Department of Artificial Intelligence

Amrita School of Engineering

Coimbatore- 641 112, Tamil Nadu, India

Parkinson's Disease Detection Using Machine Learning

A project submitted
in partial fulfilment of the requirements for the degree of
Masters of Technology in Artificial Intelligence

By

Zarak Jahan
(CB.SC.P2AIE23005)

Supervised by:
Dr. SenthilKumar T

Nov, 2023

MACHINE LEARNING

Table 1. Team Members

Roll No	Name	Official Email id	Contribution	Photo
CB.SC.P2AIE23005	Zarak Jahan	Cb.sc.p2aie23005@cb.students.amrita.edu	All the sections are worked on by me only.	

GitHub URL of the project page:

https://github.com/Oyezeejay/Parkinson-s-Disease-Detection-using-Machine-Learning

Kaggle URL of the dataset page:

https://www.kaggle.com/datasets/vikasukani/parkinsons-disease-data-set

Colab Link:

https://colab.research.google.com/drive/1Bs59Xcxe203QFDg5_GLbCRTR-Hem3q_y?usp=sharing

https://colab.research.google.com/drive/1ApM9Rw6Za1FolNbPgNuGqGOxMz7j-vkl?usp=drive_link

SECTION 1

1.1 Application Name: Parkinson's Disease Detection Using Machine Learning

1.2Provide a set of analytical questions

1.2.1 Why:

Motivation: What motivates the use of machine learning for Parkinson's disease detection? How can machine learning contribute to early detection and intervention in Parkinson's disease cases?

Advantages: Why choose machine learning over traditional diagnostic methods for Parkinson's disease? What advantages does machine learning offer in terms of accuracy, efficiency, and scalability?

Impact: How can machine learning in Parkinson's disease detection impact patient outcomes and quality of life? What are the potential societal and healthcare system benefits?

1.2.2 What:

Features: What features or biomarkers are commonly used in machine learning models for Parkinson's disease detection? How are these features related to the physiological aspects of the disease?

Algorithms: What machine learning algorithms have shown promise in detecting Parkinson's disease? How do different algorithms compare in terms of sensitivity, specificity, and computational efficiency?

Data: What types of data are typically used in training machine learning models for Parkinson's disease detection? How do factors such as data volume, quality, and diversity impact model performance?

1.2.3 How:

Preprocessing: How is data preprocessed before feeding it into machine learning models for Parkinson's disease detection? What preprocessing techniques are commonly employed to enhance the quality of input data?

Model Selection: How do researchers and practitioners select the most suitable machine learning model for Parkinson's disease detection? What considerations are taken into account when choosing between different algorithms?

Validation: How are machine learning models validated in the context of Parkinson's disease detection? What metrics and methodologies are commonly used to assess the performance and generalizability of the models?

1.3 Provide a set of questions for prediction

1.3.1 Data Preparation:

Feature Selection: Which features or biomarkers are considered most relevant for predicting Parkinson's disease? How are these features selected, and what is their biological significance?

Data Sources: What are the primary sources of data used for training and testing predictive models in Parkinson's disease detection? How do researchers ensure the representativeness and diversity of the dataset?

Missing Data: How is missing data handled in the dataset? What imputation techniques or strategies are employed to address gaps in the data?

1.3.2 Model Development:

Algorithm Choice: What machine learning algorithms are commonly employed for predicting Parkinson's disease? How does the choice of algorithm impact the predictive accuracy and interpretability of the model?

Model Complexity: How do researchers balance model complexity in Parkinson's disease prediction? What trade-offs exist between simple models and more complex ones in terms of interpretability and generalizability?

Hyperparameter Tuning: How is hyperparameter tuning performed to optimize the performance of predictive models for Parkinson's disease? What methods are used to avoid overfitting or underfitting?

1.3.3 Model Evaluation:

Performance Metrics: Which performance metrics are most relevant for evaluating the accuracy and reliability of predictive models for Parkinson's disease? How do sensitivity, specificity, and area under the curve (AUC) contribute to model assessment?

Cross-Validation: How is cross-validation applied to assess the robustness of predictive models? What strategies are employed to ensure that the model performs well on unseen data?

1.3.4 Clinical Applicability:

Early Detection: To what extent can predictive models aid in the early detection of Parkinson's disease? How does early prediction impact treatment outcomes and patient care?

Integration with Clinical Practices: How can predictive models be integrated into clinical workflows for Parkinson's disease diagnosis and monitoring? What challenges and considerations arise when implementing machine learning predictions in real-world healthcare settings?

Patient Privacy and Ethical Considerations: What measures are taken to ensure patient privacy when utilizing predictive models for Parkinson's disease detection? How are ethical considerations addressed in the development and deployment of these models?

1.4Technologies Used:

Table 2. Technologies Used

Editor	Google Colab
Language	Python

1.5 Why is this Application required?

Detecting Parkinson's disease early is crucial for several reasons:

Early Intervention and Treatment: Early detection allows for prompt intervention and treatment. While there is currently no cure for Parkinson's disease, certain medications and therapies can help manage symptoms and improve the quality of life for patients. Initiating treatment at an early stage may slow down the progression of the disease and alleviate some symptoms.

Improved Quality of Life: Parkinson's disease can have a significant impact on a person's daily life, affecting motor skills, balance, and coordination. Early detection enables individuals to receive appropriate medical care and therapies that can help manage symptoms, maintain mobility, and enhance overall well-being.

Clinical Trials and Research: Early detection is essential for recruiting participants for clinical trials and research studies. These studies aim to understand the underlying causes of Parkinson's disease, develop new treatment strategies, and eventually find a cure. Identifying individuals with Parkinson's at an early stage allows researchers to study the disease progression and evaluate the effectiveness of potential interventions.

Care Planning: Early diagnosis allows patients and their families to plan for the future and make informed decisions about care and lifestyle adjustments. It provides an opportunity for

healthcare professionals to work with patients in developing personalized care plans that address their specific needs and challenges.

Symptom Management: Parkinson's disease is characterized by a range of symptoms, including tremors, stiffness, and difficulty with balance and coordination. Early detection allows for the implementation of targeted interventions to manage these symptoms, such as physical therapy, occupational therapy, and medication adjustments.

Educational and Support Resources: Early diagnosis provides individuals with Parkinson's disease and their families access to educational resources and support networks. Understanding the condition and connecting with others who are facing similar challenges can be valuable in coping with the emotional and practical aspects of living with Parkinson's.

1.6 List of similar applications:

Table 3. Applications in the Market

Application Name	URL
iPrognosis mobile app	https://play.google.com/store/apps/details?id=com.iprognosis.gdatasuite&hl=en_US&pli=1
Smartphone App could help advance the early detection of Parkinson's disease and severe COVID-19	https://www.news-medical.net/news/20221005/Smartphone-App-could-help-advance-the-early-detection-of-Parkinsons-disease-and-severe-COVID-19.aspx

1.7 What is unique in your project:

I have implemented multiple classification algorithms like logistic regression, KNN Random Forest, Gaussian NB, and SVC and evaluated all their accuracies in the detection of Parkinson's disease when new input data is given as shown in Figure 1.

Figure 1. Predictive Result for Parkinson's Disease Detection

SECTION 2

2.1Conference Papers

Table 4. Conference Papers

S.No	Paper Name	Conference Name	Citation
1	Machine Learning Approaches for Detection and Diagnosis of Parkinson's Disease-A Review	2021 7th International Conference on Advanced Computing & Communication Systems (ICACCS)	I. Nissar, W. A. Mir, Izharuddin, and T. A. Shaikh, "Machine learning approaches for detection and diagnosis of Parkinson's disease - A review," in 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS), 2021.
2	Parkinson's Detection Using Machine Learning	Proceedings of the Fifth International Conference on Intelligent Computing and Control Systems (ICICCS 2021)	S. Tadse, M. Jain, and P. Chandankhede, "Parkinson's detection using machine learning," in 2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS), 2021
3	A Deep Learning Based Method for Parkinson's Disease Detection Using Dynamic Features of Speech	IEEE Access (Volume: 9)	C. Quan, K. Ren, and Z. Luo, "A deep learning based method for Parkinson's disease detection using dynamic features of speech," IEEE Access, vol. 9, pp. 10239–10252, 2021.

2.2 Journal Details

Table 5. Journal Papers

S.No	Paper Name	Journal Name	Citation
1	A hybrid system for Parkinson's disease diagnosis using machine learning techniques	International Journal of Speech Technology (2022)	R. Lamba, T. Gulati, H. F. Alharbi, and A. Jain, "A hybrid system for Parkinson's disease diagnosis using machine learning techniques," Int. J. Speech Technol., vol. 25, no. 3, pp. 583–593, 2022.
2	A Comparative Study of Existing Machine Learning Approaches for Parkinson's Disease Detection	IETE Journal of Research	G. Pahuja and T. N. Nagabhushan, "A comparative study of existing machine learning approaches for Parkinson's disease detection," *IETE J. Res.*, vol. 67, no. 1, pp. 4–14, 2021.
3	Machine Learning for the Diagnosis of Parkinson's Disease: A Review of Literature	Front. Aging Neurosci., 06 May 2021	Mei J, Desrosiers C and Frasnelli J (2021) Machine Learning for the Diagnosis of Parkinson's Disease: A Review of Literature. Front. Aging Neurosci. 13:633752. doi: 10.3389/fnagi.2021.633752
4	Machine learning models for Parkinson's disease detection and stage classification based on spatial-temporal gait parameters	www.elsevier.com/locate/gaitpost	M. I. A. S. N. Ferreira, F. A. Barbieri, V. C. Moreno, T. Penedo, and J. M. R. S. Tavares, "Machine learning models for Parkinson's disease detection and stage classification based on spatial-temporal gait parameters," Gait Posture, vol. 98, pp. 49–55, 2022.
5	Machine Learning Approaches in Parkinson's Disease	Current Medicinal Chemistry, Volume 28, Number 32, 2021	A. Landolfi et al., "Machine Learning approaches in Parkinson's disease," Curr. Med. Chem., vol. 28, no. 32, pp. 6548–6568, 2021.

6	Mining imaging and clinical data with machine learning approaches for the diagnosis and early detection of Parkinson's disease	www.nature.com/npjparkd	J. Zhang, "Mining imaging and clinical data with machine learning approaches for the diagnosis and early detection of Parkinson's disease," NPJ Parkinsons Dis., vol. 8, no. 1, 2022.
7	Efficient detection of Parkinson's disease using deep learning techniques over medical data	Expert Syst., vol. 39, no. 3, 2022.	L. Sahu, R. Sharma, I. Sahu, M. Das, B. Sahu, and R. Kumar, "Efficient detection of Parkinson's disease using deep learning techniques over medical data," Expert Syst., vol. 39, no. 3, 2022.
8	An adaptive intelligent diagnostic system to predict early-stage of Parkinson's disease using two-stage dimension reduction with genetically optimized light gum algorithm	Neural Computing and Applications	J. Dhar, "An adaptive intelligent diagnostic system to predict the early stage of Parkinson's disease using two-stage dimension reduction with genetically optimized light gum algorithm," Neural Comput. Appl., vol. 34, no. 6, pp. 4567–4593, 2022.
9	Vocal Feature Extraction-Based Artificial Intelligent Model for Parkinson's Disease Detection	MDPI Open Access Journal	M. Hoq, M. N. Uddin, and S.-B. Park, "Vocal feature extraction-based artificial intelligent model for Parkinson's disease detection," Diagnostics (Basel), vol. 11, no. 6, p. 1076, 2021.
10	Optimized ANFIS Model Using Hybrid Metaheuristic Algorithms for Parkinson's Disease	IEEE Access (Volume: 8)	I. M. El-Hasnony, S. I. Barakat, and R. R. Mostafa, "Optimized ANFIS model using hybrid metaheuristic algorithms for Parkinson's disease prediction in IoT environment," IEEE

	Prediction in IoT Environment		Access, vol. 8, pp. 119252–119270, 2020.

2.3 Model Diagram:

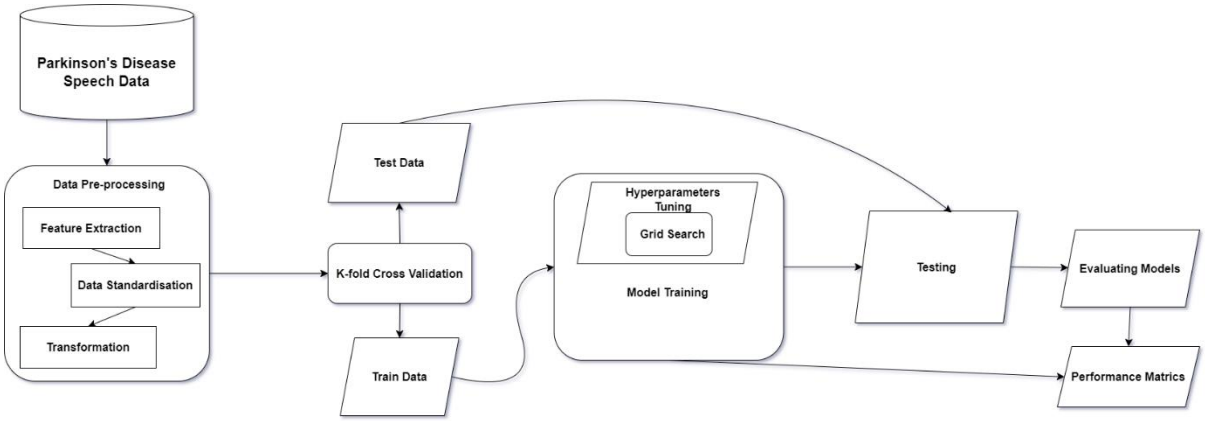


Figure 2. Model Diagram

In the realm of machine learning, data standardization emerges as a crucial technique to ensure a fair playing field for diverse features. Its primary goal is to prevent attributes with larger numerical values from exerting undue influence on the model compared to those with smaller values. Z-score standardization, a specific approach, transforms attribute values into standard scores, thereby centering the data around a mean of 0 and a standard deviation of 1. This normalization process sets the stage for unbiased model training and ensures that all features contribute equally to the learning process. To robustly assess the model's performance, I employ K-fold Cross Validation. This technique involves partitioning the dataset into k subsets, iteratively training the model on k-1 subsets, and testing it on the remaining subset. For model training, I explore various classification algorithms such as Support Vector Machines (SVM), Logistic Regression, and k-nearest Neighbors (KNN). Each algorithm brings unique strengths to the table, contributing to a well-rounded and versatile predictive model. To gauge the effectiveness of the trained model, I utilize a range of performance metrics, including accuracy, recall, and validation accuracy. This multifaceted evaluation provides a comprehensive understanding of the model's proficiency in different aspects of classification.

SECTION 3

3.1 Dataset Description

This dataset comprises a range of biomedical voice measurements from 31 people, 23 with Parkinson's disease (PD). Each column in the table is a particular voice measure, and each row corresponds to one of 195 voice recordings from these individuals ("name" column). The main aim of the data is to discriminate healthy people from those with PD, according to the "status" column which is set to 0 for healthy and 1 for PD.

Author:

Little, Max. (2008). Parkinson's. UCI Machine Learning Repository. https://doi.org/10.24432/C59C74.

3.2 Preprocessing

In machine learning, preprocessing is a crucial step involving the transformation and refinement of raw data before it is input into a model. The process encompasses various techniques, including cleaning data by handling missing values and removing duplicates, transforming features through standardization or normalization, encoding categorical variables, and conducting dimensionality reduction. Additionally, preprocessing involves addressing imbalanced data, splitting the dataset into training and testing sets, handling outliers, and, in the case of natural language processing, performing text-specific tasks like tokenization and stemming. Efficient preprocessing enhances the model's performance by mitigating challenges such as overfitting and biases, ultimately contributing to the creation of accurate and robust machine-learning models.

3.2.1 Descriptive and Statistical Feature Analysis

https://colab.research.google.com/drive/1ApM9Rw6Za1FolNbPgNuGqGOxMz7j-vkl#scrollTo=-vVNoGxhwyJ0

Df.head() – It prints the first 5 rows of all the features.

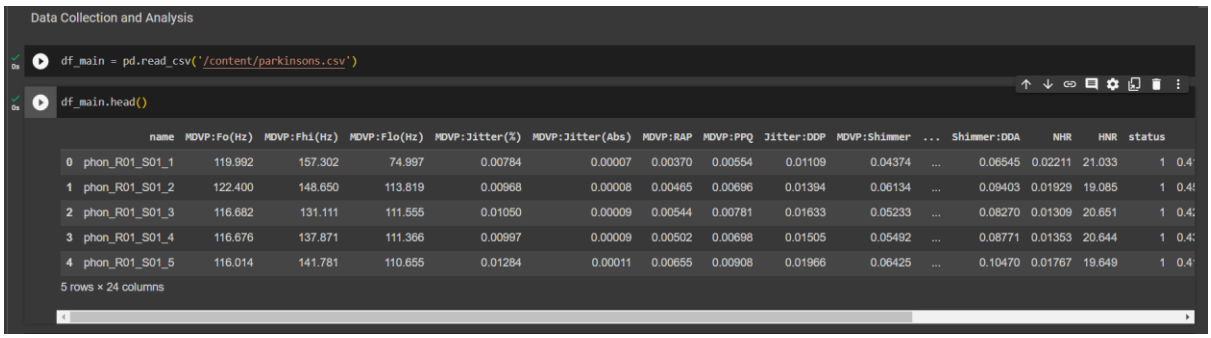


Figure 3. Df.head()

Df.shape - It gives the number of rows and columns in the dataset.

```
[4]  df_main.shape #Gives no. of row and columns in the dataset

     (195, 24)
```

Figure 4. Df.shape

Df.info() – It gives the number of values of each feature, displays if it has null values and also gives the type of Dtype.

```
df_main.info() #non-null values means no value is missing in the 195 rows of a column

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 195 entries, 0 to 194
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   name              195 non-null    object
 1   MDVP:Fo(Hz)       195 non-null    float64
 2   MDVP:Fhi(Hz)      195 non-null    float64
 3   MDVP:Flo(Hz)      195 non-null    float64
 4   MDVP:Jitter(%)    195 non-null    float64
 5   MDVP:Jitter(Abs)  195 non-null    float64
 6   MDVP:RAP          195 non-null    float64
 7   MDVP:PPQ          195 non-null    float64
 8   Jitter:DDP        195 non-null    float64
 9   MDVP:Shimmer      195 non-null    float64
 10  MDVP:Shimmer(dB)  195 non-null    float64
 11  Shimmer:APQ3      195 non-null    float64
 12  Shimmer:APQ5      195 non-null    float64
 13  MDVP:APQ          195 non-null    float64
 14  Shimmer:DDA       195 non-null    float64
 15  NHR               195 non-null    float64
 16  HNR               195 non-null    float64
 17  status            195 non-null    int64
 18  RPDE              195 non-null    float64
 19  DFA               195 non-null    float64
 20  spread1           195 non-null    float64
 21  spread2           195 non-null    float64
 22  D2                195 non-null    float64
 23  PPE               195 non-null    float64
dtypes: float64(22), int64(1), object(1)
memory usage: 36.7+ KB
```

Figure 5. Df.info()

Df.isnull().sum() – It check if the sum of all the null values for each feature in the dataset.

```
df_main.isnull().sum() #checking if the dataset have any missing values
```

```
name              0
MDVP:Fo(Hz)       0
MDVP:Fhi(Hz)      0
MDVP:Flo(Hz)      0
MDVP:Jitter(%)    0
MDVP:Jitter(Abs)  0
MDVP:RAP          0
MDVP:PPQ          0
Jitter:DDP        0
MDVP:Shimmer      0
MDVP:Shimmer(dB)  0
Shimmer:APQ3      0
Shimmer:APQ5      0
MDVP:APQ          0
Shimmer:DDA       0
NHR               0
HNR               0
status            0
RPDE              0
DFA               0
spread1           0
spread2           0
D2                0
PPE               0
dtype: int64
```

Figure 6. Df.isnull().sum()

Df.describe() – It gives the different statistical measures of the data.

```
# Statistical measures of the data
df_main.describe()
```

	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)	MDVP:RAP	MDVP:PPQ	Jitter:DDP	MDVP:Shimmer	MDVP:Shimmer(dB)	...	Shimmer:DDA	NHR	
count	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	...	195.000000	195.000000	195.000
mean	154.228641	197.104918	116.324631	0.006220	0.000044	0.003306	0.003446	0.009920	0.029709	0.282251	...	0.046993	0.024847	21.885
std	41.390065	91.491548	43.521413	0.004848	0.000035	0.002968	0.002759	0.008903	0.018857	0.194877	...	0.030459	0.040418	4.425
min	88.333000	102.145000	65.476000	0.001680	0.000007	0.000680	0.000920	0.002040	0.009540	0.085000	...	0.013640	0.000650	8.441
25%	117.572000	134.862500	84.291000	0.003460	0.000020	0.001660	0.001860	0.004985	0.016505	0.148500	...	0.024735	0.005925	19.198
50%	148.790000	175.829000	104.315000	0.004940	0.000030	0.002500	0.002690	0.007490	0.022970	0.221000	...	0.038360	0.011660	22.085
75%	182.769000	224.205500	140.018500	0.007365	0.000060	0.003835	0.003955	0.011505	0.037885	0.350000	...	0.060795	0.025640	25.075
max	260.105000	592.030000	239.170000	0.033160	0.000260	0.021440	0.019580	0.064330	0.119080	1.302000	...	0.169420	0.314820	33.047

8 rows × 23 columns

Figure 7. Df.describe()

Df[‘target’].value_counts() – It prints out the number of values corresponding to one category and same for the other categories.

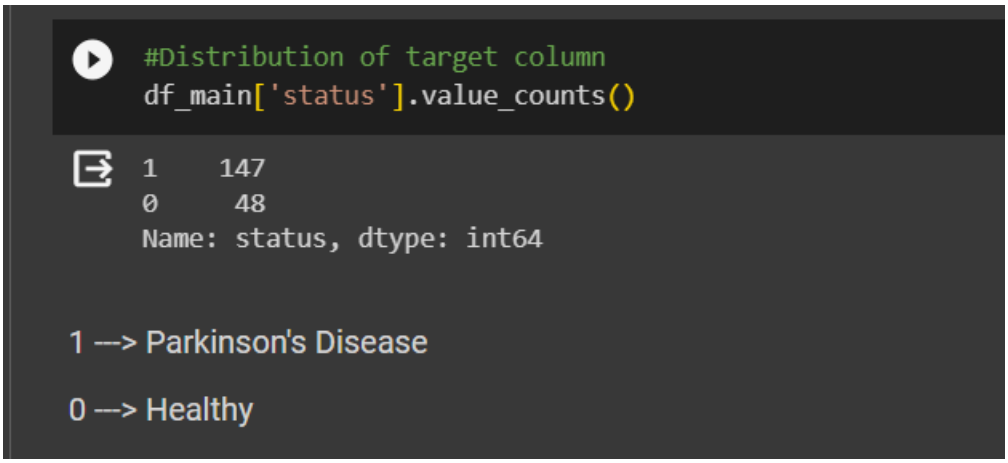


Figure 8. Df[‘target’].value_counts()

Df.groupby(‘status’).mean() – It gives the mean of all the variables for each categorical value of the target features.

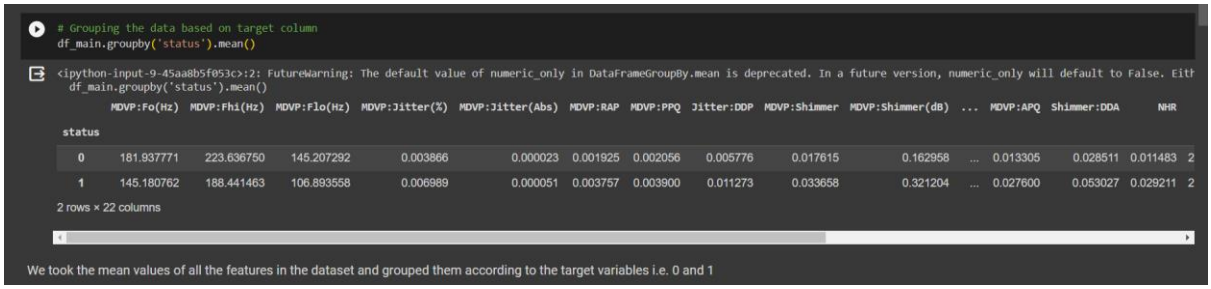


Figure 9. Df.groupby(‘status’).mean()

Df.drop_duplicates() – It deletes all the duplicate in the data.

