each model on the test set
```

```
    lda.test.predictions <- predict(lda.model, test.scale)$class
    qda.test.predictions <- predict(qda.model, test.scale)$class
    bayes.test.predictions <- predict.fun(test.scale) # Bayes model predictions
    # Store model accuracies
    if (m == 1) {
      accuracy[m, i] <- mean(lda.test.predictions == response.test)
      print(paste("LDA Fold",i))
      print(table(Predicted = lda.test.predictions, Actual = response.test ))
    } else if (m == 2) {
      accuracy[m, i] <- mean(qda.test.predictions == response.test)
      print(paste("QDA Fold",i))
      print(table(Predicted = qda.test.predictions, Actual = response.test ))
    } else {
      accuracy[m, i] <- mean(bayes.test.predictions == response.test)
      print(paste("Bayes Fold",i))
      print(table(Predicted = bayes.test.predictions, Actual = response.test ))
    }
  }
}
# Print the accuracy for each fold and the average accuracy
print(accuracy*100)
accuracy_means <- rowMeans(accuracy) * 100
print(accuracy_means)
##############################################################################
# Choosing priors by grid search
library(ROSE)
library(caret)
library(class)
library(MASS)
library(tidyverse)
library(e1071)
# Load and prepare the data
parkinson <- read.csv("no_outlier_data.csv")[, -1]
parkinson$status <- as.factor(parkinson$status) # levels: 0, 1
# Separate features and target variable
X <- parkinson[, !(colnames(parkinson) %in% c("status"))]
y <- parkinson$status
data_balanced <- ovun.sample(status ~ ., data = parkinson[,-1],
                             method = "over", N = max(table(y)) * 2)$data
folds <- createFolds(data_balanced$status, k = 5) # Randomization Step
# Define Bayes Model functions
kde.prior.estimate <- function(x.vec, prior) {
  group0.x.vec <- 1
  group1.x.vec <- 1
  for (i in 1:length(x.vec)) {
    group0.x.vec <- group0.x.vec * density(group0.train.scale[[i]], from =
    ↪  x.vec[i], to = x.vec[i], n = 1)$y
```

```
    group1.x.vec <- group1.x.vec * density(group1.train.scale[[i]], from =
    ↪  x.vec[i], to = x.vec[i], n = 1)$y
  }
  return(as.vector(prior * c(group0.x.vec, group1.x.vec))) # Apply priors
}
predict.fun <- function(feature.mat) {
  n <- nrow(feature.mat)
  y.predict <- c()
  for (i in 1:n) {
    x.vec <- unlist(unname(feature.mat[i, ]))
    y.predict <- c(y.predict, which.max(kde.prior.estimate(x.vec, prior)) - 1)
  }
  return(y.predict)
}
# Initialize matrix to store accuracy for each fold
accuracy <- array(0, dim=c(3, 5, 10))
prior.vec <- seq(0.01, 0.98, length.out = 10)
model <- 1:3
# Loop over models
for (p in 1:length(prior.vec))
{
  print(p)
  prior <- c(1-prior.vec[p],prior.vec[p])
  for (m in model) {
    for (i in 1:5) {
      # Split data into training and testing sets for the i-th fold
      test_indices <- folds[[i]]
      test <- data_balanced[test_indices, ]
      train <- data_balanced[-test_indices, ]
      # Split features and response
      feature.train <- train[, colnames(train) != "status"]
      response.train <- train[, colnames(train) == "status"]
      feature.test <- test[, colnames(test) != "status"]
      response.test <- test[, colnames(test) == "status"]
      # Scale the features
      train.scale <- scale(feature.train)
      means <- attr(train.scale, "scaled:center")
      sds <- attr(train.scale, "scaled:scale")
      train.scale <- as.data.frame(train.scale)
      test.scale <- scale(feature.test, center = means, scale = sds)
      test.scale <- as.data.frame(test.scale)
      # Define new variables for Bayes Model
      group0.train <- train[train$status == 0, !(colnames(train) == "status")]
      group1.train <- train[train$status == 1, !(colnames(train) == "status")]
      group0.train.scale <- as.data.frame(scale(group0.train, center = means,
      ↪  scale = sds))
      group1.train.scale <- as.data.frame(scale(group1.train, center = means,
      ↪  scale = sds))
```

