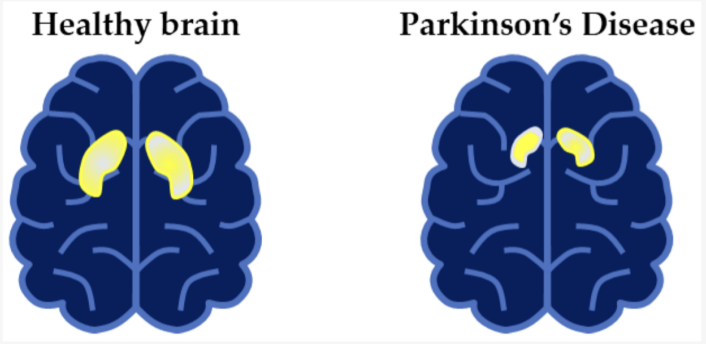
From Data to Diagnosis:Developing a Parkinson's Disease Detection Algorithm



By : Shivansh Katoch

Content

From Data to Diagnosis:Developing a Parkinson's Disease Detection Algorithm

Shivansh Katoch, Bachelor of Technology (CSE), JUIT Wakhnaghat, Solan

Abstract

The program implements a machine learning pipeline for Parkinson's disease categorization. The dataset is loaded using Pandas after importing the necessary libraries. Then, in order to learn more about the data, exploratory data analysis methods are used. These methods include viewing the dataset, verifying its dimensions, and producing descriptive statistics. Additionally, the code makes use of Matplotlib and Seaborn to show the features using correlation matrices and histograms. The 'name' column is then removed from the dataset and a check is made for any null values. To assess the relative relevance of each feature, labels and features are separated, and Fisher scores are computed. Using train_test_split from scikit-learn, the dataset is divided into training and testing sets. To guarantee that all of the features have the same scale, the features are scaled using StandardScaler.code implements a machine learning pipeline for the classification of Parkinson's disease. It begins by importing essential libraries and loading the dataset using Pandas. Exploratory data analysis techniques are then applied to gain insights into the data, including viewing the dataset, checking its dimensions, and generating descriptive statistics. The code also visualizes the features through histograms and correlation matrices using Matplotlib and Seaborn. Next, the dataset is preprocessed by removing the 'name' column and checking for any null values. The features and labels are separated, and Fisher scores are calculated to determine the importance of each feature. The dataset is split into training and testing sets using train_test_split from scikit-learn. The features are scaled using StandardScaler to ensure all features have the same scale. Three classifiers, namely K-nearest neighbors (KNN), XGBoost, and support vector machine (SVM), are trained and evaluated using accuracy and F1 score metrics from scikit-learn's metrics module.

Confusion matrices are generated and plotted using Seaborn's heatmap function to visualize the performance of each model. Finally, the code identifies the model with the highest accuracy and prints the corresponding model name. Overall, this code provides a comprehensive pipeline for Parkinson's disease classification, encompassing data loading, preprocessing, feature selection, model training, evaluation, and result visualization.

Keywords : *Parkinson's disease, machine learning pipeline, confusion matrix, model evaluation, heatmap visualization, scikit-learn, Pandas, NumPy, Matplotlib, Seaborn.*

Introduction

Parkinson's Disease (PD) is a degenerative neurological disorder characterized by a decrease in dopamine levels in the brain. This condition leads to a deterioration of movement, including the presence of tremors and stiffness. Additionally, PD often manifests in speech-related issues such as dysarthria (difficulty articulating sounds), hypophonia (lowered volume), and monotone (reduced pitch range). Furthermore, cognitive impairments, mood changes, and an increased risk of dementia can occur as the disease progresses.

The traditional diagnosis of Parkinson's Disease heavily relies on clinicians taking a neurological history of the patient and observing their motor skills in various situations. Since there is currently no definitive laboratory test available for PD, the diagnosis process can be challenging, particularly in the early stages when motor symptoms are not yet severe. Monitoring the progression of the disease over time also necessitates repeated clinic visits by the patient. Therefore, there is a pressing need for an effective screening process that can be conducted remotely without requiring a clinic visit.