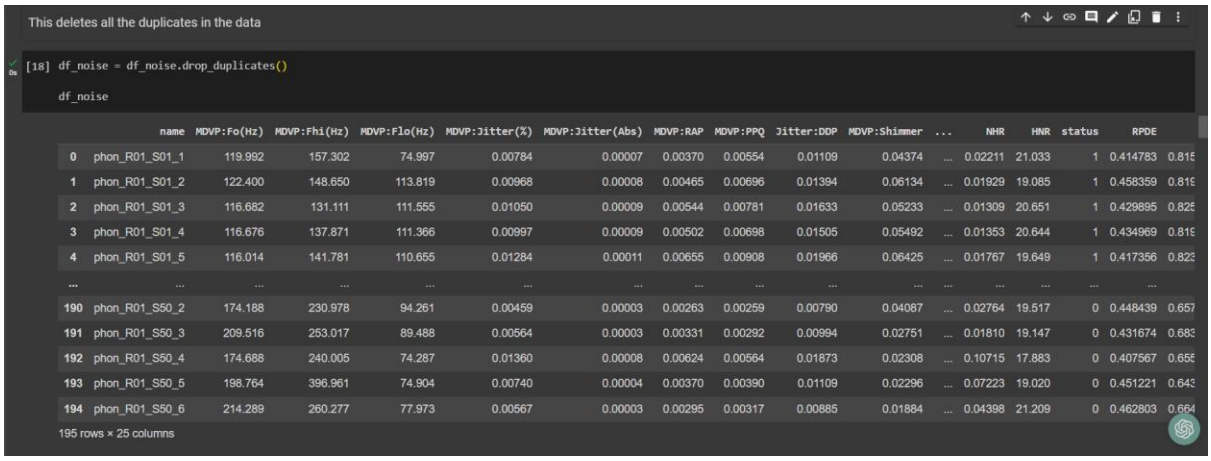
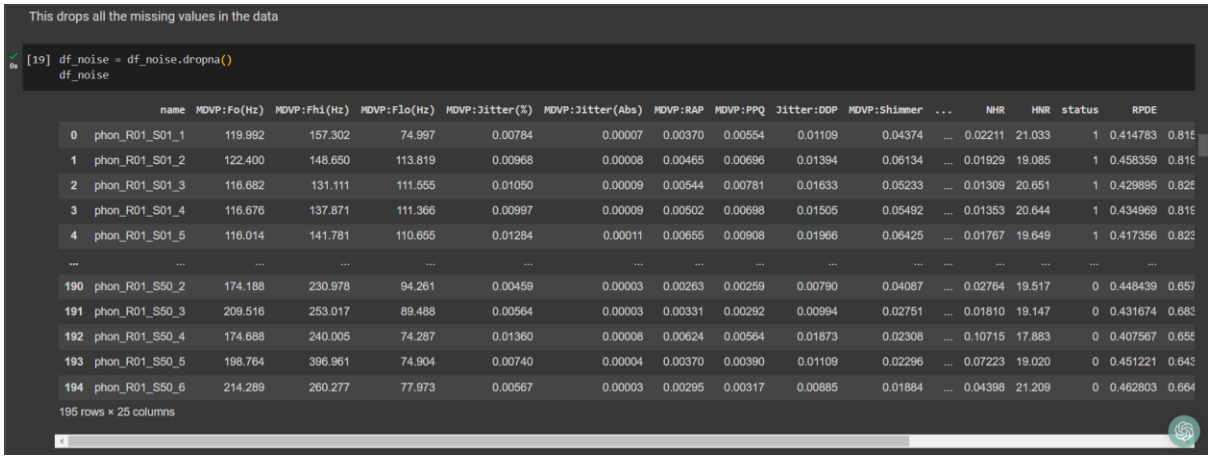


Figure 10. Df.drop_duplicates()

Df.dropna() – This deletes all the cells with missing values in the data.



This drops all the missing values in the data

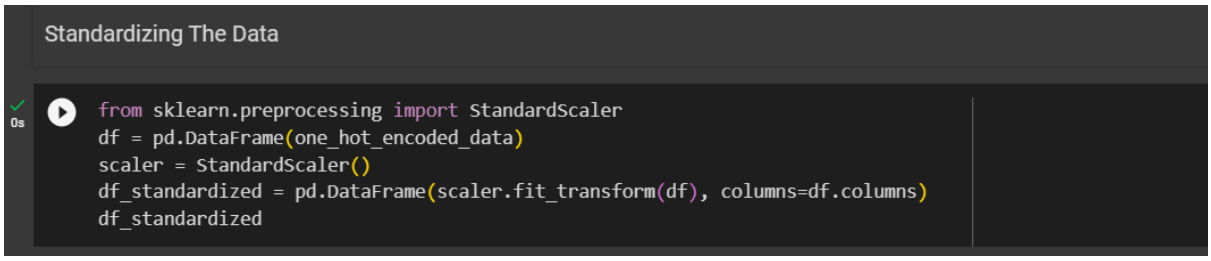
```
[19] df_noise = df_noise.dropna()
     df_noise
```

	name	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)	MDVP:RAP	MDVP:PPQ	Jitter:DDP	MDVP:Shimmer	...	NHR	HNR	status	RPDE	
0	phon_R01_S01_1	119.992	157.302	74.997	0.00784	0.00007	0.00370	0.00554	0.01109	0.04374	...	0.02211	21.033	1	0.414783	0.815
1	phon_R01_S01_2	122.400	148.650	113.819	0.00968	0.00008	0.00465	0.00696	0.01394	0.06134	...	0.01929	19.085	1	0.458359	0.819
2	phon_R01_S01_3	116.682	131.111	111.555	0.01050	0.00009	0.00544	0.00781	0.01633	0.05233	...	0.01309	20.651	1	0.429895	0.825
3	phon_R01_S01_4	116.676	137.871	111.366	0.00997	0.00009	0.00502	0.00698	0.01505	0.05492	...	0.01353	20.644	1	0.434969	0.819
4	phon_R01_S01_5	116.014	141.781	110.655	0.01284	0.00011	0.00655	0.00908	0.01966	0.06425	...	0.01767	19.649	1	0.417356	0.823
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
190	phon_R01_S50_2	174.188	230.978	94.261	0.00459	0.00003	0.00263	0.00259	0.00790	0.04087	...	0.02764	19.517	0	0.448439	0.657
191	phon_R01_S50_3	209.516	253.017	89.488	0.00564	0.00003	0.00331	0.00292	0.00994	0.02751	...	0.01810	19.147	0	0.431674	0.683
192	phon_R01_S50_4	174.688	240.005	74.287	0.01360	0.00008	0.00624	0.00564	0.01873	0.02308	...	0.10715	17.883	0	0.407567	0.655
193	phon_R01_S50_5	198.764	396.961	74.904	0.00740	0.00004	0.00370	0.00390	0.01109	0.02296	...	0.07223	19.020	0	0.451221	0.643
194	phon_R01_S50_6	214.289	260.277	77.973	0.00567	0.00003	0.00295	0.00317	0.00885	0.01884	...	0.04398	21.209	0	0.462803	0.664

195 rows × 25 columns

Figure 11. Df.dropna()

StandardScalar()-



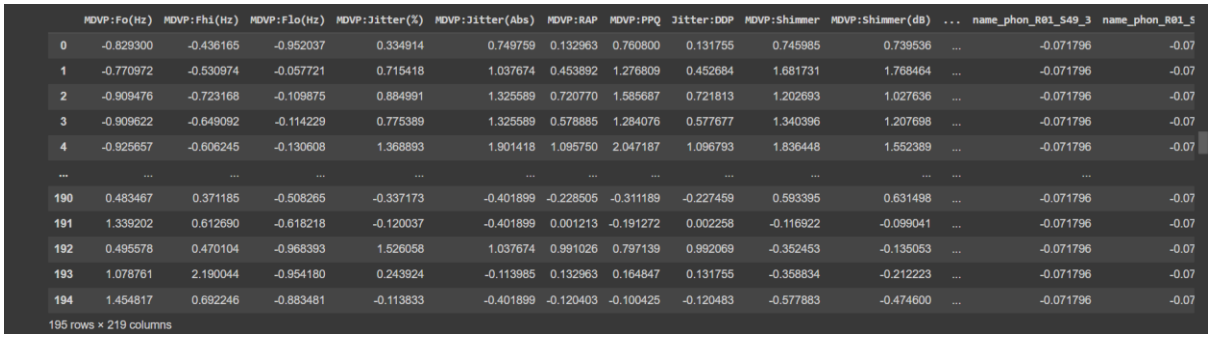
Standardizing The Data

```
from sklearn.preprocessing import StandardScaler
df = pd.DataFrame(one_hot_encoded_data)
scaler = StandardScaler()
df_standardized = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
df_standardized
```

Figure 12. Standardisation code

This python code standardizes the features by subtracting the mean and scaling to unit variance.

It is calculated as $z = \frac{(x-\mu)}{s}$ where μ is mean and S is Standard Deviation.



	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)	MDVP:RAP	MDVP:PPQ	Jitter:DDP	MDVP:Shimmer	MDVP:Shimmer(dB)	...	name_phon_R01_S49_3	name_phon_R01_S
0	-0.829300	-0.436165	-0.952037	0.334914	0.749759	0.132963	0.760800	0.131755	0.745985	0.739536	...	-0.071796	-0.07
1	-0.770972	-0.530974	-0.057721	0.715418	1.037674	0.453892	1.276809	0.452684	1.681731	1.768464	...	-0.071796	-0.07
2	-0.909476	-0.723168	-0.109875	0.884991	1.325589	0.720770	1.585687	0.721813	1.202693	1.027636	...	-0.071796	-0.07
3	-0.909622	-0.649092	-0.114229	0.775389	1.325589	0.578885	1.284076	0.577677	1.340396	1.207698	...	-0.071796	-0.07
4	-0.925657	-0.606245	-0.130608	1.368893	1.901418	1.095750	2.047187	1.096793	1.836448	1.552389	...	-0.071796	-0.07
...	...	...	...	...	...	...	...	...	...	...	...	...	
190	0.483467	0.371185	-0.508265	-0.337173	-0.401899	-0.228505	-0.311189	-0.227459	0.593395	0.631498	...	-0.071796	-0.07
191	1.339202	0.612690	-0.618218	-0.120037	-0.401899	0.001213	-0.191272	0.002258	-0.116922	-0.099041	...	-0.071796	-0.07
192	0.495578	0.470104	-0.968393	1.526058	1.037674	0.991026	0.797139	0.992069	-0.352453	-0.135053	...	-0.071796	-0.07
193	1.078761	2.190044	-0.954180	0.243924	-0.113985	0.132963	0.164847	0.131755	-0.358834	-0.212223	...	-0.071796	-0.07
194	1.454817	0.692246	-0.883481	-0.113833	-0.401899	-0.120403	-0.100425	-0.120483	-0.577883	-0.474600	...	-0.071796	-0.07

195 rows × 219 columns

Figure 13. Standardization output

3.2.2 Noise Removal (SMOTE Algorithm)

https://colab.research.google.com/drive/1ApM9Rw6Za1FolNbPgNuGqGOxMz7j-vkl#scrollTo=rp8XfM4AhK41

Synthetic Minority Oversampling Technique - SMOTE is an oversampling technique where the synthetic samples are generated for the minority class. This algorithm helps to overcome the overfitting problem posed by random oversampling. It focuses on the feature space to

generate new instances with the help of interpolation between the positive instances that lie together.

SMOTE: Synthetic Minority Oversampling Technique

```
from imblearn.over_sampling import SMOTE
from collections import Counter

counter_Y = Counter(Y_train)
print('Before', counter_Y)

smt = SMOTE()
X_train_smt, Y_train_smt = smt.fit_resample(X_train, Y_train)

counter_Y = Counter(Y_train_smt)
print("After", counter_Y)
```

```
Before Counter({1: 116, 0: 40})
After Counter({1: 116, 0: 116})
```

Figure 14. Implementation of SMOTE and its output

Before performing SMOTE, the number of values of each category [1,0] in Y_train is 1: 116 and 0: 40 but after performing SMOTE algorithm on Y_train, the number of values changes to 1: 116, 0: 116.

3.2.3 One Hot Encoding

https://colab.research.google.com/drive/1ApM9Rw6Za1FolNbPgNuGqGOxMz7j-vkl#scrollTo=rp8XfM4AhK41

It is used to represent categorical variables as numerical values in a machine-learning model. Before performing One hot encoding the dataset has a categorical column called Name which has string values and we need to change them to numerical value.

Trying Onehotencoding on the dataset

It is used to represent categorical variables as numerical values in a machine learning model.

```
[13] df_one = pd.read_csv('/content/with noise_parkinsons - Copy.csv')
```

```
[14] df_one.head()
```

	name	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)	MDVP:RAP	MDVP:PPQ	Jitter:DDP	MDVP:Shimmer	...	NHR	HNR	status	RPDE	DF
0	phon_R01_S01_1	119.992	157.302	74.997	0.00784	0.00007	0.00370	0.00554	0.01109	0.04374	...	0.02211	21.033	1	0.414783	0.81528
1	phon_R01_S01_2	122.400	148.650	113.819	0.00968	0.00008	0.00465	0.00696	0.01394	0.06134	...	0.01929	19.085	1	0.458359	0.81952
2	phon_R01_S01_3	116.682	131.111	111.555	0.01050	0.00009	0.00544	0.00781	0.01633	0.05233	...	0.01309	20.651	1	0.429895	0.82528
3	phon_R01_S01_4	116.676	137.871	111.366	0.00997	0.00009	0.00502	0.00698	0.01505	0.05492	...	0.01353	20.644	1	0.434969	0.81923
4	phon_R01_S01_5	116.014	141.781	110.655	0.01284	0.00011	0.00655	0.00908	0.01966	0.06425	...	0.01767	19.649	1	0.417356	0.82348

5 rows × 25 columns

Figure 15. Dataset before OneHotEncoding

After Performing Onehotencoding the string values of the column [Name] are transformed to categorical value [1,0].

```
one_hot_encoded_data = pd.get_dummies(df_one, columns = ['name'])
one_hot_encoded_data.head(194)
```

MDVP:Shimmer	MDVP:Shimmer(dB)	...	name_phon_R01_S49_3	name_phon_R01_S49_4	name_phon_R01_S49_5	name_phon_R01_S49_6	name_phon_R01_S50_1	name_phon_R01_S50_2	name_phon_R01_S50_3	n
0.04374	0.426	...	0	0	0	0	0	0	0	
0.06134	0.626	...	0	0	0	0	0	0	0	
0.05233	0.482	...	0	0	0	0	0	0	0	
0.05492	0.517	...	0	0	0	0	0	0	0	
0.06425	0.584	...	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	
0.02574	0.255	...	0	0	0	0	1	0	0	
0.04087	0.405	...	0	0	0	0	0	1	0	
0.02751	0.263	...	0	0	0	0	0	0	1	
0.02308	0.256	...	0	0	0	0	0	0	0	
0.02296	0.241	...	0	0	0	0	0	0	0	

Figure 16. Dataset after OneHotEncoding

SECTION 4

4.1 Regression

Regression in machine learning is a supervised learning technique designed for predicting continuous numerical values based on input features. The primary goal is to establish a mathematical relationship between the independent variables and the dependent variable, allowing the model to make predictions for new data points. Linear regression is a fundamental approach, assuming a linear relationship between variables, while more complex models like polynomial regression and support vector regression can capture nonlinear patterns. During training, the model learns the optimal parameters that minimize the difference between predicted and actual values. Evaluation metrics, such as Mean Squared Error or R-squared, assess the model's accuracy. Regression finds extensive applications in various domains, including finance, economics, and healthcare, where predicting numerical outcomes is essential for decision-making and analysis**.**

4.1.1 Linear Regression

Linear regression is a fundamental supervised machine learning technique used for predicting a continuous numerical outcome based on one or more input features. The model assumes a linear relationship between the independent variables and the dependent variable. In simple linear regression, there is one independent variable, while in multiple linear regression, there are multiple independent variables. The model aims to find the best-fit line that minimizes the sum of squared differences between the predicted and actual values. The equation of a simple linear regression model is typically expressed as $y = mx + b$, where y is the dependent variable, x is the independent variable, m is the slope of the line, and b is the y-intercept. The coefficients m and b are determined during the training phase using methods like the least squares approach. Linear regression is widely applied in various fields, including economics, finance, and science, to model and understand the relationships between variables. Evaluation metrics such as Mean Squared Error or R-squared are commonly used to assess the model's performance.

Linear Regression without Hyperparameters

Linear Regression without Hyperparameter

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

data = pd.read_csv('/content/parkinsons.csv')

X = data[['MDVP:Fo(Hz)']]
y = data['status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = LinearRegression()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')

plt.scatter(X_test, y_test, color='black')
plt.plot(X_test, y_pred, color='blue', linewidth=3)
plt.title('Linear Regression')
plt.xlabel('X')
plt.ylabel('y')
plt.show()
```

Figure 17. Linear Regression without Hyperparameters

Output:

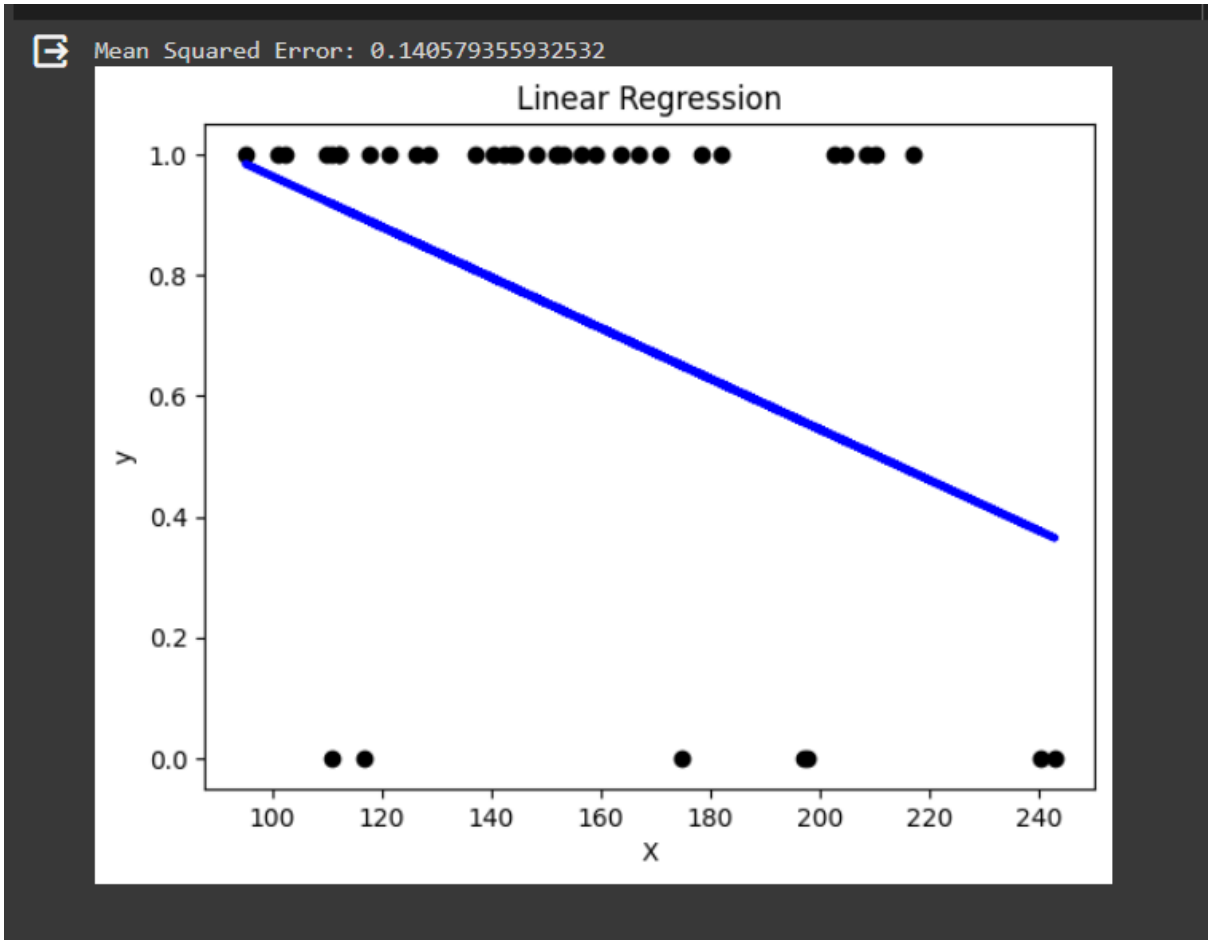


Figure 18. Output of Linear Regression without hyperparameter

Inference:

In this code, I'm utilizing the scikit-learn library in Python to perform a simple linear regression analysis. I first import the necessary modules, including LinearRegression for building the regression model and mean_squared_error for evaluating its performance. The dataset, loaded from a CSV file containing Parkinson's disease-related features, is then prepared by selecting the 'MDVP:Fo(Hz)' attribute as the independent variable (X) and the 'status' attribute as the dependent variable (y). I split the data into training and testing sets using the train_test_split function from scikit-learn. Subsequently, I instantiate a linear regression model, fit it to the training data, and make predictions on the test set. The mean squared error (MSE) between the true and predicted values is calculated as a performance metric. Finally, I visualize the regression line along with the test data points using matplotlib. This plot provides a visual representation of how well the linear regression model fits the data.

Table 6. Parameters for Linear regression without Hyperparameter

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42

Linear Regression with Hyperparameter

Linear regression doesn't involve hyperparameter tuning in the same way as some other machine learning algorithms, such as support vector machines or neural networks. Linear regression is a simple algorithm, and it generally doesn't have hyperparameters that need fine-tuning. However, we can enhance the linear regression model by considering variations like regularized linear regression, such as Ridge or Lasso regression, which do involve hyperparameters.

Linear Regression With Hyperparameter

```
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

data = pd.read_csv('/content/parkinsons.csv')

X = data[['MDVP:Fo(Hz)']]
y = data['status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

ridge_model = Ridge()

param_grid = {'alpha': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

grid_search = GridSearchCV(ridge_model, param_grid, cv=5)
grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
print(f'Best Hyperparameters: {best_params}')

best_model = grid_search.best_estimator_

y_pred_ridge = best_model.predict(X_test)

mse_ridge = mean_squared_error(y_test, y_pred_ridge)
print(f'Mean Squared Error (Ridge): {mse_ridge}')

plt.scatter(X_test, y_test, color='black')
plt.plot(X_test, y_pred_ridge, color='blue', linewidth=3)
plt.title('Ridge Regression')
plt.xlabel('X')
plt.ylabel('y')
plt.show()
```

Figure 19. Source Code of Linear Regression with Hyperparameter

Output:

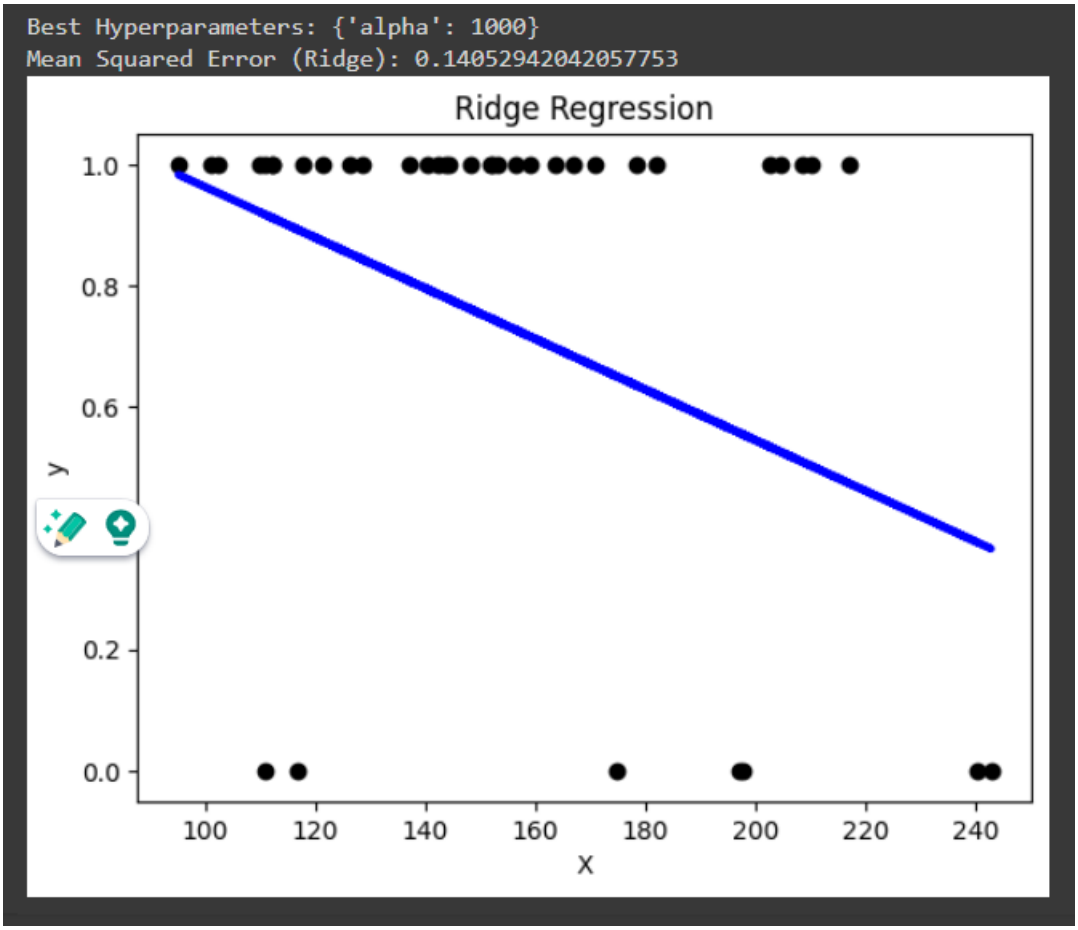


Figure 20. Output for Linear Regression with Hyperparameter

Inference:

In this code, I'm employing Ridge regression on a Parkinson's disease dataset using scikit-learn. The dataset is loaded, and the 'MDVP:Fo(Hz)' feature is selected as the independent variable, while the 'status' attribute is set as the target variable. The data is then split into training and testing sets with an 80-20 split ratio, maintaining reproducibility with a random state of 42. A Ridge regression model is instantiated, and a grid search is performed to find the optimal hyperparameter, 'alpha', which controls the regularization strength. The grid search explores various alpha values using 5-fold cross-validation. The best hyperparameter and corresponding model are identified, and predictions are made on the test set. The mean squared error is calculated to quantify the model's performance. The scatter plot visually presents the test data points alongside the Ridge regression curve, offering insights into how well the model fits the data. The output includes the best hyperparameters and the mean squared error, providing a quantitative assessment of the Ridge regression model's effectiveness in predicting the 'status' variable.

Table 7. Parameters for Linear Regression with Hyperparameter

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Grid Search	The grid search explores various alpha values using 5-fold cross-validation.	0.001, 0.01, 0.1, 1, 10, 100, 1000

4.1.2 Logistic Regression

In the realm of machine learning, logistic regression is a versatile and widely-used algorithm that I find particularly fascinating. Despite its name, it's employed for binary classification tasks rather than regression. The core idea is to model the probability that a given input belongs to a particular class. I appreciate its simplicity and interpretability, as it computes a weighted sum of input features, applies a sigmoid function to transform this sum into a probability score between 0 and 1, and then classifies instances based on a chosen threshold. During training, the model learns optimal weights through techniques like gradient descent, minimizing the binary cross-entropy loss. I often leverage logistic regression when dealing with binary outcomes, such as predicting whether an email is spam or not. Its straightforward implementation, clear decision boundaries, and ability to handle high-dimensional data make it an invaluable tool in my machine-learning toolbox.

Logistic regression without Hyperparameter

Logistic Regression without hyperparameter

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, mean_squared_error
import pandas as pd

data = pd.read_csv('/content/parkinsons.csv')

X = data.drop(columns=['name', 'status'], axis=1)
y = data['status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

model = LogisticRegression()

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

accuracy = accuracy_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f'Accuracy: {accuracy}')
print(f'Mean Squared Error: {mse}')
```

Figure 21. Logistic Regression without Hyperparameters

Output:

```
Accuracy: 0.8974358974358975
Mean Squared Error: 0.10256410256410256
```

Figure 22. Output of Logistic Regression without Hyperparameter

Inference:

In this code, I'm working with a Parkinson's disease dataset, and my goal is to predict the disease status using logistic regression. After loading the dataset, I split it into training and testing sets, reserving 20% of the data for testing to evaluate the model's performance. To ensure a fair comparison, I scale the features using StandardScaler to center and normalize them. Next, I instantiate a logistic regression model and train it on the scaled training data. I then use the trained model to predict the disease status on the scaled test set. The code calculates and prints two important metrics for model evaluation: accuracy and mean squared error (MSE). Accuracy represents the proportion of correctly classified instances, while MSE quantifies the average squared difference between the true and predicted values. Both metrics contribute to a comprehensive assessment of the logistic regression model's effectiveness in predicting Parkinson's disease status.