```r

      # Train LDA and QDA models on training data
      lda.model <- lda(status ~ ., data = cbind(train.scale, status =
      ↪  response.train),
                       prior = prior)
      qda.model <- qda(status ~ ., data = cbind(train.scale, status =
      ↪  response.train),
                       prior = prior)
      # Make predictions for each model on the test set
      lda.test.predictions <- predict(lda.model, test.scale)$class
      qda.test.predictions <- predict(qda.model, test.scale)$class
      bayes.test.predictions <- predict.fun(test.scale) # Bayes model predictions
      # Store model accuracies
      if (m == 1) {
        accuracy[m, i, p] <- mean(lda.test.predictions == response.test)
      } else if (m == 2) {
        accuracy[m, i, p] <- mean(qda.test.predictions == response.test)
      } else {
        accuracy[m, i, p] <- mean(bayes.test.predictions == response.test)
      }
    }
  }
}
accuracy <- accuracy * 100
plot(prior.vec, 100 - apply(accuracy, c(1, 3), mean)[1,], type = "l", col =
↪  "red",
     xlab = "Prior probability",
     ylab = "Misclassification Error Rate expressed in percentage (%)",
     ylim = c(0,100), lwd = 2)
lines(prior.vec, 100 - apply(accuracy, c(1, 3), mean)[2,], type = "l", col =
↪  "blue", lwd = 2)
lines(prior.vec, 100 - apply(accuracy, c(1, 3), mean)[3,], type = "l", col =
↪  "green", lwd = 2)
# Add a legend
legend("topright", legend = c("LDA", "QDA", "Naive Bayes"),
       col = c("red", "blue", "green"), lty = 1, lwd = 2)
```

Non Parametric Models

1. K Nearest Neighbor Classifier

```r
# Load and prepare the data
parkinson <- read.csv("parkinson_no_outlier.csv")[, -1]
parkinson$status <- as.factor(parkinson$status) # levels: 0, 1
table(parkinson$status)
# Separate features and target variable
X <- parkinson[, !(colnames(parkinson) %in% c("status"))]
```

```r
y <- parkinson$status
data_balanced <- ovun.sample(status ~ ., data = parkinson[,-1],
                             method = "over", N = max(table(y)) * 2)$data
folds <- createFolds(data_balanced$status, k = 5) # Randomization Step
table(data_balanced$status)
# Initialize matrix to store accuracy for each fold
accuracy <- matrix(0, nrow = 5, ncol = 1)
k.vec <- 1:1
for (k in 1:length(k.vec)){
  for (i in 1:5) {
    # Split data into training and testing sets for the i-th fold
    test_indices <- folds[[i]]
    test <- data_balanced[test_indices, ]
    train <- data_balanced[-test_indices, ]
    # Split features and response
    feature.train <- train[, colnames(train) != "status"]
    response.train <- train[, colnames(train) == "status"]
    feature.test <- test[, colnames(test) != "status"]
    response.test <- test[, colnames(test) == "status"]
    # Scale the features
    train.scale <- scale(feature.train)
    means <- attr(train.scale, "scaled:center")
    sds <- attr(train.scale, "scaled:scale")
    train.scale <- as.data.frame(train.scale)
    test.scale <- scale(feature.test, center = means, scale = sds)
    test.scale <- as.data.frame(test.scale)

    # Make knn predictions on the test set
    knn.test.predictions <- knn(train = train.scale,
                                test = test.scale,
                                cl = train$status,
                                k = k)
    # Store model accuracies
    accuracy[i,k] <- mean(knn.test.predictions==response.test)
    print(table(Predicted = knn.test.predictions, Actual = response.test ))
  }
}
# Print the accuracy for each fold and the average accuracy
print(accuracy*100)
accuracy_means <- colMeans(accuracy) * 100
print(accuracy_means)
plot(x = k.vec, y = 100-accuracy_means, type = "l", col="red",
     lwd = 2, ylab = "Misclassification Error Rate expressed in (%)",
     xlab = "Number of Neighbors (K)")
```

2. Support Vector Machine

```r
# Load necessary libraries
library(e1071)
library(caret)
# Load the dataset
# Update with the actual file path
data <- read.csv("parkinson_no_outlier.csv")
data <- data[,-1]
# View the structure of the dataset
str(data)
# Ensure 'status' is a factor
data$status <- as.factor(data$status)
# Split the dataset into training and testing sets
set.seed(123)  # For reproducibility
trainIndex <- createDataPartition(data$status, p = 0.8, list = FALSE)
train_data <- data[trainIndex, ]
test_data <- data[-trainIndex, ]
# Train the SVM model
svm_model <- svm(status ~ ., data = train_data, kernel = "radial", cost = 1,
 ↪  gamma = 0.1)
# Print the model summary
summary(svm_model)
# Make predictions on the test set
predictions <- predict(svm_model, newdata = test_data)
# Evaluate the model performance
conf_matrix <- confusionMatrix(predictions, test_data$status)
print(conf_matrix)
```

3. Decision Trees