Fortunately, research has shown promising results in utilizing machine learning algorithms to analyze voice recordings as a potential diagnostic tool for Parkinson's Disease. PD patients exhibit distinct vocal characteristics, including the aforementioned dysarthria, hypophonia, and monotone. By leveraging machine learning techniques and training algorithms on a dataset of voice recordings

from both PD patients and healthy individuals, it is possible to develop a model capable of accurately diagnosing PD based on voice features.

The primary advantage of using voice recordings as a screening tool is its non-invasive nature and the ability to conduct it remotely. Patients can easily record their voices using smartphones or dedicated devices, and the recordings can be securely transmitted to healthcare professionals for analysis. If machine learning algorithms demonstrate high accuracy in diagnosing PD, voice analysis could serve as an effective initial step in the diagnostic process, streamlining the identification of individuals who may require further evaluation by a clinician.

However, it is important to note that voice analysis should not replace a comprehensive clinical evaluation conducted by qualified healthcare professionals. Machine learning algorithms are valuable tools for assisting in diagnosis, but they should always be used in conjunction with other clinical assessments and medical expertise.

To advance this field and enhance the early detection and monitoring of Parkinson's Disease, further research and validation studies are necessary. Collaboration between researchers, clinicians, and data scientists can drive advancements in machine learning models for PD diagnosis using voice recordings, leading to improved accessibility and convenience in the diagnostic process.

Hence, Parkinson's Disease diagnosis can benefit from the application of machine learning algorithms to analyze voice recordings. This non-invasive approach has the potential to provide an effective screening step before an appointment with a clinician, aiding in the early detection and monitoring of PD. However, continued research and collaboration are crucial to ensure the development of robust and accurate models that complement clinical evaluations and medical expertise.

Description and Working of Algorithm

About Dataset

In this project dataset, given by Little MA, McSharry PE, Roberts SJ, Costello DAE, Moroz IM. (26 June 2007). 'Exploiting Nonlinear Recurrence and Fractal Scaling Properties for Voice Disorder Detection', BioMedical Engineering OnLine 2007 was used in this algorithm.

Attribute Information:

Matrix column entries (attributes):

name - ASCII subject name and recording number

MDVP:Fo(Hz) - Average vocal fundamental frequency

MDVP:Fhi(Hz) - Maximum vocal fundamental frequency

MDVP:Flo(Hz) - Minimum vocal fundamental frequency

MDVP:Jitter(%),MDVP:Jitter(Abs),MDVP:RAP,MDVP:PPQ,Jitter:DDP - Several measures of variation in fundamental frequency

MDVP:Shimmer,MDVP:Shimmer(dB),Shimmer:APQ3,Shimmer:APQ5,MDVP:APQ,Shimmer:DDA - Several measures of variation in amplitude

NHR,HNR - Two measures of ratio of noise to tonal components in the voice

status - Health status of the subject (one) - Parkinson's, (zero) - healthy

RPDE,D2 - Two nonlinear dynamical complexity measures

DFA - Signal fractal scaling exponent

spread1,spread2,PPE - Three nonlinear measures of fundamental frequency variation

Libraries Used :

Pandas

Pandas is a powerful open-source library for data manipulation, analysis, and cleaning in Python. It provides easy-to-use data structures and data analysis tools, making it a popular choice among data scientists, analysts, and developers working with tabular or structured data.

Some key features and functionalities of the Pandas library include:

1. Data-frame: The central data structure in pandas is the Data-frame, which represents tabular data with rows and columns. It provides various methods for indexing, selecting, filtering, transforming, and aggregating data.

2. Data manipulation: Pandas offers a wide range of data manipulation capabilities, such as merging and joining datasets, reshaping data, handling missing values, handling duplicates, and creating derived columns based on existing data.

3. Data input and output: Pandas supports reading data from and writing data to various file formats, including CSV, Excel, SQL databases, and more. It simplifies the process of loading data into memory and enables seamless integration with other data processing or analysis libraries.

4. Data exploration and analysis: With Pandas, you can perform exploratory data analysis tasks, such as descriptive statistics, data visualization, time series analysis, grouping and aggregating data, and applying mathematical and statistical operations to data.

5. Integration with other libraries: Pandas integrates well with other popular libraries in the Python ecosystem, such as NumPy, Matplotlib, Scikit-learn, and more. It provides interoperability and compatibility with these libraries, allowing you to leverage their functionalities in conjunction with pandas.