Table 8. Parameters for Logistic Regression

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42

Logistic Regression with Hyperparameter

Logistic Regression with hyperparameter

```
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, mean_squared_error

data = pd.read_csv('/content/parkinsons.csv')

X = data.drop(columns=['name', 'status'], axis=1)
y = data['status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

model = LogisticRegression(penalty='l1', solver='liblinear')

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

grid_search = GridSearchCV(model, param_grid, cv=5)
```

Figure 23. Source code for Logistic regression with Hyperparameter part 1

```
+ Code  + Text
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

model = LogisticRegression(penalty='l1', solver='liblinear')

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

grid_search = GridSearchCV(model, param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)

best_params = grid_search.best_params_
print(f'Best Hyperparameters: {best_params}')

best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test_scaled)

accuracy = accuracy_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)

print(f'Accuracy: {accuracy}')
print(f'Mean Squared Error: {mse}')
```

Figure 24. Source code for Logistic regression with Hyperparameter part 2

Output:

```
Best Hyperparameters: {'C': 10}
Accuracy: 0.9230769230769231
Mean Squared Error: 0.07692307692307693
```

Figure 25. Output for Logistic regression with Hyperparameter

Inference:

In this code, I'm working with a Parkinson's disease dataset to build a logistic regression model for disease status prediction. The dataset is loaded, and the features are scaled using StandardScaler to ensure uniformity and prevent any feature from dominating due to scale differences. The data is split into training and testing sets, with 20% reserved for testing. I employ logistic regression with L1 regularization (penalty='l1') and the 'liblinear' solver. A grid search is performed to find the optimal regularization parameter 'C' through cross-validated training. The best hyperparameters are printed, and the model with the optimal configuration is selected. Subsequently, the model is used to predict the disease status on the scaled test set. Two key performance metrics, accuracy, and mean squared error (MSE) are calculated and printed to assess the model's predictive accuracy and precision. The accuracy score represents the proportion of correctly predicted instances, while MSE provides a measure of the average squared difference between the true and predicted values. These metrics collectively offer a comprehensive evaluation of the logistic regression model's efficacy in predicting Parkinson's disease status.

Table 9. Parameters for Logistic Regression with Hyperparameter

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Grid Search	The grid search explores various alpha values using 5-fold cross-validation.	0.001, 0.01, 0.1, 1, 10, 100, 1000

4.1.3 Lasso Regression

Lasso regression, or Least Absolute Shrinkage and Selection Operator, is a regularization technique that I find particularly powerful in the realm of machine learning. What sets Lasso apart is its ability to not only fit a predictive model but also perform feature selection by driving certain coefficients to exactly zero. This regularization method adds a penalty term to the linear regression cost function, which is proportional to the absolute values of the coefficients. By tuning the regularization strength parameter, often denoted as 'alpha,' Lasso encourages sparsity in the model, effectively eliminating less influential features. This feature selection property proves beneficial when dealing with high-dimensional datasets, where identifying and using only the most impactful features can enhance model interpretability and generalization to new data. I appreciate Lasso regression for its role in not just prediction but also in simplifying models by emphasizing the most relevant features, making it a valuable tool in my machine learning endeavours.

Lasso Regression without Hyperparameter

Lasso Regression without hyperparameter

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

data = pd.read_csv('/content/parkinsons.csv')

X = df_main.drop(columns = ['name', 'status'], axis = 1)
y = df_main['status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)


alpha = 0.01
lasso_model = Lasso(alpha=alpha)
lasso_model.fit(X_train_scaled, y_train)

y_pred = lasso_model.predict(X_test_scaled)
```

Figure 26. Lasso Regression without Hyperparameter part 1

```
alpha = 0.01
lasso_model = Lasso(alpha=alpha)
lasso_model.fit(X_train_scaled, y_train)

y_pred = lasso_model.predict(X_test_scaled)

mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')


coefficients = pd.DataFrame({'Feature': X.columns, 'Coefficient': lasso_model.coef_})
print(coefficients)

plt.scatter(y_test, y_pred)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted Values')
plt.show()
```

Figure 27. Lasso Regression without Hyperparameter part 2

Output:

```
Mean Squared Error: 0.1118176815117204
             Feature  Coefficient
0        MDVP:Fo(Hz)    -0.058409
1       MDVP:Fhi(Hz)    -0.042709
2       MDVP:Flo(Hz)    -0.057327
3     MDVP:Jitter(%)    -0.023903
4   MDVP:Jitter(Abs)    -0.000000
5           MDVP:RAP    -0.000000
6           MDVP:PPQ    -0.000000
7         Jitter:DDP    -0.000000
8       MDVP:Shimmer     0.000000
9   MDVP:Shimmer(dB)     0.000000
10      Shimmer:APQ3     0.000000
11      Shimmer:APQ5     0.000000
12          MDVP:APQ     0.000000
13       Shimmer:DDA     0.000000
14               NHR    -0.000000
15               HNR    -0.000000
16              RPDE    -0.015488
17               DFA     0.033400
18           spread1     0.142613
19           spread2     0.025688
20                D2     0.094859
21               PPE     0.000000
```

Figure 28. Output for Lasso regression without Hyperparameter part 1

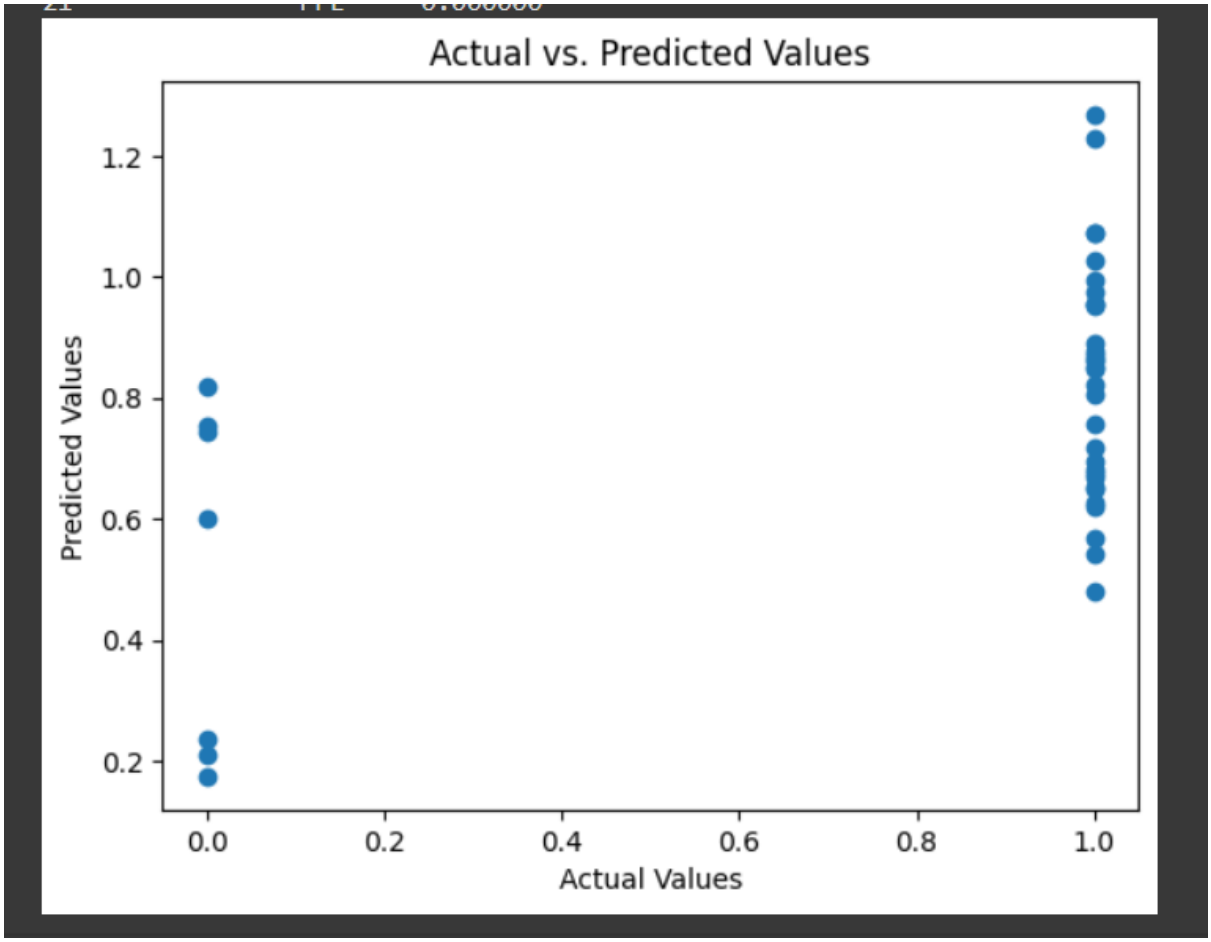


Figure 29. Output for Lasso regression without Hyperparameter part 2

Inference:

In this code, I'm exploring the application of Lasso regression on a Parkinson's disease dataset. After loading the data, I preprocess it by splitting it into training and testing sets and scaling the features using StandardScaler. The Lasso regression model is then instantiated with a regularization parameter, alpha, set to 0.01. This hyperparameter controls the strength of the regularization, influencing the sparsity of the model. The model is trained on the scaled training data, and predictions are made on the test set. The mean squared error (MSE) is calculated to evaluate the predictive performance of the model. Additionally, I inspect the coefficients of the features after Lasso regularization, showcasing the extent to which certain features contribute to the model. Finally, a scatter plot visualizes the relationship between the actual and predicted values, providing a qualitative assessment of the model's accuracy. This code not only quantifies the model's performance but also aids in understanding the impact of Lasso regularization on feature selection and prediction quality.

Table 10. Parameters for Lasso Regression

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Alpha	The regularization parameter controls the amount of shrinkage applied to the coefficients of the model	0.01

Lasso Regression with Hyperparameter

Lasso Regression with hyperparaters

```
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

data = pd.read_csv('/content/parkinsons.csv')

X = data.drop(columns=['name', 'status'], axis=1)
y = data['status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

lasso_model = Lasso()

param_grid = {'alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Figure 30. Lasso Regression with Hyperparameter part 1

```python
lasso_model = Lasso()

param_grid = {'alpha': [0.001, 0.01, 0.1, 1, 10, 100]}

grid_search = GridSearchCV(lasso_model, param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)

best_params = grid_search.best_params_

print(f'Best Hyperparameters: {best_params}')

best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test_scaled)

mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')

coefficients = pd.DataFrame({'Feature': X.columns, 'Coefficient': best_model.coef_})
print(coefficients)

plt.scatter(y_test, y_pred)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted Values')
plt.show()
```

Figure 31. Lasso Regression with Hyperparameter part 2

Output:

```
Best Hyperparameters: {'alpha': 0.001}
Mean Squared Error: 0.10607698891776611
             Feature  Coefficient
0        MDVP:Fo(Hz)    -0.113766
1       MDVP:Fhi(Hz)    -0.022511
2       MDVP:Flo(Hz)    -0.064668
3     MDVP:Jitter(%)    -1.047414
4   MDVP:Jitter(Abs)    -0.091697
5           MDVP:RAP     1.039121
6           MDVP:PPQ    -0.005234
7         Jitter:DDP     0.005396
8       MDVP:Shimmer     0.089002
9   MDVP:Shimmer(dB)     0.113434
10      Shimmer:APQ3     0.000000
11      Shimmer:APQ5    -0.186732
12          MDVP:APQ     0.024245
13       Shimmer:DDA     0.000000
14               NHR    -0.082764
15               HNR    -0.053368
16              RPDE    -0.091985
17               DFA     0.022009
18           spread1     0.139168
19           spread2     0.096197
20                D2     0.044894
21               PPE     0.102131
```

Figure 32. Output for Lasso Regression with Hyperparameter part 1

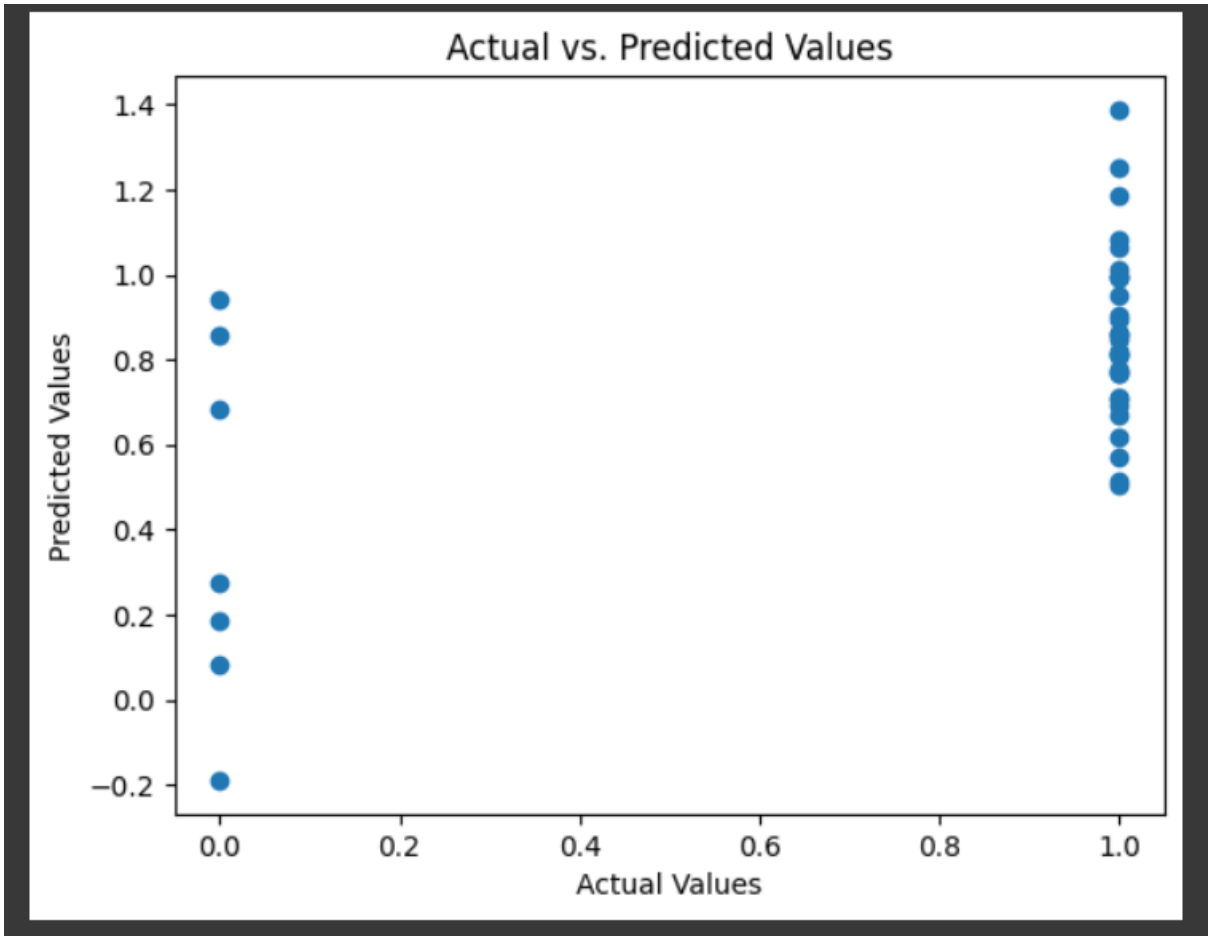


Figure 33. Output for Lasso Regression with Hyperparameter part 2

Inference:

In this code, I'm utilizing Lasso regression on a Parkinson's disease dataset to predict the disease status. The data is loaded, and after splitting it into training and testing sets, I employ StandardScaler to scale the features and ensure consistent magnitudes. The Lasso regression model is then instantiated, and a grid search is conducted to find the optimal alpha, the regularization parameter. The grid search explores various alpha values using 5-fold cross-validation, and the best hyperparameters are identified. The model with the optimal configuration is then selected, and predictions are made on the scaled test set. Mean Squared Error (MSE) is calculated to quantify the predictive performance of the model. Additionally, I examine the coefficients of the features, showcasing the impact of Lasso regularization on feature selection. The scatter plot visualizes the relationship between the actual and predicted values, offering an intuitive understanding of the model's accuracy. This code not only evaluates the model's effectiveness but also provides insights into the selected features and their contributions to the prediction.

Table 11. Parameters for Lasso Regression with hyperparameter

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Alpha	The regularization parameter controls the amount of shrinkage applied to the coefficients of the model	0.01
Grid Search	The grid search explores various alpha values using 5-fold cross-validation.	0.001, 0.01, 0.1, 1, 10, 100, 1000

4.1.4 Ridge Regression

Ridge regression, a technique that I often find valuable in machine learning, is a form of linear regression that incorporates regularization to mitigate multicollinearity and potential overfitting. What sets Ridge regression apart is its inclusion of a regularization term in the cost function, proportional to the square of the magnitude of the coefficients. By introducing this penalty term, Ridge regression encourages the model to not only fit the data well but also to keep the coefficients small. This can be particularly useful when dealing with datasets with high-dimensional features or when certain features are highly correlated. The regularization strength is controlled by a hyperparameter, often denoted as 'alpha,' and tuning this parameter allows for a balance between fitting the data and preventing overfitting. Ridge regression, therefore, serves as a robust tool in my machine learning toolkit, offering a way to handle complex datasets while promoting stability in model coefficient

Ridge regression without Hyperparameter

Ridge Regression without Hyperparameters

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

data = pd.read_csv('/content/parkinsons.csv')

X = data.drop(columns=['name', 'status'], axis=1)
y = data['status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

alpha_ridge = 0.01
ridge_model = Ridge(alpha=alpha_ridge)
ridge_model.fit(X_train_scaled, y_train)

y_pred_ridge = ridge_model.predict(X_test_scaled)
```

Figure 34. Source code for Ridge Regression part 1

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

alpha_ridge = 0.01
ridge_model = Ridge(alpha=alpha_ridge)
ridge_model.fit(X_train_scaled, y_train)

y_pred_ridge = ridge_model.predict(X_test_scaled)

mse_ridge = mean_squared_error(y_test, y_pred_ridge)
print(f'Mean Squared Error (Ridge): {mse_ridge}')

coefficients_ridge = pd.DataFrame({'Feature': X.columns, 'Coefficient': ridge_model.coef_})
print(coefficients_ridge)

plt.scatter(y_test, y_pred_ridge)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values (Ridge)')
plt.title('Actual vs. Predicted Values (Ridge)')
plt.show()
```

Figure 35. Source code for Ridge Regression part 2

Output:

```
Mean Squared Error (Ridge): 0.11797164876823849
              Feature  Coefficient
0        MDVP:Fo(Hz)    -0.119191
1       MDVP:Fhi(Hz)    -0.021311
2       MDVP:Flo(Hz)    -0.067647
3     MDVP:Jitter(%)    -1.441772
4   MDVP:Jitter(Abs)    -0.125170
5           MDVP:RAP     0.665050
6           MDVP:PPQ     0.123960
7         Jitter:DDP     0.671594
8       MDVP:Shimmer     0.765245
9   MDVP:Shimmer(dB)     0.058944
10      Shimmer:APQ3    -0.087786
11      Shimmer:APQ5    -0.473525
12          MDVP:APQ    -0.056108
13       Shimmer:DDA    -0.183033
14               NHR    -0.102608
15               HNR    -0.070198
16              RPDE    -0.106402
17               DFA     0.027035
18           spread1     0.095401
19           spread2     0.111474
20                D2     0.035648
21               PPE     0.160426
```

Figure 36. Output for Ridge Regression part 1

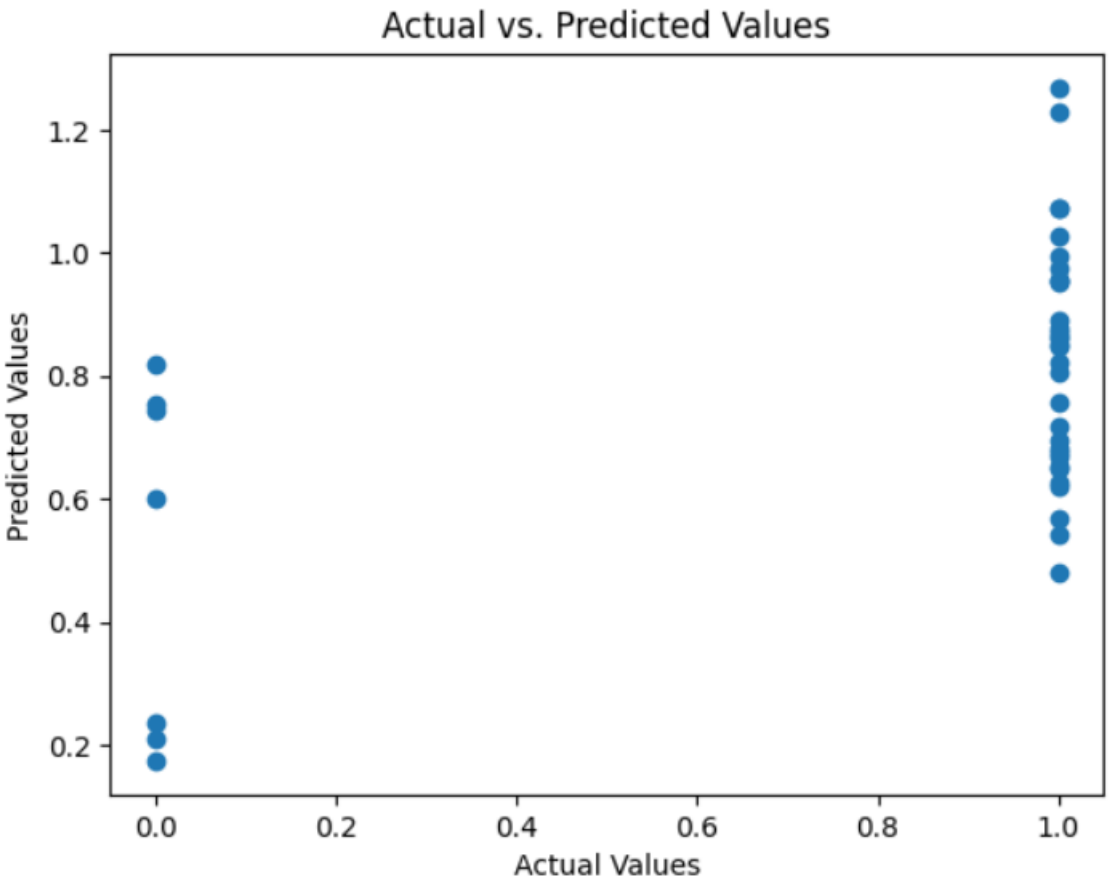


Figure 37. Output for Ridge Regression part 2

Inference:

In this code, I'm utilizing Ridge regression on a Parkinson's disease dataset to predict disease status. After loading the data, I split it into training and testing sets and standardize the features using StandardScaler to ensure consistent scales. Ridge regression is then applied with a regularization parameter, alpha, set to 0.01, controlling the strength of regularization. The model is trained on the scaled training data, and predictions are made on the test set. Mean Squared Error (MSE) is calculated to evaluate predictive performance, providing a measure of the average squared difference between the true and predicted values. Moreover, I examine the coefficients of the features after Ridge regularization, which helps mitigate multicollinearity and potential overfitting. A scatter plot visually illustrates the relationship between the actual and predicted values, offering insights into the accuracy of the Ridge regression model. This code not only quantifies the model's effectiveness but also sheds light on the impact of Ridge regularization on feature coefficients and its influence on prediction quality.