```r
# Load necessary libraries
library(rpart)
library(caret)
# Load the dataset
data <- read.csv("parkinson_no_outlier.csv")
data <- data[,-1]
# Ensure 'status' is a factor
data$status <- as.factor(data$status)
# Split the dataset into training and testing sets
set.seed(123)  # For reproducibility
trainIndex <- createDataPartition(data$status, p = 0.8, list = FALSE)
train_data <- data[trainIndex, ]
test_data <- data[-trainIndex, ]
# Train the Decision Tree model
dt_model <- rpart(status ~ ., data = train_data, method = "class")
# Print the model summary
print(dt_model)
```

```r
# Save the plot as a PNG file
png("decision_tree_plot.png", width = 800, height = 600)
plot(dt_model)
text(dt_model, use.n = TRUE, all = TRUE, cex = 0.8)
dev.off()
# Make predictions on the test set
dt_predictions <- predict(dt_model, newdata = test_data, type = "class")
# Evaluate the model performance
dt_conf_matrix <- confusionMatrix(dt_predictions, test_data$status)
print(dt_conf_matrix)
```

4. Artificial Neural Network (python)

```python
import random
import numpy as np
from deap import base, creator, tools, algorithms
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
def evaluate_ann(individual):
    model = Sequential()
    for i in range(individual[0]):  # individual[0] is the number of hidden layers
        model.add(Dense(individual[1], activation='relu'))  # individual[1] is the number of neurons in each layer
    model.add(Dense(1, activation='sigmoid'))  # Output layer for binary classification
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    model.fit(X_resampled, y_resampled, epochs=10, batch_size=32, verbose=0)
    _, accuracy = model.evaluate(X_resampled, y_resampled, verbose=0)
    return accuracy,
# Setup Genetic Algorithm
creator.create("FitnessMax", base.Fitness, weights=(1.0,))  # Maximize accuracy
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()
toolbox.register("num_layers", random.randint, 1, 5)  # Random number of layers (1 to 5)
toolbox.register("num_neurons", random.randint, 5, 50)  # Random number of neurons (5 to 50)
toolbox.register("individual", tools.initCycle, creator.Individual, (toolbox.num_layers, toolbox.num_neurons), n=1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("mate", tools.cxUniform, indpb=0.5)  # Specify indpb
toolbox.register("mutate", tools.mutFlipBit, indpb=0.1)  # Mutate genes with 10% probability
```

```python
toolbox.register("select", tools.selBest)
toolbox.register("evaluate", evaluate_ann)
# Parameters
population_size = 10
generations = 5
# Run Genetic Algorithm
pop = toolbox.population(n=population_size)
fitnesses = list(map(toolbox.evaluate, pop))
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit
best_accuracies = []
for gen in range(generations):
    print(f"Generation {gen + 1}")
    offspring = toolbox.select(pop, len(pop))
    offspring = list(map(toolbox.clone, offspring))
    # Apply crossover and mutation
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if random.random() < 0.5:  # Crossover probability
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values
    for mutant in offspring:
        if random.random() < 0.2:  # Mutation probability
            toolbox.mutate(mutant)
            del mutant.fitness.values
    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = list(map(toolbox.evaluate, invalid_ind))
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit
    pop[:] = offspring
    best_accuracy = max(ind.fitness.values[0] for ind in pop)
    best_accuracies.append(best_accuracy)
    print(f"Best Accuracy in Generation {gen + 1}: {best_accuracy}")
```

5. Random Forest

```r
# Load necessary libraries
library(randomForest)
library(caret)
# Load the dataset
data <- read.csv("parkinson_no_outlier.csv")
data <- data[,-1]
# Ensure 'status' is a factor
data$status <- as.factor(data$status)
# Split the dataset into training and testing sets
set.seed(123)  # For reproducibility
```

```r
trainIndex <- createDataPartition(data$status, p = 0.8, list = FALSE)
train_data <- data[trainIndex, ]
test_data <- data[-trainIndex, ]
# Train the Random Forest model
rf_model <- randomForest(status ~ ., data = train_data, ntree = 500, mtry = 3,
↪  importance = TRUE)
# Print the model summary
print(rf_model)
# Save plot as PNG
png("variable_importance.png", width = 800, height = 600)
varImpPlot(rf_model)
dev.off()
# Make predictions on the test set
rf_predictions <- predict(rf_model, newdata = test_data)
# Evaluate the model performance
rf_conf_matrix <- confusionMatrix(rf_predictions, test_data$status)
print(rf_conf_matrix)
```