Table 12. Parameters for Ridge Regression

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Alpha	The regularization parameter controls the amount of shrinkage applied to the coefficients of the model	0.01

Ridge Regression with Hyperparameter

Ridge Regression with Hyperparamater

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

data = pd.read_csv('/content/parkinsons.csv')

X = data.drop(columns=['name', 'status'], axis=1)
y = data['status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

ridge_model = Ridge()

param_grid = {'alpha': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}
```

Figure 38. Source Code for Ridge Regression with Hyperparameters part 1

```
grid_search = GridSearchCV(ridge_model, param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)

best_params = grid_search.best_params_

print(f'Best Hyperparameters: {best_params}')

best_model = grid_search.best_estimator_
y_pred_ridge = best_model.predict(X_test_scaled)

mse_ridge = mean_squared_error(y_test, y_pred_ridge)
print(f'Mean Squared Error (Ridge): {mse_ridge}')

coefficients_ridge = pd.DataFrame({'Feature': X.columns, 'Coefficient': best_model.coef_})
print(coefficients_ridge)

plt.scatter(y_test, y_pred_ridge)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values (Ridge)')
plt.title('Actual vs. Predicted Values (Ridge)')
plt.show()
```

Figure 39. Source Code for Ridge Regression with Hyperparameters part 2

Output:

```
Best Hyperparameters: {'alpha': 0.1}
Mean Squared Error (Ridge): 0.1099029040982521
             Feature  Coefficient
0        MDVP:Fo(Hz)    -0.123309
1       MDVP:Fhi(Hz)    -0.021637
2       MDVP:Flo(Hz)    -0.064749
3     MDVP:Jitter(%)    -1.182487
4   MDVP:Jitter(Abs)    -0.140278
5           MDVP:RAP     0.607226
6           MDVP:PPQ    -0.001612
7         Jitter:DDP     0.605365
8       MDVP:Shimmer     0.453921
9   MDVP:Shimmer(dB)     0.114651
10      Shimmer:APQ3    -0.074294
11      Shimmer:APQ5    -0.338799
12          MDVP:APQ    -0.034380
13       Shimmer:DDA    -0.084002
14               NHR    -0.102978
15               HNR    -0.064417
16              RPDE    -0.102891
17               DFA     0.024007
18           spread1     0.101661
19           spread2     0.105134
20                D2     0.036155
21               PPE     0.158613
```

Figure 40. Output for Ridge Regression with Hyperparameter part 1

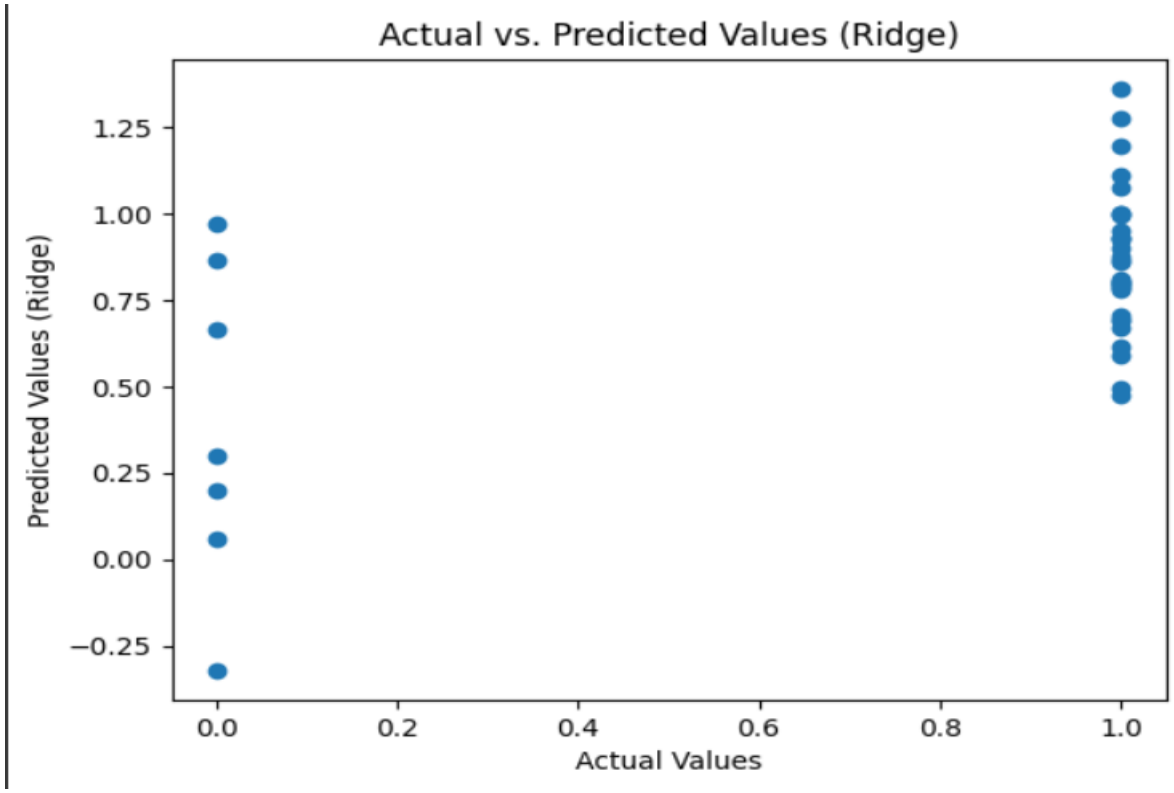


Figure 41. Output for Ridge Regression with Hyperparameter part 2

Inference:

In this code, I'm employing Ridge regression on a Parkinson's disease dataset to predict disease status. After loading the data, I split it into training and testing sets and standardize the features using StandardScaler to ensure consistent scales. Ridge regression is applied with a regularization parameter, alpha, selected through a grid search with values ranging from 0.001 to 1000. The grid search employs 5-fold cross-validation to identify the optimal hyperparameters. The best hyperparameters are then used to train the Ridge model on the scaled training data, and predictions are made on the test set. Mean Squared Error (MSE) is calculated to assess the model's predictive performance. Furthermore, I explore the coefficients of the features after Ridge regularization, which helps address multicollinearity. A scatter plot visually represents the relationship between the actual and predicted values, providing insights into the accuracy of the Ridge regression model. This code not only quantifies the model's effectiveness but also reveals the impact of Ridge regularization on feature coefficients and its influence on prediction quality.

Table 12. Parameters of Ridge Regression with Hyperparameter

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process	42

	becomes deterministic, allowing for result reproducibility.	
Alpha	The regularization parameter controls the amount of shrinkage applied to the coefficients of the model	0.01
Grid Search CV	The grid search explores various alpha values using 5-fold cross-validation.	0.001, 0.01, 0.1, 1, 10, 100, 1000

4.1.5 Comparison of Regression Models

Table 13. Comparison of regression models

	Parameter -1 (Test Size)	**Parameter -2 (Random State)**	**Parameter -3 (Alpha)**	**Parameter -4 (Grid Search CV)**
Linear without Hyperparameter	0.2	42	-	-
Linear with Hyperparameter	0.2	42	-	0.001, 0.01, 0.1, 1, 10, 100, 1000
Logistic without Hyperparameter	0.2	42	-	-
Logistic with Hyperparameter	0.2	42	0.01	0.001, 0.01, 0.1, 1, 10, 100, 1000
Lasso without Hyperparameter	0.2	42	0.01	-
Lasso with Hyperparameter	0.2	42	0.01	0.001, 0.01, 0.1, 1, 10, 100, 1000
Ridge without Hyperparameter	0.2	42	0.01	-
Ridge with Hyperparameter	0.2	42	0.01	0.001, 0.01, 0.1, 1, 10, 100, 1000

SECTION 5

5.1 Classifiers

In the realm of machine learning, classifiers are pivotal components that I frequently leverage in my work. These algorithms are designed to assign labels or categories to input data based on patterns learned from training examples. The primary goal is to generalize this learning to accurately predict the class or category of unseen instances. As I navigate the diverse landscape of classifiers, including but not limited to support vector machines, decision trees, and neural networks, each exhibits unique characteristics and strengths. Their effectiveness hinges on their ability to discern underlying patterns in the data, enabling them to make informed predictions on new, unseen data points. I find classifiers particularly useful in tasks such as image recognition, spam detection, and sentiment analysis, where the ability to categorize input data accurately is crucial for successful model deployment and decision-making.

5.1.1 K-Means with KNN

In my exploration of machine learning techniques, I often encounter and utilize both K-means clustering and K-nearest neighbors (KNN) algorithms. K-means clustering is an unsupervised learning method that partitions data into distinct clusters based on similarity, with the number of clusters, denoted as 'K,' pre-defined. This technique is instrumental in uncovering inherent patterns and groupings within datasets. On the other hand, K-nearest neighbours (KNN) is a supervised learning algorithm used for classification and regression tasks. It determines the label of a data point by considering the majority class or averaging the values of its K nearest neighbours. While K-means identifies clusters in the feature space, KNN leverages the proximity of data points to make predictions. The synergy of these two methods can be powerful, especially when clustering data into meaningful groups with K-means and subsequently employing KNN for classification tasks within each cluster. This integrated approach enhances my ability to discern patterns, make predictions, and derive meaningful insights from diverse datasets.

K-means with KNN with Hyperparamters

```
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from matplotlib.colors import ListedColormap

dataset = pd.read_csv('/content/with noise_parkinsons - Copy.csv')

features = ['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)', 'MDVP:Flo(Hz)']

y = dataset['status']

X_kmeans = dataset[features]
kmeans = KMeans(n_clusters=2, random_state=42)
dataset['cluster'] = kmeans.fit_predict(X_kmeans)

plt.scatter(X_kmeans.iloc[:, 0], X_kmeans.iloc[:, 1], c=dataset['status'], cmap='viridis', edgecolor='k')
```

Figure 42. Source Code of K-means with KNN with Hyperparameter part 1

```
dataset['cluster'] = kmeans.fit_predict(X_kmeans)

plt.scatter(X_kmeans.iloc[:, 0], X_kmeans.iloc[:, 1], c=dataset['status'], cmap='viridis', edgecolor='k')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=300, c='red', marker='X', label='Centroids')
plt.title('K-Means Clustering')
plt.xlabel('MDVP:Fo(Hz)')
plt.ylabel('MDVP:Fhi(Hz)')
plt.legend()
plt.show()

X_knn = dataset[features]
X_train, X_test, y_train, y_test = train_test_split(X_knn, y, test_size=0.2, random_state=42)

# Standardize features for KNN
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

param_grid = {'n_neighbors': [3, 5, 7, 9]}
knn_classifier = KNeighborsClassifier()
grid_search = GridSearchCV(knn_classifier, param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)

print("Best Hyperparameters:", grid_search.best_params_)

y_pred = grid_search.predict(X_test_scaled)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

Figure 43. Source Code of K-means with KNN with Hyperparameter part 2

```
y_pred = grid_search.predict(X_test_scaled)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix (KNN)')
plt.show()

def plot_decision_boundary(X, y, model, h=0.02):
    cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#0000FF'])

    x_min, x_max = X.iloc[:, 0].min() - 1, X.iloc[:, 0].max() + 1
    y_min, y_max = X.iloc[:, 1].min() - 1, X.iloc[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("Decision Boundary (KNN)")
    plt.show()
```

Figure 44. Source Code of K-means with KNN with Hyperparameter part 3

Hyperparameter Tuning approach used: Grid Search CV

Inference:

In this code, I'm exploring the combination of K-means clustering and K-nearest neighbors (KNN) for a Parkinson's disease dataset. I begin by applying K-means clustering to three selected acoustic features, creating two clusters. The scatter plot showcases the distribution of data points coloured by their true status and includes centroids marked in red. Next, I use KNN for classification on the same feature set, optimizing the number of neighbors through grid search. The accuracy of the KNN model is then evaluated on a test set, and a confusion matrix visualizes the model's performance. Additionally, a custom function plots the decision boundary of the KNN classifier on the original data, illustrating how the model classifies regions in the feature space. This combined approach provides insights into both cluster patterns and classification boundaries, offering a comprehensive understanding of the dataset's structure and the predictive capabilities of the KNN model.

Output:

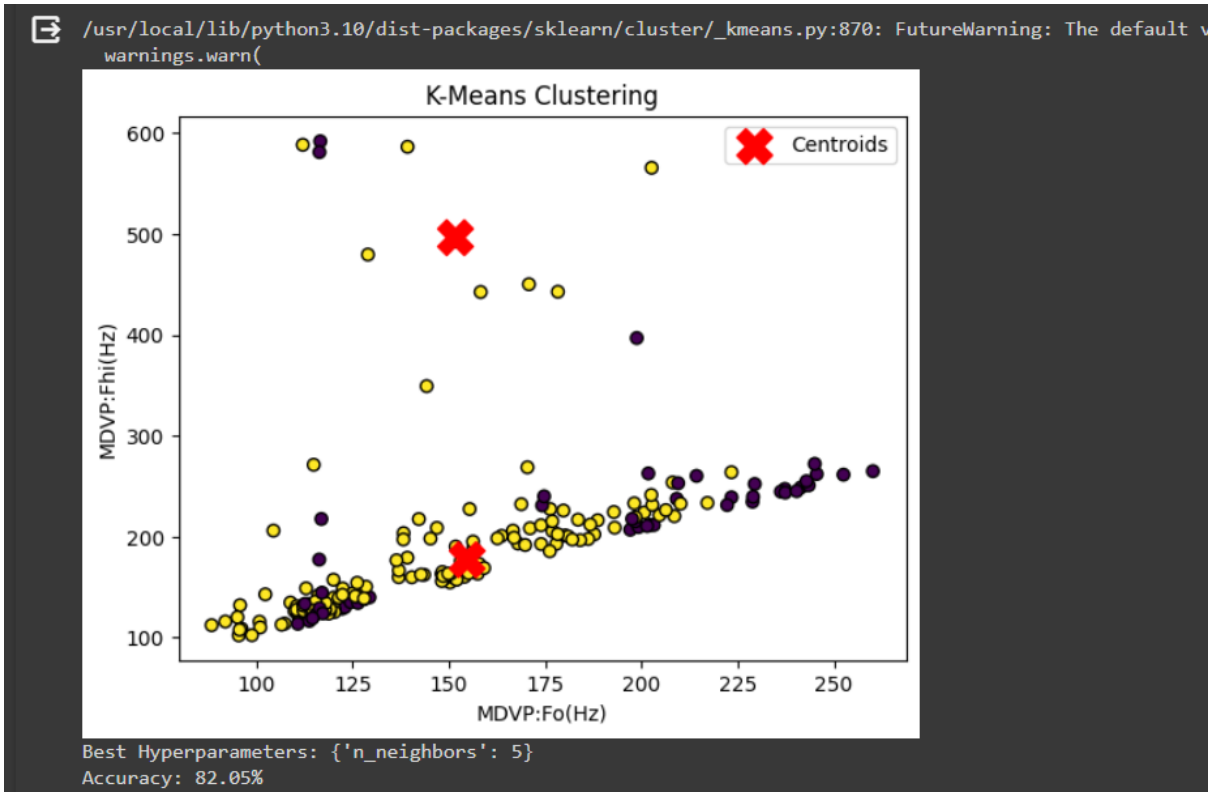


Figure 45. Output for K-means with KNN part 1

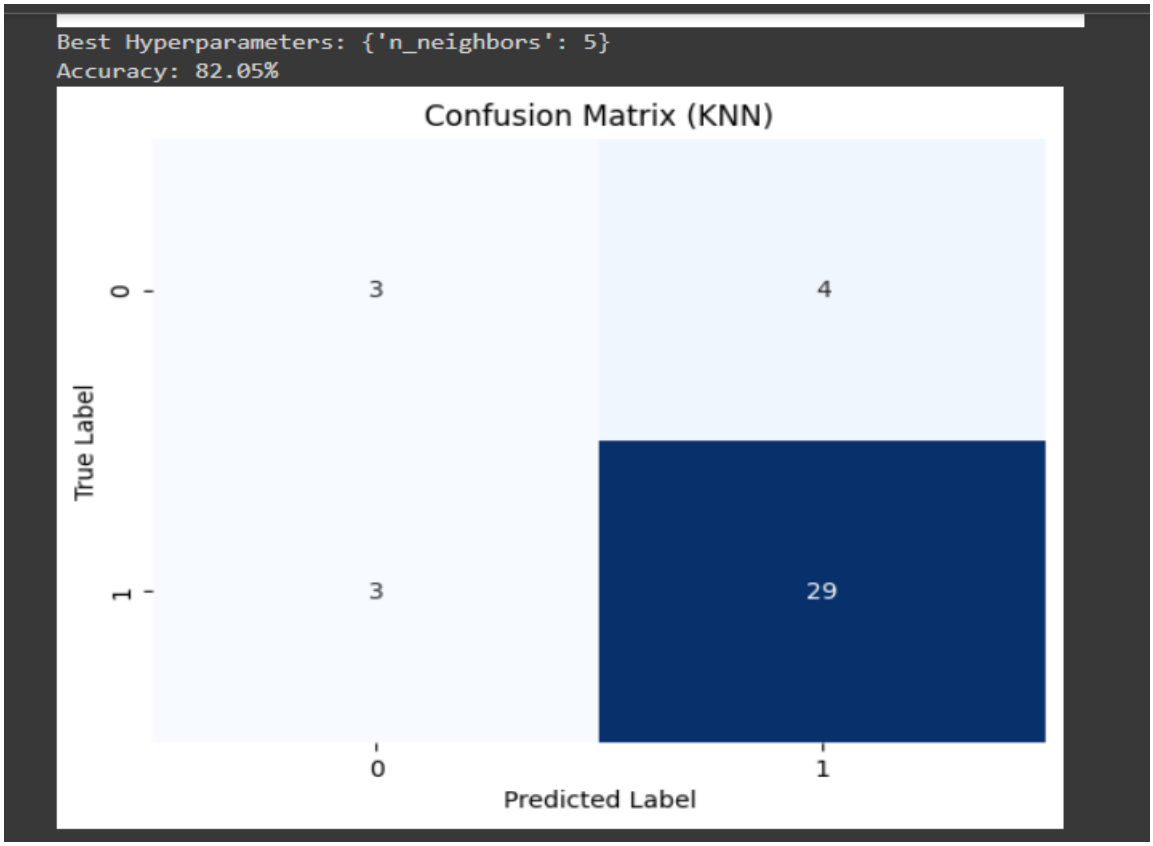


Figure 46. Output for K-means with KNN part 2

Table 14. Parameter table for K-means with KNN

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
No. of Clusters	Number of the centroid which will be formed	2
Grid Search CV	The grid search explores various alpha values using 5-fold cross-validation.	n_neighbors: 3, 5, 7, 9

5.1.2 Fuzzy C-Means with KNN

In this code, I'm working with a Parkinson's disease dataset and exploring a combination of Fuzzy C-means (FCM) clustering and K-nearest neighbors (KNN) classification. Initially, I read the dataset and select three specific acoustic features. To prepare the data for clustering and classification, I standardize the features and then apply Principal Component Analysis (PCA) to reduce dimensionality to two components. The FCM algorithm is employed to create two clusters in the reduced feature space. The resulting clusters are visualized in a scatter plot using the first two principal components. The distinct colors represent the assigned clusters. Subsequently, I split the dataset into training and testing sets, standardize the features, and perform hyperparameter tuning for the KNN classifier using grid search. The optimal number of neighbors is determined, and the KNN model is trained on the training set. The accuracy of the model is evaluated on the test set, and a confusion matrix is generated to assess the classification performance. The heatmap in the confusion matrix provides a visual representation of the true and predicted labels. This combined approach allows for a deeper understanding of the dataset's structure through clustering while also evaluating the predictive power of the KNN classifier. The visualization aids in interpreting how well the data is separated into clusters and how effectively the KNN model classifies instances based on the clustered features.

Fuzzy C-means with KNN with Hyperparameter

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from matplotlib.colors import ListedColormap
from sklearn.decomposition import PCA
from fcmeans import FCM

dataset = pd.read_csv('/content/with noise_parkinsons - Copy.csv')

features = ['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)', 'MDVP:Flo(Hz)']

y = dataset['status']

X_fcm = dataset[features]

scaler = StandardScaler()
X_fcm_scaled = scaler.fit_transform(X_fcm)
pca = PCA(n_components=2)
X_fcm_pca = pca.fit_transform(X_fcm_scaled)

fcm = FCM(n_clusters=2)
```

Figure 47. Source code for Fuzzy C-means with KNN part 1

```
X_fcm_scaled = scaler.fit_transform(X_fcm)
pca = PCA(n_components=2)
X_fcm_pca = pca.fit_transform(X_fcm_scaled)

fcm = FCM(n_clusters=2)
fcm.fit(X_fcm_scaled)

y_pred_fcm = fcm.predict(X_fcm_scaled)

plt.scatter(X_fcm_pca[:, 0], X_fcm_pca[:, 1], c=y_pred_fcm.ravel(), cmap='viridis', edgecolor='k')
plt.title('FCM Clustering')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

X_train, X_test, y_train, y_test = train_test_split(X_fcm_pca, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

param_grid = {'n_neighbors': [3, 5, 7, 9]}
knn_classifier = KNeighborsClassifier()
grid_search = GridSearchCV(knn_classifier, param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)

print("Best Hyperparameters:", grid_search.best_params_)

y_pred = grid_search.predict(X_test_scaled)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

cm = confusion_matrix(y_test, y_pred)
```

Figure 48. Source code for Fuzzy C-means with KNN part 2

```
y_pred_fcm = fcm.predict(X_fcm_scaled)

plt.scatter(X_fcm_pca[:, 0], X_fcm_pca[:, 1], c=y_pred_fcm.ravel(), cmap='viridis', edgecolor='k')
plt.title('FCM Clustering')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

X_train, X_test, y_train, y_test = train_test_split(X_fcm_pca, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

param_grid = {'n_neighbors': [3, 5, 7, 9]}
knn_classifier = KNeighborsClassifier()
grid_search = GridSearchCV(knn_classifier, param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)

print("Best Hyperparameters:", grid_search.best_params_)

y_pred = grid_search.predict(X_test_scaled)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix (KNN)')
plt.show()
```

Figure 49. Source code for Fuzzy C-means with KNN part 3

Hyperparameter Tuning approach used: Grid Search CV

Inference:

In this code, I'm utilizing Fuzzy C-means (FCM) clustering and K-nearest neighbors (KNN) classification on a Parkinson's disease dataset. Initially, I read the dataset and select three specific acoustic features. To prepare the data for clustering and classification, I standardize the features and apply Principal Component Analysis (PCA) to reduce dimensionality to two components for visualization purposes. The FCM algorithm is then applied to create two clusters in the reduced feature space. The resulting clusters are visualized in a scatter plot, where each point is colored according to its assigned cluster. This visualization helps to understand how the FCM algorithm groups instances based on the specified features. Following the clustering step, I split the dataset into training and testing sets and standardize the features. Hyperparameter tuning for the KNN classifier is performed using grid search, optimizing for the number of neighbors. The best hyperparameters are identified, and the KNN model is trained on the training set. The accuracy of the model is then evaluated on the test set, and a confusion matrix is generated to assess the classification performance. The heatmap in the confusion matrix provides a visual representation of the true and predicted labels. This combination of FCM clustering and KNN classification allows for a comprehensive analysis of the dataset, exploring both unsupervised clustering patterns and supervised classification

performance. The visualizations aid in interpreting the clustering results and understanding how well the KNN model can classify instances based on the clustered features.

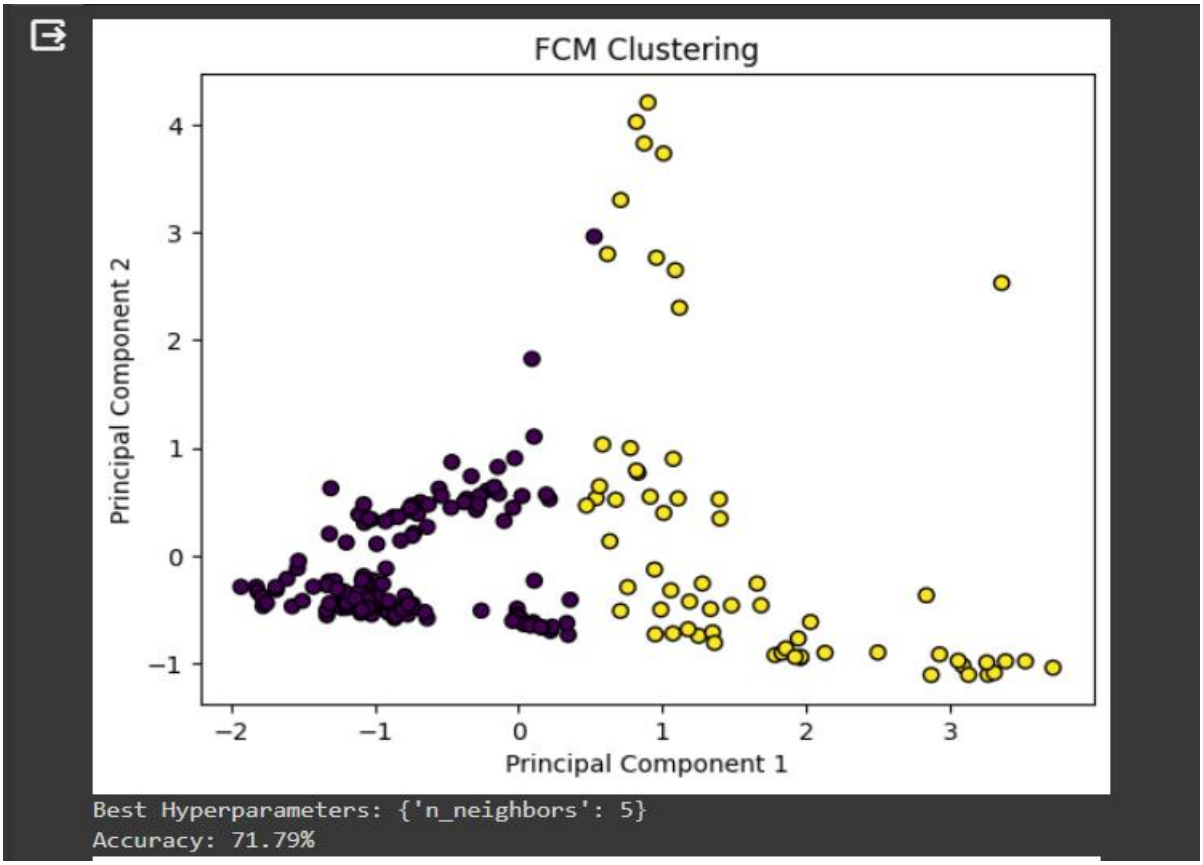


Figure 50. Output for Fuzzy C-means with KNN part 1

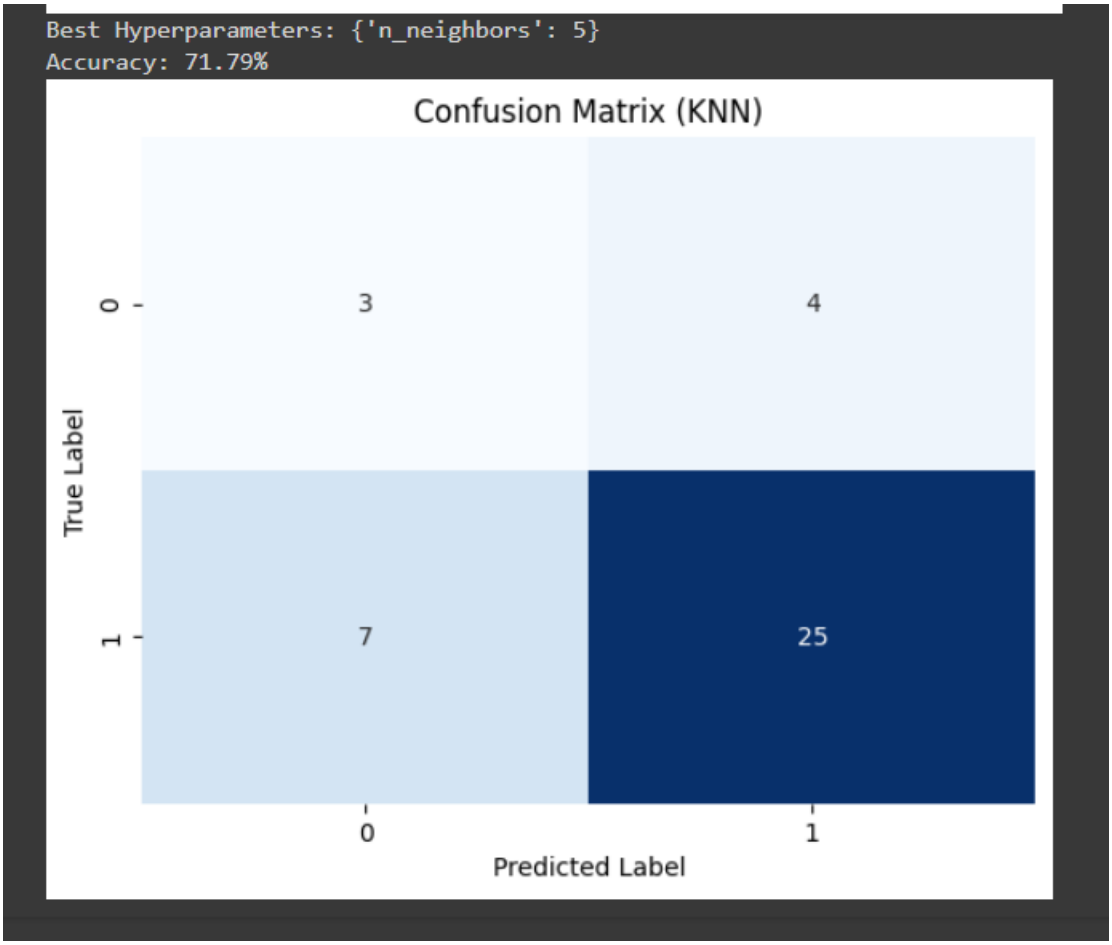


Figure 51. Output for Fuzzy C-means with KNN part 2

Table 15. Parameter table for Fuzzy C-means with KNN

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
No. of Clusters	Number of the centroid which will be formed	2
Grid Search CV	The grid search explores various alpha values using 5-fold cross-validation.	n_neighbors: 3, 5, 7, 9

5.1.3 K-Means with SVM

K-means is a clustering algorithm where I start by dividing a dataset into K clusters based on the similarity of data points. After the clusters are formed, I utilize Support Vector Machines, a supervised learning algorithm, to classify and separate the data points within each cluster. SVM works by finding the hyperplane that best separates different classes in a high-dimensional space. In the context of K-means, SVM helps to refine the clusters by establishing clear boundaries between them, enhancing the overall accuracy and robustness of the clustering results. This combined approach of K-means and SVM allows me to not only identify patterns within data but also create more distinct and well-defined clusters for better classification performance.

K-means with SVM with Hyperparamters

```
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.cluster import KMeans
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap
from sklearn.decomposition import PCA

dataset = pd.read_csv('/content/with noise_parkinsons - Copy.csv')

features = ['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)', 'MDVP:Flo(Hz)']

y = dataset['status']

X_train, X_test, y_train, y_test = train_test_split(dataset[features], y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(X_train_scaled)
```

Figure 52. Source code for K-means with SVM part 1

```
+ Code  + Text
X_test_scaled = scaler.transform(X_test)

kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(X_train_scaled)

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=kmeans.labels_, cmap='viridis', edgecolor='k')
plt.title('K-means Clustering')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

svm_params = {'C': [0.1, 1, 10], 'kernel': ['linear']}
svm_classifier = SVC()
svm_grid_search = GridSearchCV(svm_classifier, svm_params, cv=5)
svm_grid_search.fit(X_train_pca, kmeans.labels_)

print("Best SVM Hyperparameters:", svm_grid_search.best_params_)

y_pred_svm = svm_grid_search.predict(X_test_pca)

accuracy_svm = accuracy_score(y_test, y_pred_svm)
print(f'SVM Accuracy: {accuracy_svm * 100:.2f}%')

cm_svm = confusion_matrix(y_test, y_pred_svm)
sns.heatmap(cm_svm, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Label (SVM)')
plt.ylabel('True Label')
plt.title('Confusion Matrix (SVM)')
plt.show()
```

Figure 53. Source code for K-means with SVM part 2

```
plt.ylabel('True Label')
plt.title('Confusion Matrix (SVM)')
plt.show()

def plot_decision_boundary_svm(X, y, model, h=0.02):
    cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#0000FF'])

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("Decision Boundary (SVM)")
    plt.show()

plot_decision_boundary_svm(X_test_pca, y_test, svm_grid_search, h=0.02)
```

Figure 54. Source code for K-means with SVM part 3

Hyperparameter Tuning approach used: Grid Search CV

Inference:

In this code, I'm working with a Parkinson's disease dataset loaded from a CSV file. I first select three features related to vocal signal characteristics: 'MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)', and 'MDVP:Flo(Hz)'. After splitting the dataset into training and testing sets, I standardize the features using a StandardScaler. Then, I apply K-means clustering with two clusters to the standardized training data. The resulting clusters are visualized in a 2D scatter plot using Principal Component Analysis (PCA) for dimensionality reduction. Next, I employ a Support Vector Machine (SVM) with a linear kernel on the PCA-transformed training data. Grid search is used to find the best hyperparameters for the SVM. The code calculates and prints the accuracy of the SVM on the test set and displays a confusion matrix. Additionally, a function called plot_decision_boundary_svm is defined to visualize the decision boundary of the SVM on the test data. This boundary is shown in a plot alongside the actual data points, aiding in understanding the classification performance. The output includes the K-means clustering plot, the confusion matrix for SVM, and the decision boundary plot, providing insights into the clustering and classification results.

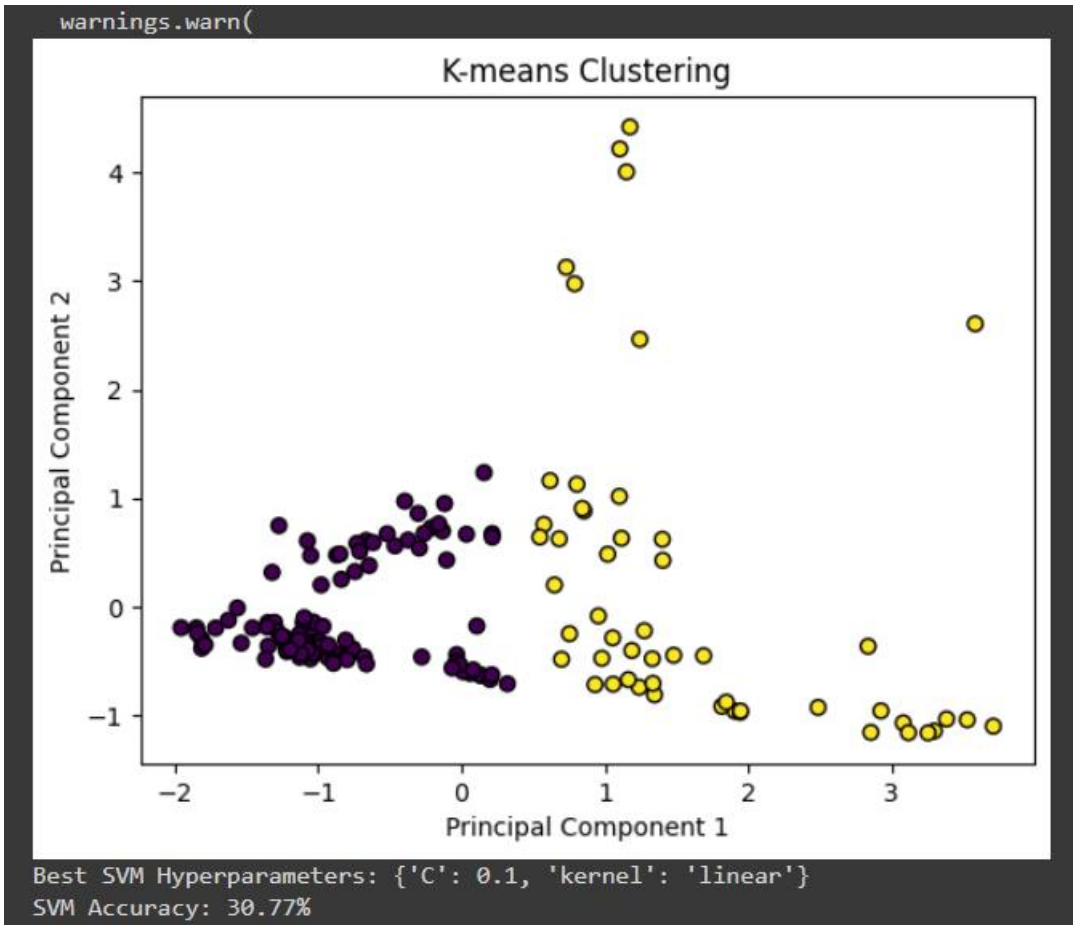


Figure 55. Output for K-means with SVM part 1

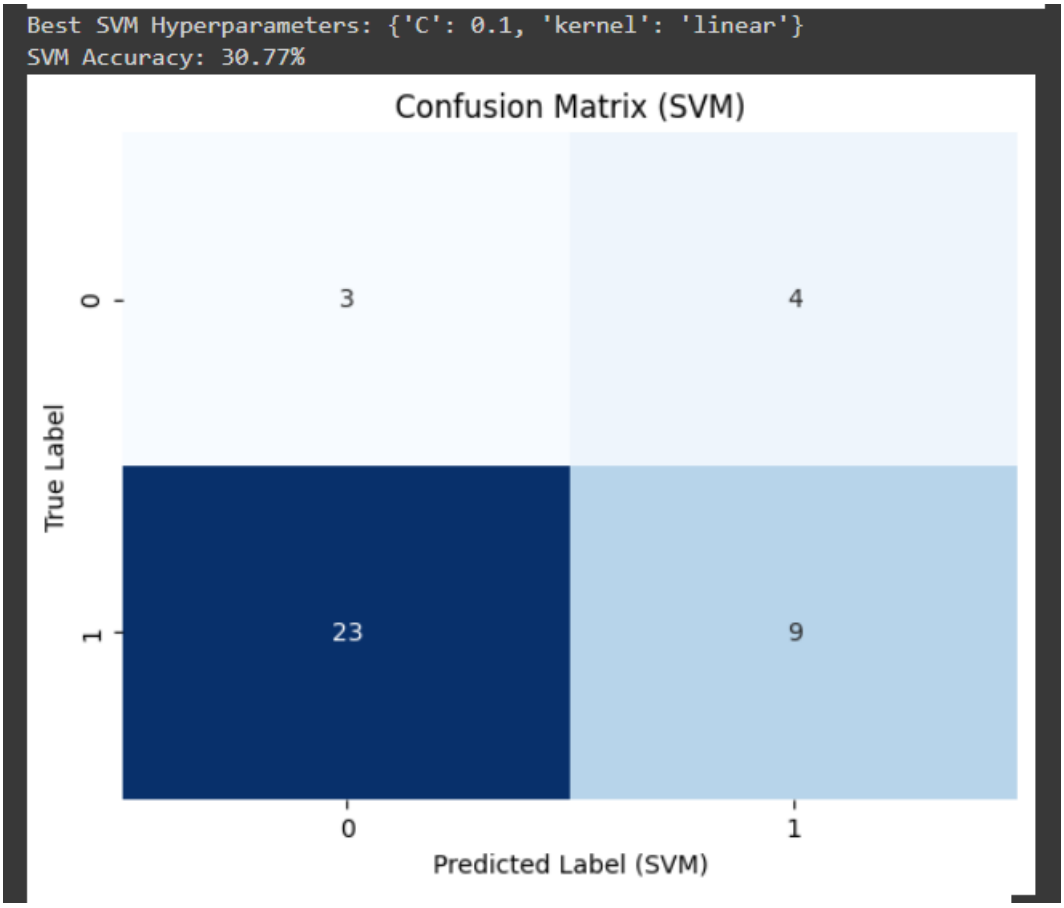


Figure 56. Output for K-means with SVM part 2

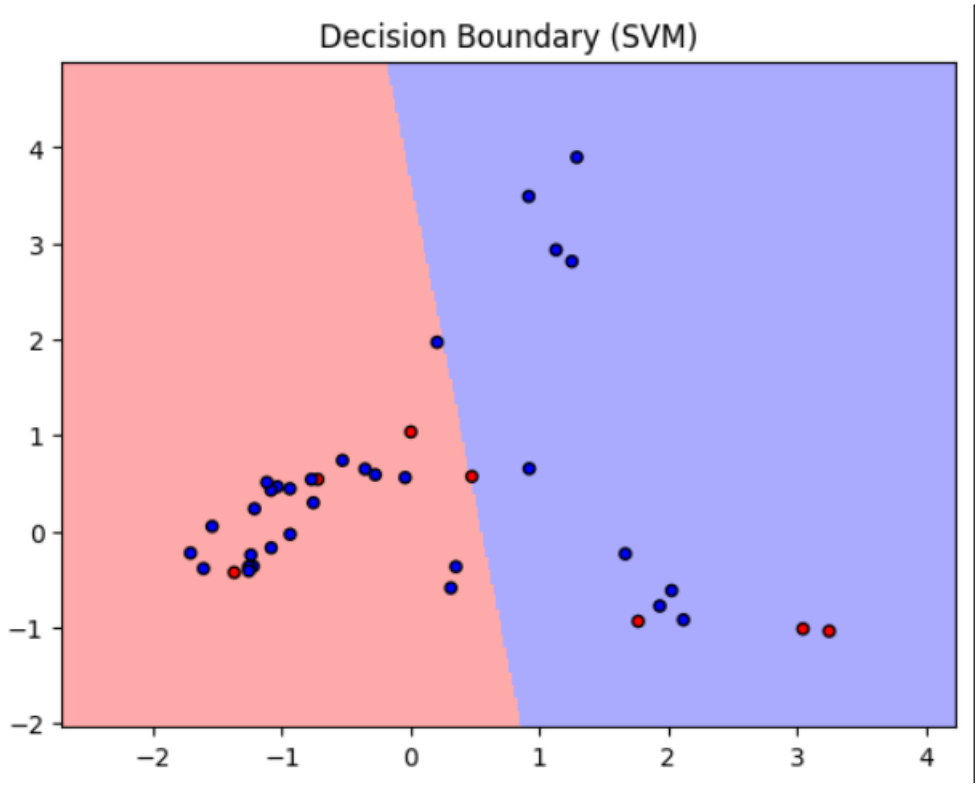


Figure 57. Output for K-means with SVM part 3

Table 16. Parameter table

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Kernel	kernel is a function that computes the dot product of two data points in a transformed feature space.	Linear
Grid Search CV	The grid search explores various alpha values using 5-fold cross-validation.	0.1, 1, 10

5.1.4 Bayesian Classifier

The Bayesian Classifier is a probabilistic model that I use to classify data based on Bayes' theorem. It's a statistical approach that leverages prior knowledge about the data to make predictions or decisions. Essentially, it calculates the probability of a particular outcome given the prior knowledge and the observed evidence. In the context of classification, it's like having a set of rules that help me determine the probability of a data point belonging to a certain category. As I encounter new data, I update my beliefs and refine my predictions, making the Bayesian Classifier a flexible and dynamic tool for handling uncertainty in various domains such as machine learning, spam filtering, and medical diagnosis.

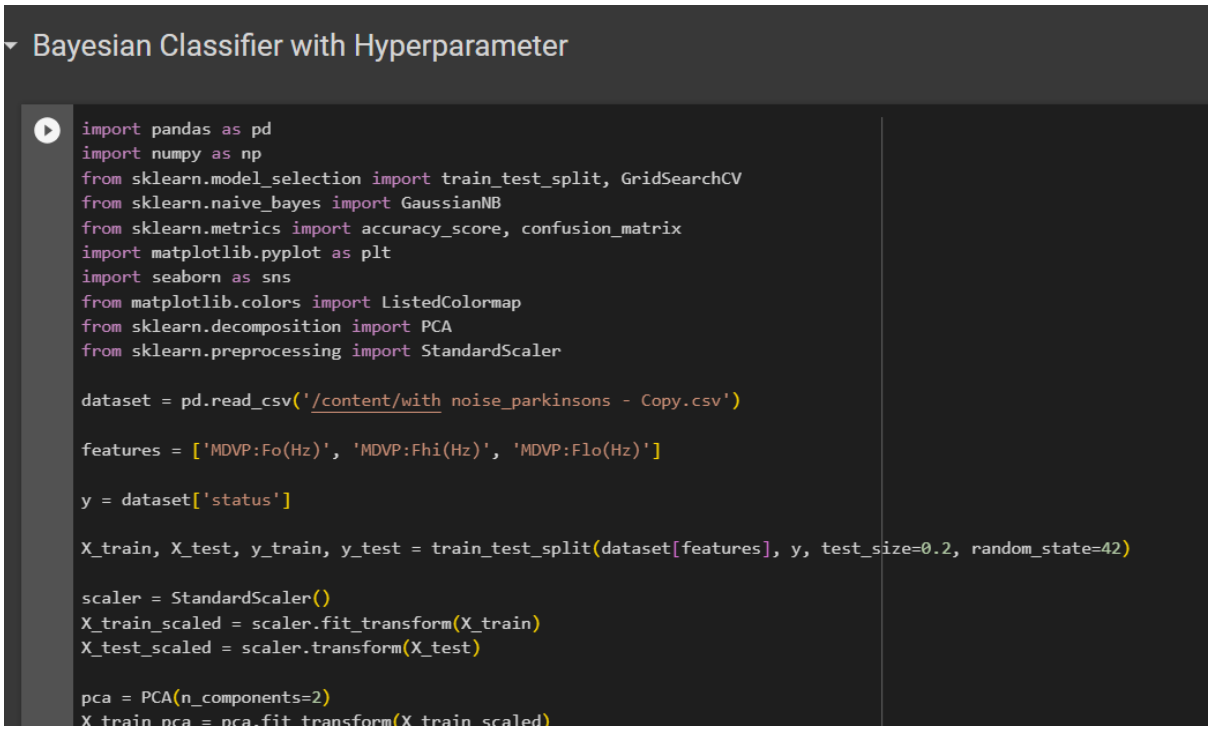


Figure 58. Source code part 1

```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

param_grid = {'var_smoothing': np.logspace(0, -9, num=100)}

nb_classifier = GaussianNB()

grid_search = GridSearchCV(nb_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_pca, y_train)

best_var_smoothing = grid_search.best_params_['var_smoothing']

nb_classifier = GaussianNB(var_smoothing=best_var_smoothing)
nb_classifier.fit(X_train_pca, y_train)

y_pred_nb = nb_classifier.predict(X_test_pca)

accuracy_nb = accuracy_score(y_test, y_pred_nb)
print(f'Gaussian Naive Bayes Accuracy: {accuracy_nb * 100:.2f}%')

cm_nb = confusion_matrix(y_test, y_pred_nb)
sns.heatmap(cm_nb, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Label (Naive Bayes)')
plt.ylabel('True Label')
plt.title('Confusion Matrix (Naive Bayes)')
plt.show()

def plot_decision_boundary_nb(X, y, model, h=0.02):
    cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#0000FF'])
```

Figure 59. Source code part 2

```
cm_nb = confusion_matrix(y_test, y_pred_nb)
sns.heatmap(cm_nb, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Label (Naive Bayes)')
plt.ylabel('True Label')
plt.title('Confusion Matrix (Naive Bayes)')
plt.show()

def plot_decision_boundary_nb(X, y, model, h=0.02):
    cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#0000FF'])

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("Decision Boundary (Naive Bayes)")
    plt.show()

plot_decision_boundary_nb(X_test_pca, y_test, nb_classifier, h=0.02)
```

Figure 60. Source code part 3

Hyperparameter Tuning approach used: Grid Search CV

Inference:

In this code, I'm implementing a Gaussian Naive Bayes classifier for a Parkinson's disease dataset using the scikit-learn library in Python. I start by loading the dataset, which contains features related to the fundamental frequency of a patient's voice. After splitting the data into training and testing sets, I standardize the features and perform Principal Component Analysis (PCA) to reduce the dimensionality to 2 for visualization purposes. I then use GridSearchCV to find the optimal value for the var_smoothing parameter in Gaussian Naive Bayes through cross-validation on the training set. Once I have the best parameter, I train the Naive Bayes classifier on the training data and evaluate its performance on the test set. The code calculates and prints the accuracy of the classifier and displays a confusion matrix to visualize the true positive, true negative, false positive, and false negative predictions. Additionally, the script includes a function, plot_decision_boundary_nb, to plot the decision boundary of the classifier in the reduced feature space. This helps visualize how well the model separates the classes. The final output consists of the accuracy score, the confusion matrix heatmap, and the decision boundary plot, providing a comprehensive assessment of the Naive Bayes classifier's performance on the Parkinson's disease dataset.

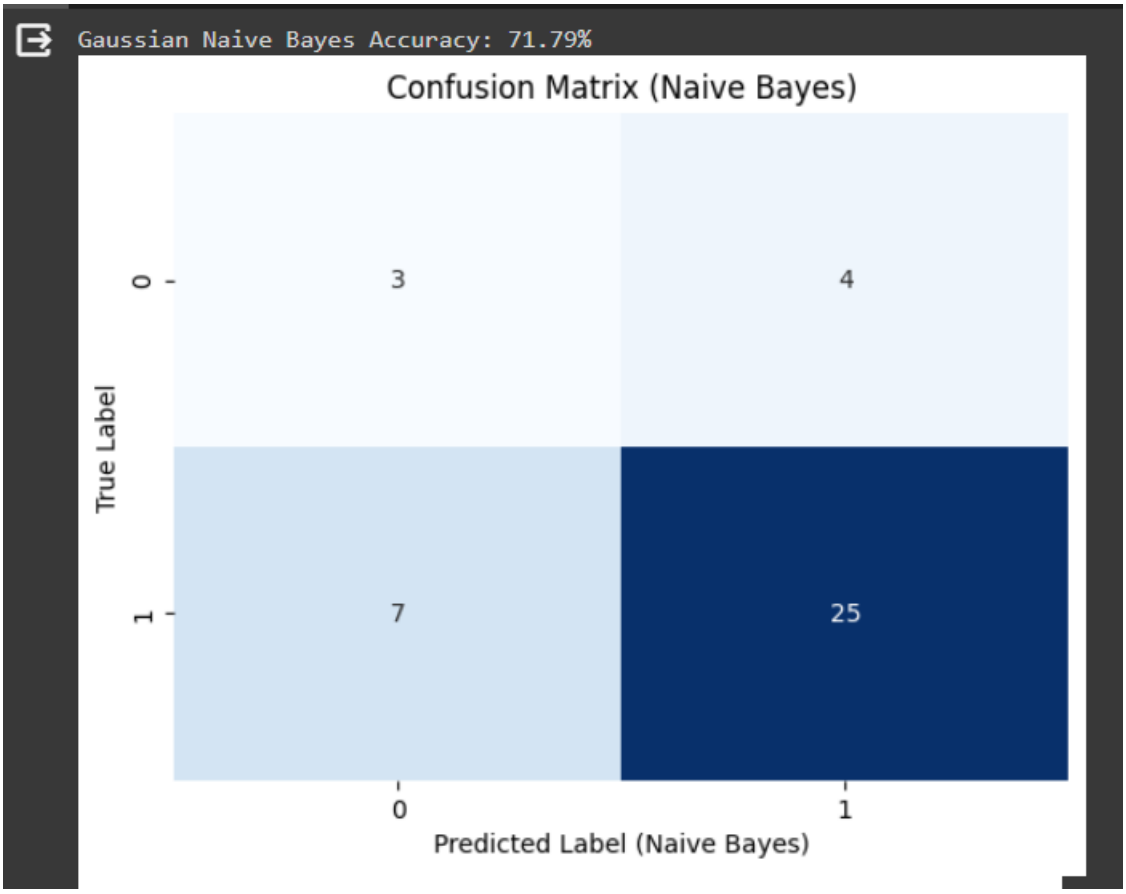


Figure 61. Output part 1

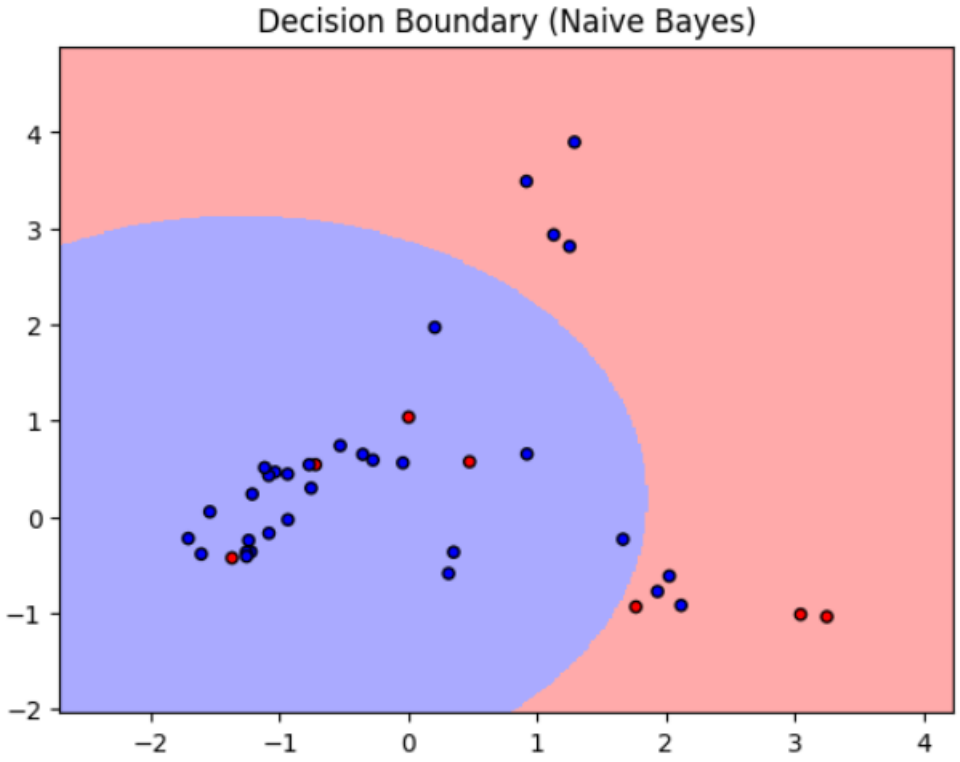


Figure 62. Output part 2

Table 17. Parameter table

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Grid Search CV	The grid search explores various alpha values using 5-fold cross-validation.	np.logspace(0, -9, num=100)

5.1.5 Naïve Bayes Classifier

Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem. It makes the "naive" assumption that the features used for classification are conditionally independent given the class label. Despite its simplicity and this assumption, Naive Bayes often performs well in practice and is computationally efficient. During training, the algorithm estimates the probabilities required by Bayes' theorem based on the given dataset, calculating prior probabilities and conditional probabilities for each feature given each class. In the prediction phase, it applies Bayes' theorem to determine the probability of each class given the observed features, ultimately assigning the class with the highest probability as the predicted class. Naive

Bayes comes in different variants, such as Multinomial, Gaussian, and Bernoulli, catering to different types of data. It is commonly used in text classification, spam filtering, and various other classification tasks.

Naïve Bayes Classifier with Hyperparameters

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

dataset = pd.read_csv('/content/with noise_parkinsons - Copy.csv')

features = ['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)', 'MDVP:Flo(Hz)']

y = dataset['status']

X_train, X_test, y_train, y_test = train_test_split(dataset[features], y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

param_grid = {'var_smoothing': np.logspace(0, -9, num=100)}
```

Figure 63. Source code part 1

```
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

param_grid = {'var_smoothing': np.logspace(0, -9, num=100)}

nb_classifier = GaussianNB()

grid_search = GridSearchCV(nb_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_pca, y_train)

best_var_smoothing = grid_search.best_params_['var_smoothing']

nb_classifier = GaussianNB(var_smoothing=best_var_smoothing)
nb_classifier.fit(X_train_pca, y_train)

y_pred_nb = nb_classifier.predict(X_test_pca)

accuracy_nb = accuracy_score(y_test, y_pred_nb)
print(f'Gaussian Naive Bayes Accuracy: {accuracy_nb * 100:.2f}%')


cm_nb = confusion_matrix(y_test, y_pred_nb)
sns.heatmap(cm_nb, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Label (Naive Bayes)')
plt.ylabel('True Label')
plt.title('Confusion Matrix (Naive Bayes)')
plt.show()

def plot_decision_boundary_nb(X, y, model, h=0.02):
    cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#0000FF'])

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

Figure 64. Source code part 2

```
plt.xlabel('Predicted Label (Naive Bayes)')
plt.ylabel('True Label')
plt.title('Confusion Matrix (Naive Bayes)')
plt.show()

def plot_decision_boundary_nb(X, y, model, h=0.02):
    cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#0000FF'])

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("Decision Boundary (Naive Bayes)")
    plt.show()

plot_decision_boundary_nb(X_test_pca, y_test, nb_classifier, h=0.02)
```

Figure 65. Source code part 3

Hyperparameter Tuning approach used: Grid Search CV

Inference:

In this code, I implemented a Gaussian Naive Bayes classifier for a Parkinson's disease dataset using Python and the scikit-learn library. The dataset is loaded from a CSV file, and three specific features related to vocal characteristics are selected. After splitting the data into training and testing sets, I standardize the features and apply Principal Component Analysis (PCA) for dimensionality reduction. The code then performs a grid search to find the optimal value for the smoothing parameter of the Gaussian Naive Bayes classifier. The best parameter is used to train the classifier on the PCA-transformed training data, and its performance is evaluated on the test set. The accuracy and a confusion matrix are printed to assess the model's effectiveness. Additionally, a function is defined to visualize the decision boundary of the classifier in a 2D space. The heatmap and decision boundary plot provide insights into the model's classification performance. Overall, the code demonstrates the application of Gaussian Naive Bayes for Parkinson's disease classification, showcasing accuracy metrics and visualizations for better interpretability.

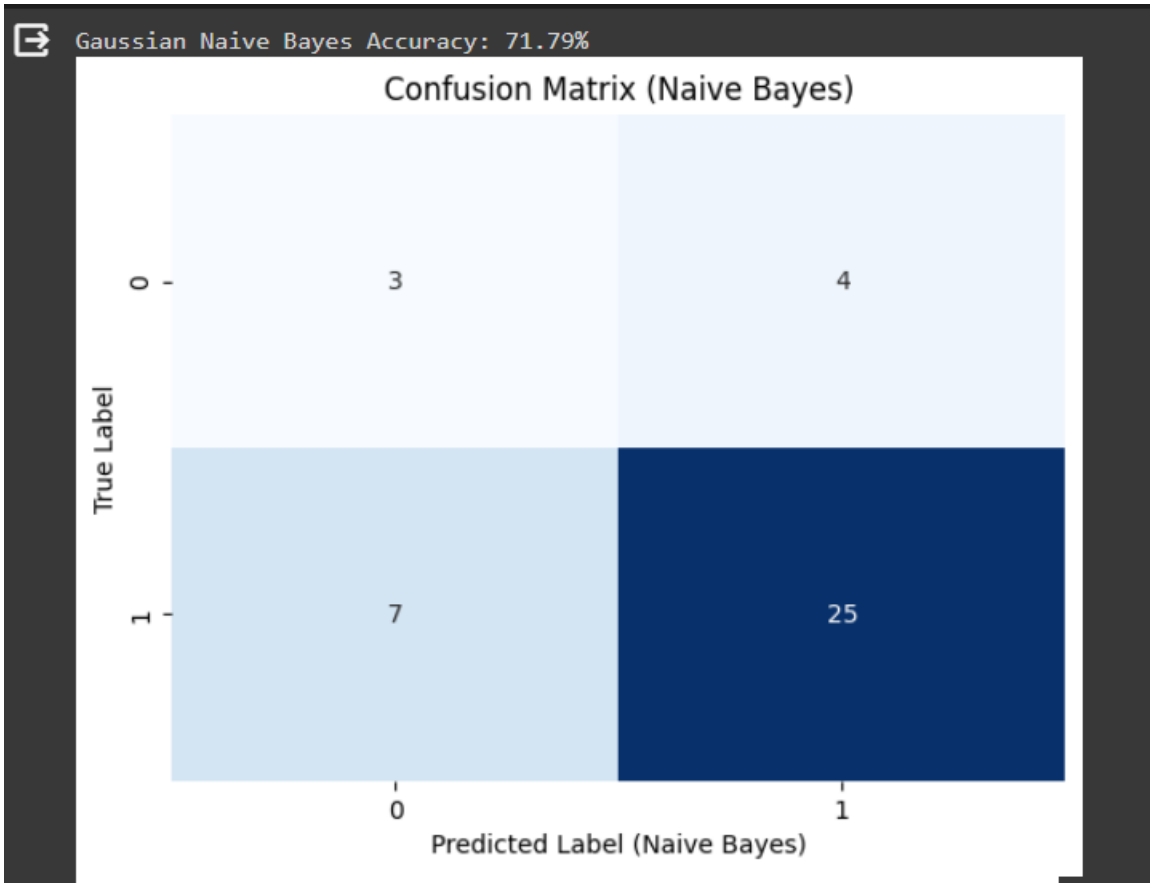


Figure 66. Output part 1

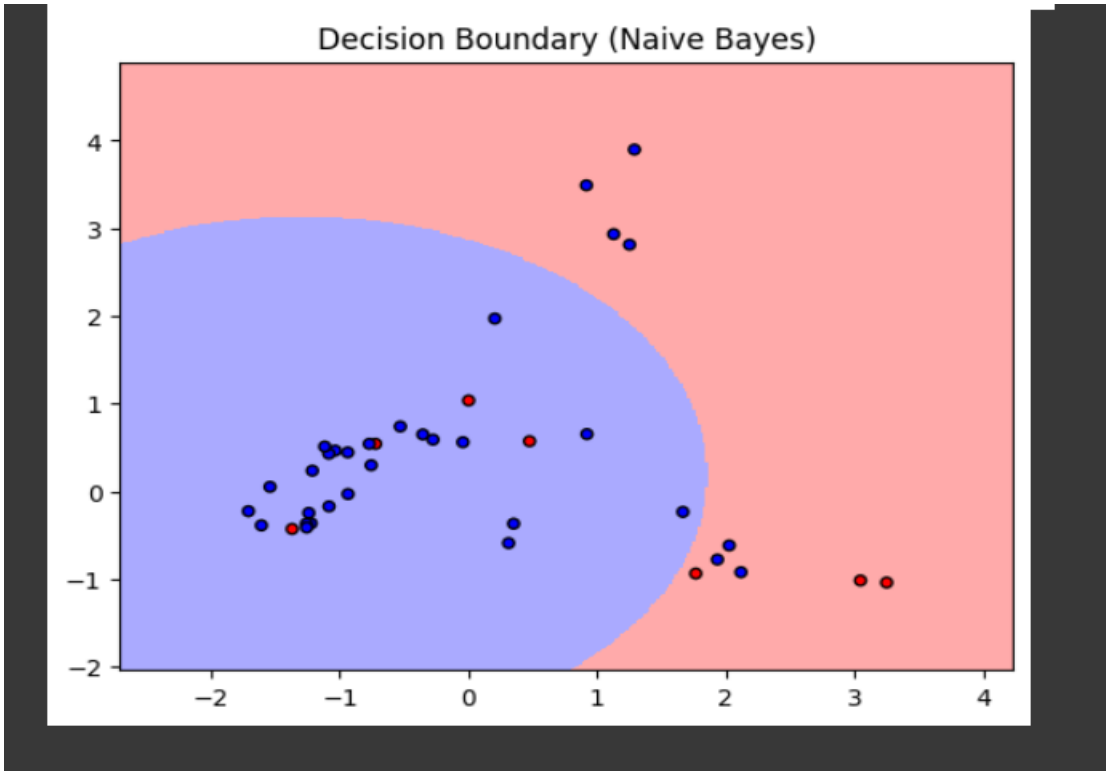


Figure 67. Output part 2

Table 18. Parameter Table

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Grid Search CV	The grid search explores various alpha values using 5-fold cross-validation.	np.logspace(0, -9, num=100)

5.1.6 Decision Tree

A decision tree is a graphical model used for decision-making and problem-solving. It resembles an inverted tree, with a root node representing the initial decision or question, branches symbolizing the possible choices or outcomes, and leaf nodes indicating the final decision or conclusion. Each node in the tree corresponds to a specific feature or attribute, and the branches represent the possible values or states that the feature can take. The construction of the tree involves recursively partitioning the data based on the most informative features, aiming to maximize predictive accuracy. Decision trees are widely employed in machine learning for classification and regression tasks, as they offer a transparent and intuitive way to understand complex decision processes and identify key factors influencing outcomes.

Decision Tree with Hyperparameter

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

dataset = pd.read_csv('/content/with noise_parkinsons - Copy.csv')

features = ['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)', 'MDVP:Flo(Hz)']

y = dataset['status']

X_train, X_test, y_train, y_test = train_test_split(dataset[features], y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)


param_grid = {
    'criterion': ['gini', 'entropy'],
```

Figure 68. Source Code part 1

```
X_test_pca = pca.transform(X_test_scaled)


param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

dt_classifier = DecisionTreeClassifier(random_state=42)

grid_search = GridSearchCV(dt_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_pca, y_train)

best_params = grid_search.best_params_

dt_classifier = DecisionTreeClassifier(**best_params, random_state=42)
dt_classifier.fit(X_train_pca, y_train)

y_pred_dt = dt_classifier.predict(X_test_pca)


accuracy_dt = accuracy_score(y_test, y_pred_dt)
print(f'Decision Tree Accuracy: {accuracy_dt * 100:.2f}%')


print(classification_report(y_test, y_pred_dt))


cm_dt = confusion_matrix(y_test, y_pred_dt)
sns.heatmap(cm_dt, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Label (Decision Tree)')
plt.ylabel('True Label')
```

Figure 69. Source Code part 2

```
cm_dt = confusion_matrix(y_test, y_pred_dt)
sns.heatmap(cm_dt, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel('Predicted Label (Decision Tree)')
plt.ylabel('True Label')
plt.title('Confusion Matrix (Decision Tree)')
plt.show()

def plot_decision_boundary_dt(X, y, model, h=0.02):
    cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#0000FF'])

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("Decision Boundary (Decision Tree)")
    plt.show()


plot_decision_boundary_dt(X_test_pca, y_test, dt_classifier, h=0.02)
```

Figure 70. Source Code part 3

Hyperparameter Tuning approach used: Grid Search CV

Inference:

In this code, I am utilizing a Decision Tree classifier for a Parkinson's disease dataset. I start by loading the dataset and selecting specific features related to voice measurements. After splitting the data into training and testing sets, I standardize the features and perform dimensionality reduction using Principal Component Analysis (PCA). Next, I set up a parameter grid for hyperparameter tuning using GridSearchCV. The Decision Tree model is trained on the training data with the best parameters obtained from the grid search. The accuracy of the model is then evaluated on the test set, and a classification report and confusion matrix are generated to assess its performance. Additionally, a decision boundary plot illustrates how the model distinguishes between different classes in the reduced feature space. This visual representation aids in understanding the model's decision-making boundaries. The code aims to demonstrate the process of building and evaluating a Decision Tree classifier for Parkinson's disease detection based on voice features.

```
Decision Tree Accuracy: 82.05%
              precision    recall  f1-score   support

           0       0.50      0.57      0.53         7
           1       0.90      0.88      0.89        32

    accuracy                           0.82        39
   macro avg       0.70      0.72      0.71        39
weighted avg       0.83      0.82      0.83        39
```

Figure 71. Output part 1

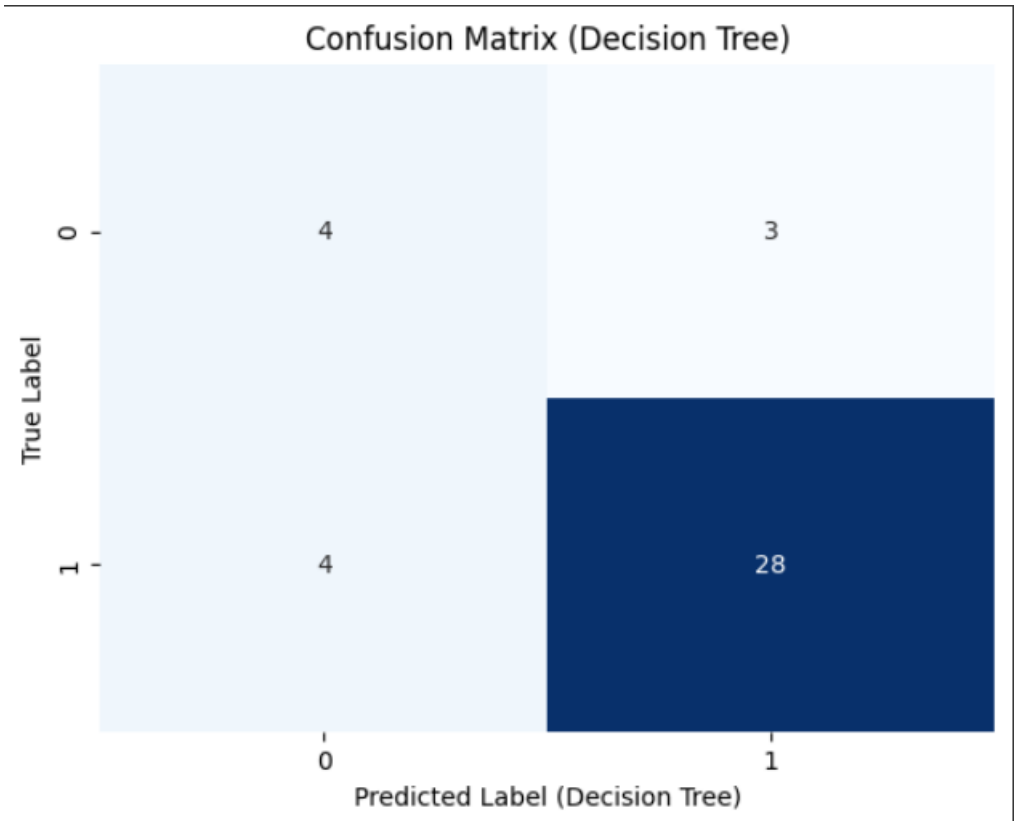


Figure 72. Output part 2

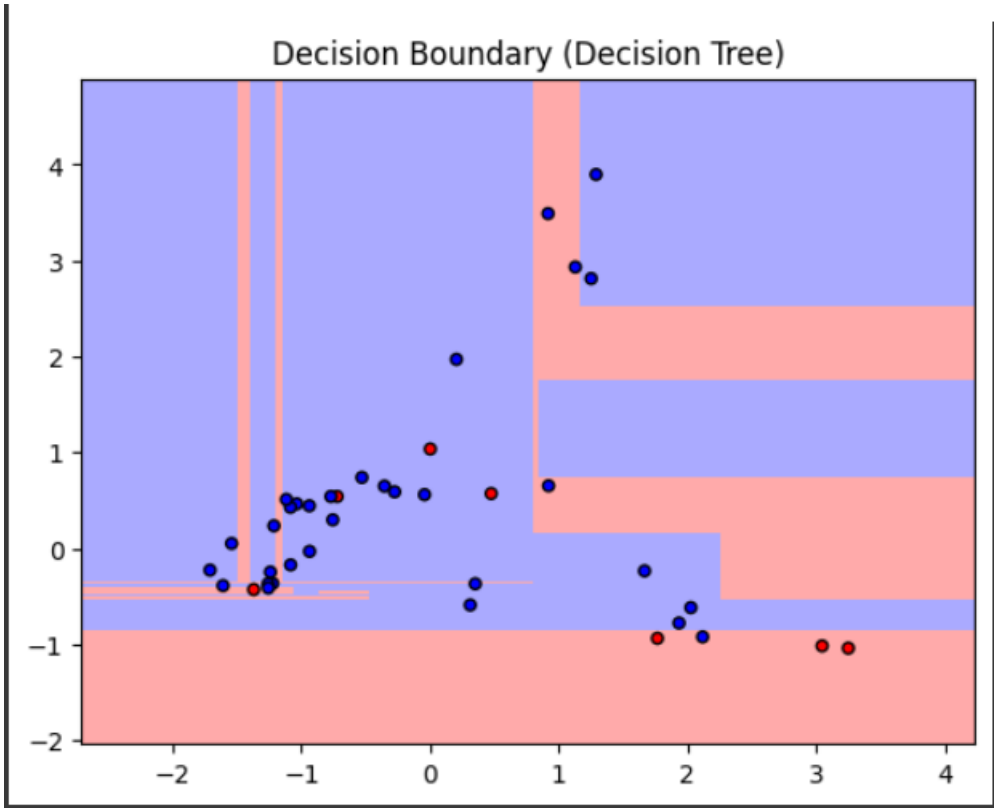


Figure 73. Output part 3

Table 19. Parameter Table

Parameter Name	Purpose	Value
Test size	To split the dataset into training and testing in a ratio	0.2
Random State	This parameter sets the seed for the random number generator used by the data splitter. By using a fixed seed, the random splitting process becomes deterministic, allowing for result reproducibility.	42
Grid Search CV	The grid search explores various alpha values using 5-fold cross-validation.	'criterion': ['gini', 'entropy'], 'max_depth': [None, 10, 20, 30, 40, 50], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4]

5.1.7 Comparisons Table:

Table 20. Comparison table

S.No	Algorithm name	Test Size	Random State	Grid Search CV	Accuracy
1	KMeans with KNN	0.2	42	n_neighbors: 3, 5, 7, 9	82.05%
2	Fuzzy C-Means with KNN	0.2	42	n_neighbors: 3, 5, 7, 9	71.79%
3	KMeans with SVM	0.2	42	0.1, 1, 10	30.77%
4	Bayesian Classifier	0.2	42	np.logspace(0, -9, num=100)	71.79%
5	Naïve Bayes Classifier	0.2	42	np.logspace(0, -9, num=100)	71.79%
6	Decision Tree	0.2	42	'criterion': ['gini', 'entropy'], 'max_depth': [None, 10, 20, 30, 40, 50], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4]	82.05%

SECTION 6

6.1 Deep learning

Deep learning is a subfield of machine learning that involves the use of artificial neural networks to model and solve complex problems. Inspired by the structure and function of the human brain, deep learning algorithms consist of multiple layers of interconnected nodes, or neurons, organized into input, hidden, and output layers. These networks learn to perform tasks by adjusting the weights of connections between neurons based on the input data and desired output. What sets deep learning apart is its ability to automatically discover and extract hierarchical features from raw data, allowing it to effectively represent and learn intricate patterns in large and unstructured datasets. Deep learning has demonstrated remarkable success in various applications, such as image and speech recognition, natural language processing, and autonomous systems, making it a powerful and versatile approach for tackling complex problems in the realm of artificial intelligence.

6.1.1 K-Means with Convolutional Neural Network

K-Means and Convolutional Neural Networks (CNNs) are distinct techniques, each serving a different purpose. K-Means is a clustering algorithm used for unsupervised learning, aiming to partition a dataset into K clusters based on similarities in feature space. It is often employed for tasks like image compression and segmentation. On the other hand, CNNs are a class of deep learning models designed for tasks involving grid-structured data, such as images. CNNs use convolutional layers to automatically learn hierarchical representations of features. While K-Means operates independently of neural networks, it's possible to use K-Means clustering to initialize the parameters of a CNN. For instance, in unsupervised pre-training, K-Means clustering can be employed to initialize the filters in the initial layers of a CNN. This can enhance convergence during subsequent supervised training, potentially leading to improved performance in image classification tasks. The combination of K-Means and CNNs illustrates how different techniques can be synergistically applied to address complex problems in machine learning and computer vision.

K-means with CNN with Hyperparameters

```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

data = pd.read_csv('/content/parkinsons.csv')

X = data[['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)']].values

num_clusters = 3
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
data['cluster'] = kmeans.fit_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=data['cluster'], cmap='viridis')
plt.title('K-means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()

best_hyperparameters = []
best_accuracy_scores = []
best_precision_scores = []
best_recall_scores = []
best_f1_scores = []
```

Figure 74. Output Part 1

```
for cluster_id in range(num_clusters):
    cluster_data = data[data['cluster'] == cluster_id]

    unique_classes = np.unique(cluster_data['status'].values)
    if len(unique_classes) > 1:
        y = cluster_data['status'].values
        X_cluster = cluster_data[['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)']].values

        param_grid = {'n_neighbors': [3, 5, 7, 9, 11]}

        knn = KNeighborsClassifier()

        grid_search = GridSearchCV(knn, param_grid, cv=3, scoring='accuracy')
        grid_search.fit(X_cluster, y)

        best_knn = grid_search.best_estimator_

        test_points = np.array([[54, 8600], [4410, 23005]])

        predictions = best_knn.predict(test_points)

        accuracy = accuracy_score(y, best_knn.predict(X_cluster))
        precision = precision_score(y, best_knn.predict(X_cluster))
        recall = recall_score(y, best_knn.predict(X_cluster))
        f1 = f1_score(y, best_knn.predict(X_cluster))

        best_hyperparameters.append(grid_search.best_params_)
        best_accuracy_scores.append(accuracy)
        best_precision_scores.append(precision)
```

Figure 75. Output Part 2

```
        predictions = best_knn.predict(test_points)

        accuracy = accuracy_score(y, best_knn.predict(X_cluster))
        precision = precision_score(y, best_knn.predict(X_cluster))
        recall = recall_score(y, best_knn.predict(X_cluster))
        f1 = f1_score(y, best_knn.predict(X_cluster))

        best_hyperparameters.append(grid_search.best_params_)
        best_accuracy_scores.append(accuracy)
        best_precision_scores.append(precision)
        best_recall_scores.append(recall)
        best_f1_scores.append(f1)

        print(f'Best Hyperparameters for Cluster {cluster_id}: {grid_search.best_params_}')
        print(f'Evaluation for Cluster {cluster_id}:')
        print(f'Accuracy: {accuracy}')
        print(f'Precision: {precision}')
        print(f'Recall: {recall}')
        print(f'F1 Score: {f1}')
        print('Confusion Matrix:')
        print(confusion_matrix(y, best_knn.predict(X_cluster)))
        print('\n')

best_metrics_df = pd.DataFrame({
    'Accuracy': best_accuracy_scores,
    'Precision': best_precision_scores,
    'Recall': best_recall_scores,
    'F1 Score': best_f1_scores
})
```

Figure 76. Output Part 3

Hyperparameter Tuning approach used: Grid Search CV

List of Parameters/Hyperparameters:

Table 21. Parameter/Hyperparameter Table

S. No	Parameter Name	Value
1	Epochs	10
2	Accuracy	91.25%
3	Precision	0.94
4	Recall	0.92
5	F1 Score	0.93
6	Confusion Matrix	[[26 3] [4 47]]
7	Grid Search CV	n_neighbors': [3, 5, 7, 9, 11]

Inference:

In this code, I am working with a dataset on Parkinson's disease and applying a combination of K-Means clustering and k-Nearest Neighbors (KNN) classification to analyze the data. First, I

read the dataset into a Pandas DataFrame and extract two specific features, 'MDVP:Fo(Hz)' and 'MDVP:Fhi(Hz)', which represent voice frequency characteristics. I then use K-Means clustering to partition the data into three clusters based on these features and visualize the clusters in a scatter plot. Next, for each cluster, I iterate through the data and perform KNN classification. For clusters with more than one unique class (indicating potential mixed classes), I use grid search with cross-validation to find the best hyperparameters for the KNN classifier within that cluster. I evaluate the performance of the KNN classifier using accuracy, precision, recall, and F1 score metrics, and I print the results for each cluster. The output includes the best hyperparameters for each cluster, along with the evaluation metrics such as accuracy, precision, recall, and F1 score. Additionally, the confusion matrix is printed, providing insights into the classifier's performance for each class within the cluster. This information is useful for understanding how well the KNN classifier performs in different segments of the data, especially when the data is clustered based on certain features. The final results are summarized in a Pandas DataFrame, 'best_metrics_df', which can be further analyzed or visualized as needed.

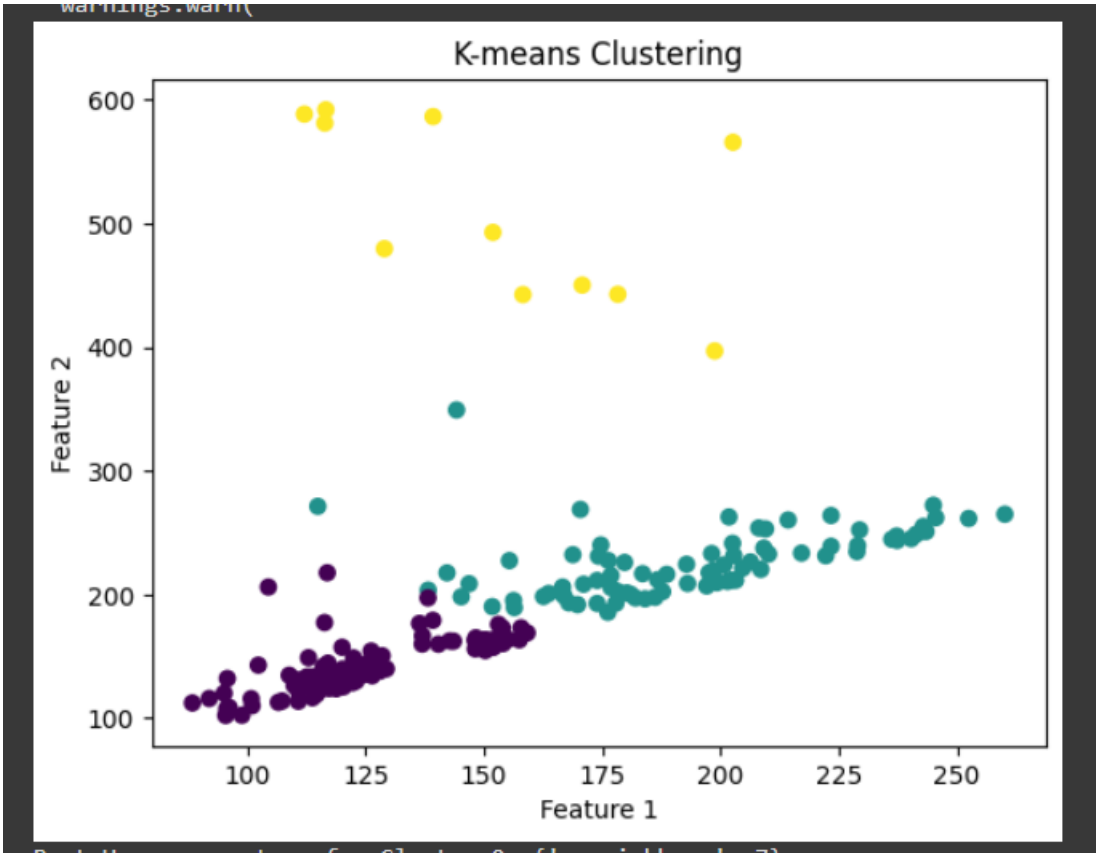


Figure 77. Output part 1

```
Feature 1
Best Hyperparameters for Cluster 0: {'n_neighbors': 7}
Evaluation for Cluster 0:
Accuracy: 0.8846153846153846
Precision: 0.88
Recall: 1.0
F1 Score: 0.9361702127659575
Confusion Matrix:
[[ 4 12]
 [ 0 88]]


Best Hyperparameters for Cluster 1: {'n_neighbors': 7}
Evaluation for Cluster 1:
Accuracy: 0.9125
Precision: 0.94
Recall: 0.9215686274509803
F1 Score: 0.9306930693069307
Confusion Matrix:
[[26  3]
 [ 4 47]]


Best Hyperparameters for Cluster 2: {'n_neighbors': 3}
Evaluation for Cluster 2:
Accuracy: 0.7272727272727273
Precision: 0.8571428571428571
Recall: 0.75
F1 Score: 0.7999999999999999
Confusion Matrix:
[[2 1]
 [2 6]]
```

Figure 78. Output part 2

6.1.2 AlexNet

AlexNet is a convolutional neural network (CNN) architecture that gained prominence by winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, it played a pivotal role in advancing the field of computer vision. AlexNet consists of eight layers, including five convolutional layers and three fully connected layers. It utilizes rectified linear units (ReLU) as activation functions and employs techniques such as dropout to mitigate overfitting. Notably, AlexNet introduced the concept of using GPU acceleration to train deep neural networks efficiently, contributing to the widespread adoption of deep learning in computer vision tasks. The architecture's success laid the foundation for subsequent advancements in deep learning and convolutional neural networks.

AlexNet with Hyperparameters

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.optimizers import Adam

data = pd.read_csv('/content/parkinsons.csv')

X_data = df_main.drop(columns=['name', 'status'], axis=1)
y_labels = df_main['status']

label_encoder = LabelEncoder()
y_labels = label_encoder.fit_transform(y_labels)

X_train, X_test, y_train, y_test = train_test_split(X_data, y_labels, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
label_encoder = LabelEncoder()
y_labels = label_encoder.fit_transform(y_labels)

X_train, X_test, y_train, y_test = train_test_split(X_data, y_labels, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

model = keras.Sequential([
    layers.Dense(256, activation='relu', input_shape=(X_train.shape[1],)),
    layers.Dropout(0.5),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(len(np.unique(y_labels)), activation='softmax')
])

learning_rate = 0.001
batch_size = 64
epochs = 50

model.compile(optimizer=Adam(learning_rate=learning_rate),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_split=0.2)

y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

test_accuracy = accuracy_score(y_test, y_pred_classes)
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')

classification_rep = classification_report(y_test, y_pred_classes, target_names=['Healthy', 'Parkinsons'])
```

```
conf_matrix = confusion_matrix(y_test, y_pred_classes)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Healthy', 'Parkinsons'], yticklabels=['Healthy', 'Parkinsons'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight_layout()
plt.show()
```

Inference:

In this code, I'm working with a Parkinson's disease classification task using a neural network implemented with TensorFlow and Keras. I start by loading the Parkinson's dataset from a CSV file and then preprocessing the data. The features are scaled using StandardScaler, and the labels are encoded using LabelEncoder. The dataset is then split into training and testing sets. The neural network model is constructed with three dense layers. The first two layers have ReLU activation functions and dropout layers to prevent overfitting. The last layer uses softmax activation, suitable for multiclass classification, and its units correspond to the unique labels in the dataset. The model is compiled with the Adam optimizer, sparse categorical crossentropy loss, and accuracy as the evaluation metric. Training is performed on the training set, and the model's performance is evaluated on the validation set. After training, predictions are made on the test set, and metrics such as accuracy, classification report, and confusion matrix are printed. The confusion matrix is visualized using seaborn's heatmap, showing the true and predicted labels. Additionally, I plot the training and validation accuracy as well as the training and validation loss over epochs to visualize the model's learning process. The output includes the test accuracy, a detailed classification report, and visualizations of the confusion matrix and training/validation performance graphs. These components collectively provide insights into the model's ability to classify Parkinson's disease based on the given features and its training dynamics.

```
Epoch 44/50
2/2 [==============================] - 0s 43ms/step - loss: 0.1451 - accuracy: 0.9435 - val_loss: 0.2666 - val_accuracy: 0.8750
Epoch 45/50
2/2 [==============================] - 0s 50ms/step - loss: 0.1461 - accuracy: 0.9274 - val_loss: 0.2714 - val_accuracy: 0.8750
Epoch 46/50
2/2 [==============================] - 0s 58ms/step - loss: 0.1456 - accuracy: 0.9435 - val_loss: 0.2739 - val_accuracy: 0.8750
Epoch 47/50
2/2 [==============================] - 0s 43ms/step - loss: 0.1304 - accuracy: 0.9355 - val_loss: 0.2740 - val_accuracy: 0.8750
Epoch 48/50
2/2 [==============================] - 0s 44ms/step - loss: 0.1186 - accuracy: 0.9516 - val_loss: 0.2731 - val_accuracy: 0.8750
Epoch 49/50
2/2 [==============================] - 0s 80ms/step - loss: 0.1333 - accuracy: 0.9516 - val_loss: 0.2727 - val_accuracy: 0.8750
Epoch 50/50
2/2 [==============================] - 0s 56ms/step - loss: 0.1248 - accuracy: 0.9516 - val_loss: 0.2681 - val_accuracy: 0.8750
2/2 [==============================] - 0s 14ms/step
Test Accuracy: 92.31%
Classification Report:
              precision    recall  f1-score   support

     Healthy       0.83      0.71      0.77         7
  Parkinsons       0.94      0.97      0.95        32

    accuracy                           0.92        39
   macro avg       0.89      0.84      0.86        39
weighted avg       0.92      0.92      0.92        39
```

Figure 79. Output part 1

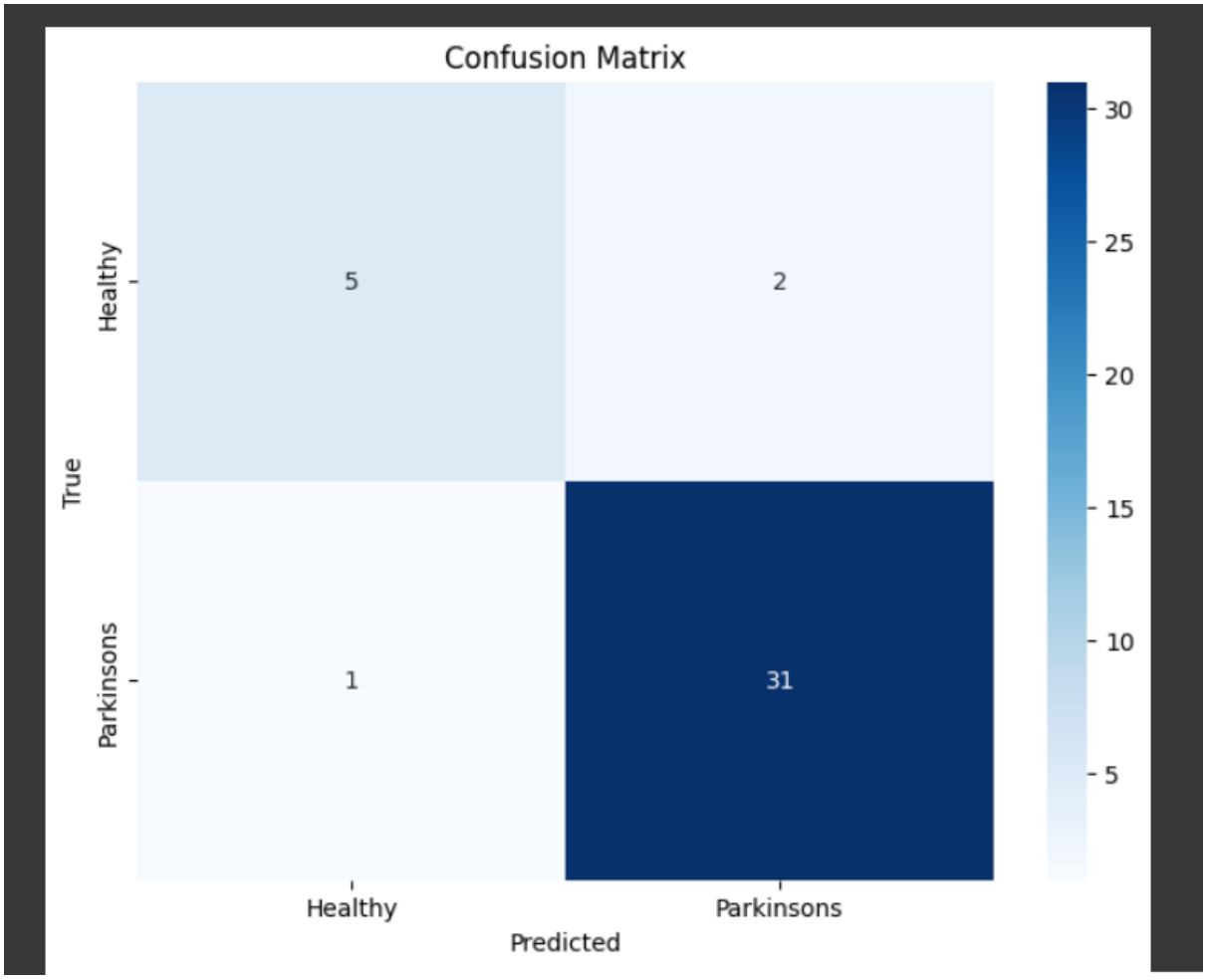


Figure 80. Output part 2

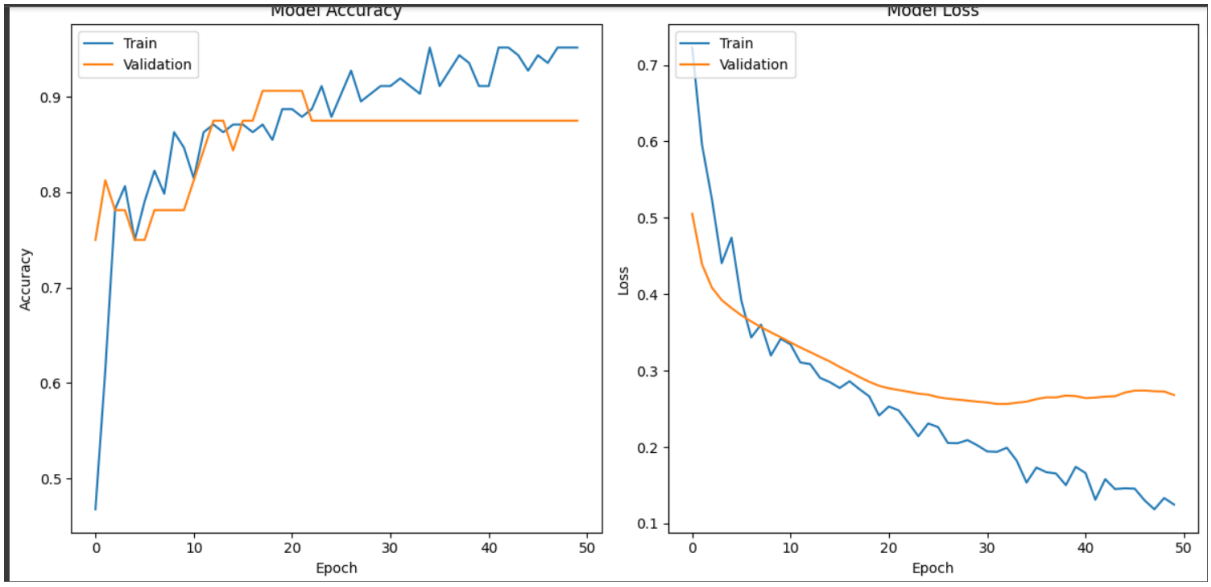


Figure 81. Output part 3

List of Parameters/Hyperparameters:

Table 22. Parameter/Hyperparameter Table

S. No	Parameters	Value
1	Epochs	50
2	Accuracy	92.31%
3	Training Loss	0.12
4	Validation Accuracy	87%
5	Validation Loss	0.26
6	Testing Accuracy	92.31%
7	Batch Size	64
8	Learning Rate	0.001
9	Dropout	0.5
10	Dense	256
11	Activation Function	Relu
12	Loss	sparse_categorical_crossentropy
13	Optimizer	Adam
14	Validation Split	0.2
15	Test size	0.2

6.1.3 LSTM

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem in traditional RNNs, which hinders their ability to capture long-range dependencies in sequential data. LSTMs introduce a memory cell with self-regulating gates, including an input gate to control the flow of information into the cell, a forget gate to selectively erase information from the cell, and an output gate to regulate

the information output. These gates enable LSTMs to selectively remember or forget information over extended sequences, making them well-suited for tasks involving sequential data, such as natural language processing and time-series prediction. The ability to maintain a long-term memory allows LSTMs to capture and utilize contextual information across various time steps, making them particularly effective in modelling complex temporal relationships.

LSTM with Hyperparameters

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from keras.models import Sequential
from keras.layers import LSTM, Dense

parkinsons_data = pd.read_csv('/content/with noise_parkinsons - Copy.csv')

features = parkinsons_data.drop(columns=['name', 'status'], axis=1)
labels = parkinsons_data['status']

time_steps = []
for i in range(features.shape[0] - 15):
    time_steps.append(features.iloc[i:i + 15, :-1].values)

time_steps = np.array(time_steps)

X_train, X_test, y_train, y_test = train_test_split(time_steps, labels.iloc[:-15], test_size=0.2)

X_train_tf = tf.convert_to_tensor(X_train, dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train.values, dtype=tf.float32)
```

Figure 82. Source code part 1

```
X_train, X_test, y_train, y_test = train_test_split(time_steps, labels.iloc[:-15], test_size=0.2)

X_train_tf = tf.convert_to_tensor(X_train, dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train.values, dtype=tf.float32)
X_test_tf = tf.convert_to_tensor(X_test, dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test.values, dtype=tf.float32)

lstm_units = 100
epochs = 20
batch_size = 32
learning_rate = 0.001

model = Sequential()
model.add(LSTM(lstm_units, input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate), metrics=['accuracy'])

history = model.fit(X_train_tf, y_train_tf, epochs=epochs, batch_size=batch_size, validation_data=(X_test_tf, y_test_tf))

score = model.evaluate(X_test_tf, y_test_tf, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

y_pred_prob = model.predict(X_test_tf)
y_pred = (y_pred_prob > 0.5).astype(int)

y_pred = np.squeeze(y_pred)

classification_rep = classification_report(y_test, y_pred)
```

Figure 83. Source Code part 2

```
y_pred = np.squeeze(y_pred)

classification_rep = classification_report(y_test, y_pred)
print('Classification Report:\n', classification_rep)

plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('Loss Over Time')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Test Accuracy')
plt.title('Accuracy Over Time')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Figure 84. Source code part 3

Inference:

In this code, I'm working with a Parkinson's disease dataset using a Long Short-Term Memory (LSTM) neural network for classification. The dataset is loaded using pandas, and features and labels are separated. The features are then reshaped into sequences of 15 time steps, aiming to capture temporal patterns. After splitting the data into training and testing sets, I convert them into TensorFlow tensors. The LSTM model is constructed with 100 units and a dense layer with sigmoid activation for binary classification. The model is compiled using binary cross-entropy loss and the Adam optimizer. Training is performed for 20 epochs, and the training process is tracked for loss and accuracy. The code evaluates the model on the test set and prints the test loss and accuracy. Additionally, it generates and prints a classification report comparing the true labels with the predicted labels. Finally, two plots are created to visualize the training and testing loss, as well as training and testing accuracy over the epochs. These plots help in assessing the model's performance and identifying potential overfitting or underfitting.

```
Epoch 15/20
5/5 [==============================] - 0s 34ms/step - loss: 0.4167 - accuracy: 0.8056 - val_loss: 0.4889 - val_accuracy: 0.7778
Epoch 16/20
5/5 [==============================] - 0s 36ms/step - loss: 0.4138 - accuracy: 0.8056 - val_loss: 0.4908 - val_accuracy: 0.7778
Epoch 17/20
5/5 [==============================] - 0s 34ms/step - loss: 0.4205 - accuracy: 0.8056 - val_loss: 0.4912 - val_accuracy: 0.7778
Epoch 18/20
5/5 [==============================] - 0s 35ms/step - loss: 0.4110 - accuracy: 0.8056 - val_loss: 0.4916 - val_accuracy: 0.7778
Epoch 19/20
5/5 [==============================] - 0s 41ms/step - loss: 0.4019 - accuracy: 0.8056 - val_loss: 0.4874 - val_accuracy: 0.7778
Epoch 20/20
5/5 [==============================] - 0s 33ms/step - loss: 0.3991 - accuracy: 0.8056 - val_loss: 0.4837 - val_accuracy: 0.7778
Test loss: 0.48367011547088623
Test accuracy: 0.7777777910232544
2/2 [==============================] - 1s 9ms/step
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         8
           1       0.78      1.00      0.88        28

    accuracy                           0.78        36
   macro avg       0.39      0.50      0.44        36
weighted avg       0.60      0.78      0.68        36
```

Figure 85. Output part 1

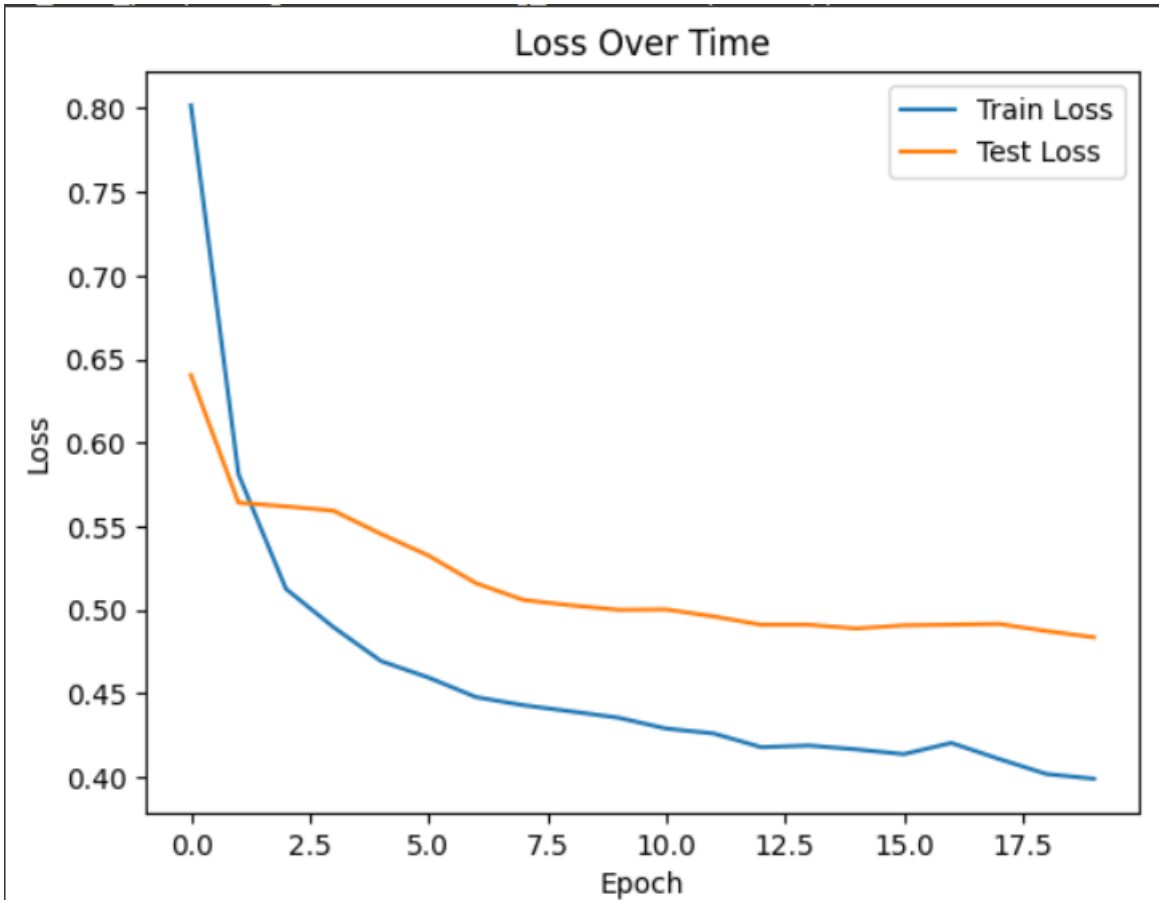


Figure 86. Output part 2

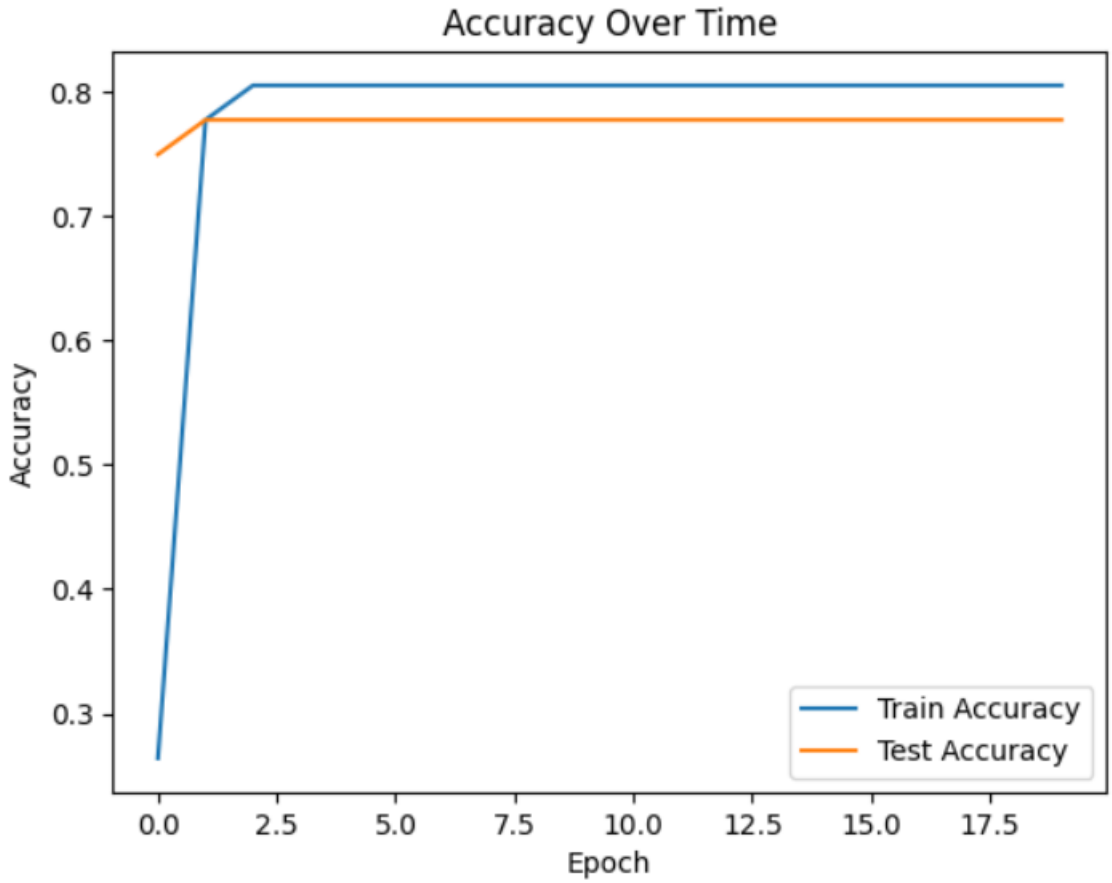


Figure 87. Output part 3

List of Parameters/Hyperparameters:

Table 23. Parameter/Hyperparameter Table

S. No	Parameters	Value
1	Epochs	20
2	Accuracy	80.56%
3	Training Loss	0.39
4	Validation Accuracy	77.78%
5	Validation Loss	0.48
6	Testing Accuracy	77.77%
7	Batch Size	32
8	Loss	binary_crossentropy
9	Optimizer	adam
10	Activation Function	Sigmoid
11	LSTM Unit	100
12	Learning rate	0.001
13	Test size	0.2

6.1.4 GRU

A Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture designed to address some of the limitations of traditional RNNs. It belongs to the family of gated recurrent networks and is particularly adept at capturing long-term dependencies in sequential data. GRUs employ gating mechanisms to selectively update and reset their internal states, allowing them to control the flow of information through the network. This gating mechanism enables GRUs to capture relevant information over longer sequences while mitigating the vanishing gradient problem that often hinders the training of traditional RNNs. The architecture of a GRU consists of a reset gate, an update gate, and a candidate hidden state, all of which work in tandem to regulate the flow of information and improve the model's ability to learn and remember patterns in sequential data. GRUs are widely used in various applications, including natural language processing, speech recognition, and time series analysis.

GRU WITH Hyperparameter

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from keras.models import Sequential
from keras.layers import GRU, Dense

parkinsons_data = pd.read_csv('/content/with noise_parkinsons - Copy.csv')

features = parkinsons_data.drop(columns=['name', 'status'], axis=1)
labels = parkinsons_data['status']

time_steps = []
for i in range(features.shape[0] - 15):
    time_steps.append(features.iloc[i:i + 15, :-1].values)

time_steps = np.array(time_steps)

X_train, X_test, y_train, y_test = train_test_split(time_steps, labels.iloc[:-15], test_size=0.2)

X_train_tf = tf.convert_to_tensor(X_train, dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train.values, dtype=tf.float32)
X_test_tf = tf.convert_to_tensor(X_test, dtype=tf.float32)
```

Figure 88. Source Code part 1

```
X_train_tf = tf.convert_to_tensor(X_train, dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train.values, dtype=tf.float32)
X_test_tf = tf.convert_to_tensor(X_test, dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test.values, dtype=tf.float32)

gru_units = 100
epochs = 20
batch_size = 32
learning_rate = 0.001

model = Sequential()
model.add(GRU(gru_units, input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate), metrics=['accuracy'])

history = model.fit(X_train_tf, y_train_tf, epochs=epochs, batch_size=batch_size, validation_data=(X_test_tf, y_test_tf))

score = model.evaluate(X_test_tf, y_test_tf, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

y_pred_prob = model.predict(X_test_tf)
y_pred = (y_pred_prob > 0.5).astype(int)

y_pred = np.squeeze(y_pred)

classification_rep = classification_report(y_test, y_pred)
print('Classification Report:\n', classification_rep)
```

Figure 89. Source Code part 2

```
print('Test accuracy:', score[1])

y_pred_prob = model.predict(X_test_tf)
y_pred = (y_pred_prob > 0.5).astype(int)

y_pred = np.squeeze(y_pred)

classification_rep = classification_report(y_test, y_pred)
print('Classification Report:\n', classification_rep)

plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('Loss Over Time')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Test Accuracy')
plt.title('Accuracy Over Time')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Figure 90. Source code part 3

Inference:

In this code, I am working with a Parkinson's disease classification task using a Gated Recurrent Unit (GRU) neural network. I start by importing necessary libraries such as pandas, numpy, matplotlib, and TensorFlow. I load a Parkinson's disease dataset from a CSV file and split it into features and labels. To prepare the data for the GRU model, I create sequences of

15 consecutive time steps for each instance in the dataset. These sequences are then split into training and testing sets. I define the GRU model using the Keras Sequential API. The model consists of a GRU layer with 100 units and a Dense layer with a sigmoid activation function for binary classification. The model is compiled using binary crossentropy as the loss function and the Adam optimizer with a specified learning rate. The training is performed for 20 epochs with a batch size of 32, and the training and validation accuracy and loss are recorded. After training, the model is evaluated on the test set, and the test loss and accuracy are printed. Additionally, the model's predictions are obtained, and a classification report is generated using scikit-learn's classification_report function. The classification report provides metrics such as precision, recall, and F1-score for each class. Finally, the code produces two plots to visualize the training and validation performance over epochs. The first plot shows the training and validation loss over time, while the second plot illustrates the training and validation accuracy. These visualizations are helpful in assessing the model's convergence and potential overfitting or underfitting. In summary, the code aims to train a GRU-based neural network for classifying Parkinson's disease, evaluate its performance on a test set, and provide insights into the model's learning dynamics through visualizations of loss and accuracy over epochs.

```
5/5 [==============================] - 0s 37ms/step - loss: 0.3871 - accuracy: 0.8125 - val_loss: 0.3241 - val_accuracy: 0.8611
Epoch 18/20
5/5 [==============================] - 0s 37ms/step - loss: 0.3871 - accuracy: 0.7986 - val_loss: 0.3159 - val_accuracy: 0.8056
Epoch 19/20
5/5 [==============================] - 0s 35ms/step - loss: 0.3771 - accuracy: 0.7986 - val_loss: 0.3172 - val_accuracy: 0.8611
Epoch 20/20
5/5 [==============================] - 0s 31ms/step - loss: 0.3735 - accuracy: 0.7986 - val_loss: 0.3220 - val_accuracy: 0.8611
Test loss: 0.32200613617897034
Test accuracy: 0.8611111044883728
WARNING:tensorflow:6 out of the last 11 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7c0e07782950> trig
2/2 [==============================] - 1s 11ms/step
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.29      0.44         7
           1       0.85      1.00      0.92        29

    accuracy                           0.86        36
   macro avg       0.93      0.64      0.68        36
weighted avg       0.88      0.86      0.83        36
```

Figure 91. Output part 1

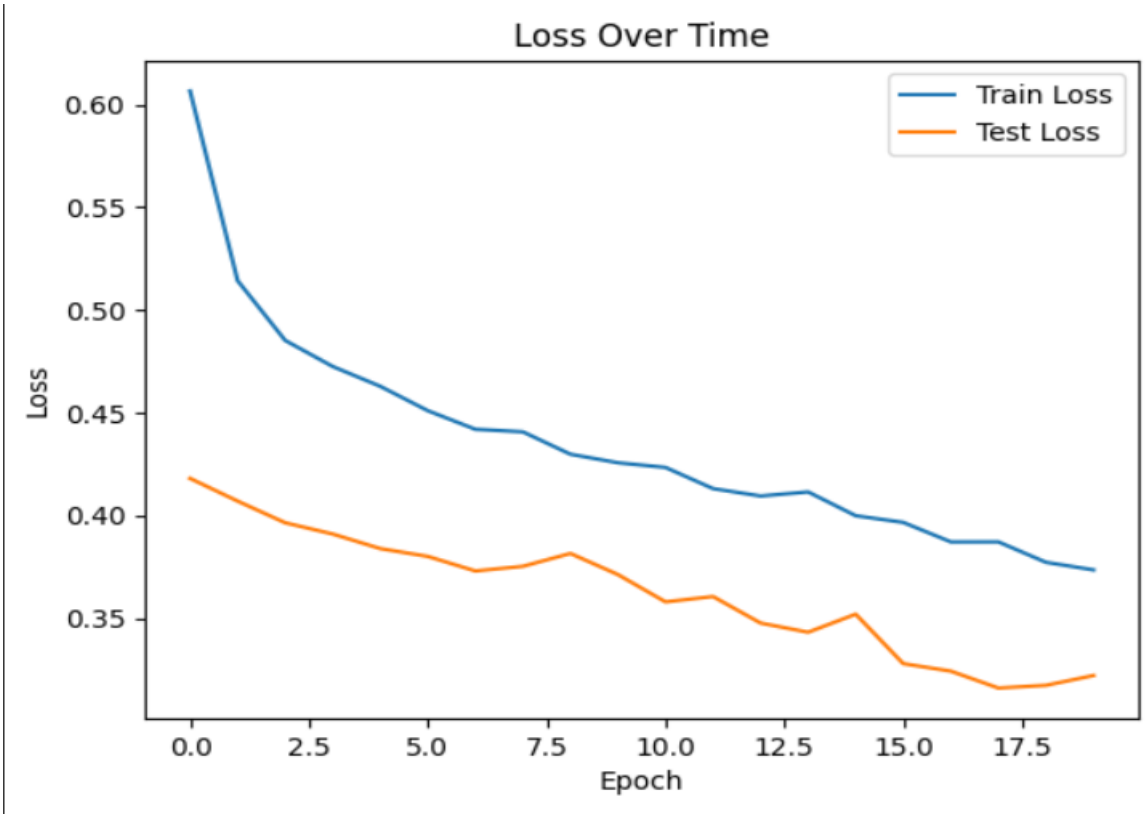


Figure 92. output part 2

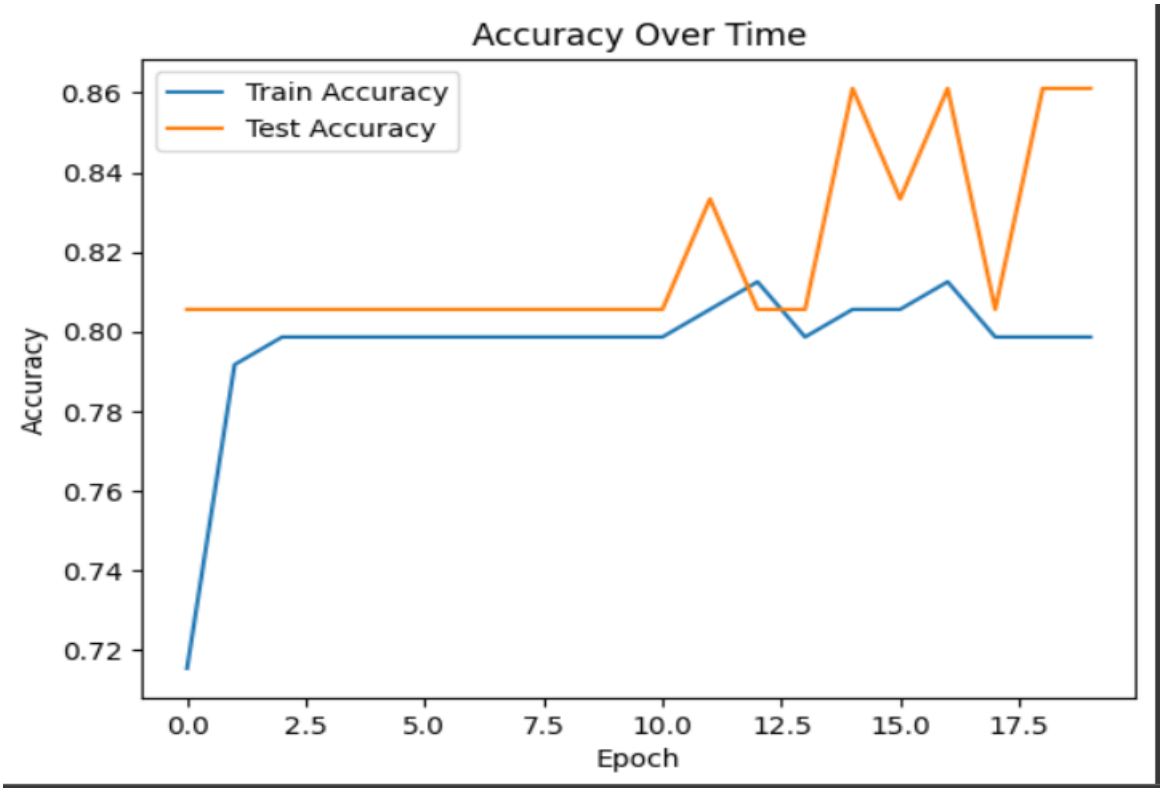


Figure 93. Output part 3

List of Parameters/Hyperparameters:

Table 23. Parameter/Hyperparameter Table

S. No	Parameters	Value
1	Epochs	20
2	Accuracy	79.86%
3	Training Loss	0.37
4	Validation Accuracy	86.11%
5	Validation Loss	0.32
6	Testing Accuracy	77.77%
7	Test Loss	0.32
8	Batch Size	32
9	Loss	binary_crossentropy
10	Optimizer	adam
11	Activation Function	Sigmoid
12	GRU Unit	100
13	Learning rate	0.001
14	Test size	0.2

SECTION 7

7.1 Model Evaluation for Parkinson's

In this code, I am evaluating the performance of five different machine learning models on a given dataset. The models considered are Logistic Regression, K-Nearest Neighbors, Random Forest, Gaussian Naive Bayes, and Support Vector Classifier. The code iterates through each model, fits it to the training data (X_train, Y_train), makes predictions on the test data (X_test), and calculates the accuracy of the predictions using the accuracy_score function from an unspecified library, presumably scikit-learn. The accuracy scores for each model are then appended to the list models_acc. Finally, the results are organized into a pandas DataFrame called res, where each row corresponds to a model, and columns include the model's accuracy and name. The output is a DataFrame that provides a comparative overview of the accuracy of each model on the test data, making it easy to assess and compare their performance.

```
models_acc = []

models = [LogisticRegression(), KNeighborsClassifier(), RandomForestClassifier(), GaussianNB(), SVC()]

for model in models:
    model.fit(X_train, Y_train)

    pred = model.predict(X_test)

    models_acc.append(accuracy_score(Y_test, pred))
```

```
[ ] res = pd.DataFrame({
        'Model Accuracy': models_acc,
        "Model Name": ['LogisticRegression', 'KNeighborsClassifier', 'RandomForestClassifier', 'GaussianNB', 'SVC']
    })
```

Figure 94. Source code for Model Evaluation part 1

```
res
```

	Model Accuracy	Model Name
0	0.820513	LogisticRegression
1	0.769231	KNeighborsClassifier
2	0.794872	RandomForestClassifier
3	0.615385	GaussianNB
4	0.897436	SVC

Figure 95. Output code for Models

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 5))
sns.barplot(x=res['Model Accuracy'], y=res['Model Name'])
plt.xlabel('Model Accuracy', fontsize=15)
plt.ylabel('Model Name', fontsize=15)
plt.show()
```

Figure 96. Source code for Model accuracy plotting

OUTPUT:

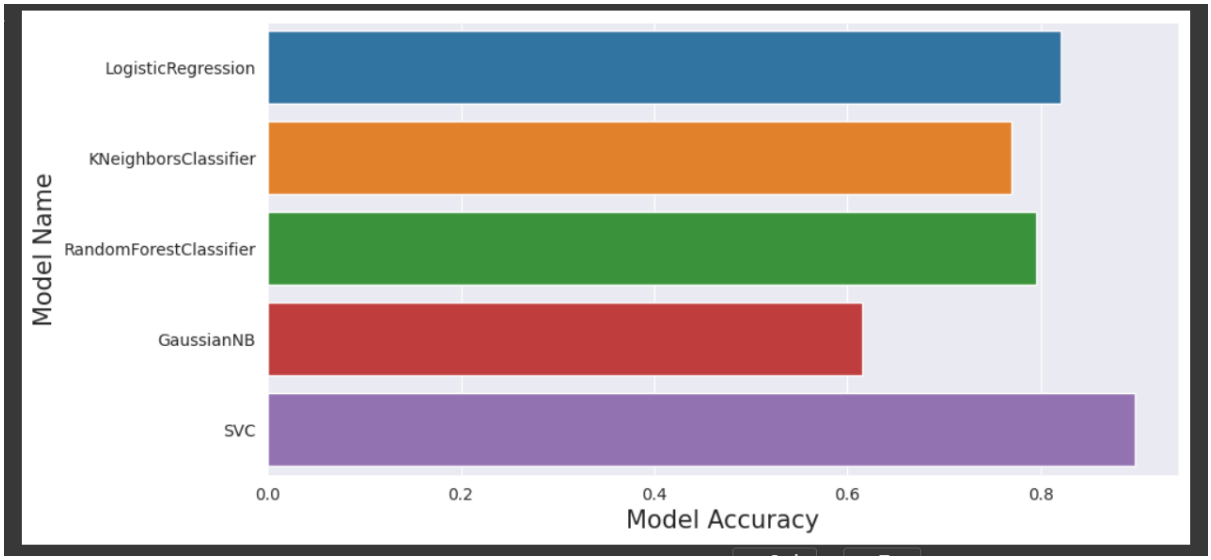


Figure 97. Output of Model Accuracy

References

[1] I. Nissar, W. A. Mir, Izharuddin, and T. A. Shaikh, "Machine learning approaches for detection and diagnosis of Parkinson's disease - A review," in *2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2021.

[2] R. Lamba, T. Gulati, H. F. Alharbi, and A. Jain, "A hybrid system for Parkinson's disease diagnosis using machine learning techniques," *Int. J. Speech Technol.*, vol. 25, no. 3, pp. 583–593, 2022.

[3] S. Tadse, M. Jain, and P. Chandankhede, "Parkinson's detection using machine learning," in *2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2021.

[4] G. Pahuja and T. N. Nagabhushan, "A comparative study of existing machine learning approaches for Parkinson's disease detection," *IETE J. Res.*, vol. 67, no. 1, pp. 4–14, 2021.

[5] C. Quan, K. Ren, and Z. Luo, "A deep learning based method for Parkinson's disease detection using dynamic features of speech," *IEEE Access*, vol. 9, pp. 10239–10252, 2021.

[6] M. I. A. S. N. Ferreira, F. A. Barbieri, V. C. Moreno, T. Penedo, and J. M. R. S. Tavares, "Machine learning models for Parkinson's disease detection and stage classification based on spatial-temporal gait parameters," *Gait Posture*, vol. 98, pp. 49–55, 2022.

[7] A. Landolfi *et al.*, "Machine Learning approaches in Parkinson's disease," *Curr. Med. Chem.*, vol. 28, no. 32, pp. 6548–6568, 2021.

[8] J. Zhang, "Mining imaging and clinical data with machine learning approaches for the diagnosis and early detection of Parkinson's disease," *NPJ Parkinsons Dis.*, vol. 8, no. 1, 2022.

[9] L. Sahu, R. Sharma, I. Sahu, M. Das, B. Sahu, and R. Kumar, "Efficient detection of Parkinson's disease using deep learning techniques over medical data," *Expert Syst.*, vol. 39, no. 3, 2022.

[10] J. Dhar, "An adaptive intelligent diagnostic system to predict early stage of parkinson's disease using two-stage dimension reduction with genetically optimized lightgbm algorithm," *Neural Comput. Appl.*, vol. 34, no. 6, pp. 4567–4593, 2022.

[11] M. Hoq, M. N. Uddin, and S.-B. Park, "Vocal feature extraction-based artificial intelligent model for Parkinson's disease detection," *Diagnostics (Basel)*, vol. 11, no. 6, p. 1076, 2021.

[12] I. M. El-Hasnony, S. I. Barakat, and R. R. Mostafa, "Optimized ANFIS model using hybrid metaheuristic algorithms for Parkinson's disease prediction in IoT environment," *IEEE Access*, vol. 8, pp. 119252–119270, 2020.