Numpy

NumPy (Numerical Python) is a fundamental open-source library for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy forms the foundation for many other scientific computing libraries in Python.

Here are some key features and functionalities of the NumPy library:

1. Multi-dimensional arrays: NumPy's main feature is the `ndarray` (n-dimensional array) object. It allows efficient storage and manipulation of homogeneous data, such as numbers, in a grid-like structure. Arrays can have one or more dimensions and support various data types.

2. Vectorized operations: NumPy provides a wide range of mathematical functions that can be applied directly to arrays, known as vectorized operations. These operations are optimized for performance and enable efficient element-wise computations, array broadcasting, and mathematical operations like addition, subtraction, multiplication, etc., without the need for explicit loops.

3. Array indexing and slicing: NumPy offers powerful indexing and slicing capabilities to access and modify elements within arrays. You can use indexing to retrieve specific elements or subsets of an array based on their position or certain conditions.

4. Broadcasting: Broadcasting is a powerful feature in NumPy that enables arithmetic operations between arrays of different shapes and sizes. NumPy automatically aligns the shapes of arrays, if possible, to perform element-wise operations, making it convenient for performing calculations on arrays with different dimensions.

5. Linear algebra operations: NumPy includes a comprehensive suite of linear algebra functions, such as matrix multiplication, matrix decomposition, solving linear equations, eigenvalues and eigenvectors, and more. These operations are crucial for many scientific and mathematical applications.

6. Integration with other libraries: NumPy seamlessly integrates with other scientific computing libraries in Python, such as SciPy, pandas, Matplotlib, and scikit-learn. It serves as the foundational library for these packages, providing efficient data structures and numerical computing capabilities.

Matplotlib

Matplotlib is a widely used open-source library in Python for creating static, animated, and interactive visualizations. It provides a comprehensive set of tools for generating a variety of plots, charts, and graphs, making it a valuable tool for data visualization and exploration.

Here are some key features and functionalities of the Matplotlib library:

1. Plotting Functions: Matplotlib provides a range of plotting functions to create different types of visualizations, such as line plots, scatter plots, bar plots, histograms, pie charts, 3D plots, and more. These functions allow you to customize the appearance of plots, including colors, markers, line styles, labels, and legends.

2. Object-Oriented Interface: Matplotlib offers an object-oriented API that allows you to have more control over your plots. You can create Figure and Axes objects and manipulate them individually to customize various aspects of your plot. This interface provides flexibility and enables the creation of complex layouts and multiple subplots.

3. Integration with NumPy and Pandas: Matplotlib seamlessly integrates with other scientific computing libraries, such as NumPy and pandas. You can directly plot NumPy arrays or pandas DataFrames, making it easy to visualize data stored in these structures. Matplotlib's compatibility with these libraries enhances its functionality and usability.

4. Customization and Styling: Matplotlib provides extensive options for customizing and styling plots. You can modify the appearance of axes, grids, titles, tick labels, legends, and more. Matplotlib also supports the use of custom color maps and styles to create visually appealing and consistent plots.

5. Saving and Exporting Plots: Matplotlib allows you to save plots in various image formats, including PNG, JPEG, PDF, SVG, and more. You can save plots for use in reports, presentations, or further processing. Additionally, Matplotlib provides interactive features for saving plots as interactive HTML files or embedding plots in graphical user interfaces (GUIs).

Seaborn

Seaborn is a Python data visualization library built on top of Matplotlib. It provides a high-level interface for creating informative and visually appealing statistical graphics. Seaborn is particularly useful for visualizing and exploring patterns in complex datasets and is widely used in data analysis, statistical modeling, and machine learning projects.

Here are some key features and functionalities of the Seaborn library:

1. Enhanced Visualizations: Seaborn offers a variety of statistical visualizations that are not readily available in Matplotlib. It provides built-in support for statistical estimation and data aggregation, making it easier to create informative and visually appealing plots. Seaborn also includes additional plot types, such as violin plots, box plots, swarm plots, and more.

2. Easy Styling and Customization: Seaborn comes with built-in themes and color palettes that can instantly enhance the visual aesthetics of your plots. It provides several pre-defined themes, such as darkgrid, whitegrid, dark, white, and ticks, which you can easily apply to your plots.

Seaborn also allows fine-grained control over plot elements, enabling customization of colors, fonts, grid lines, and more.

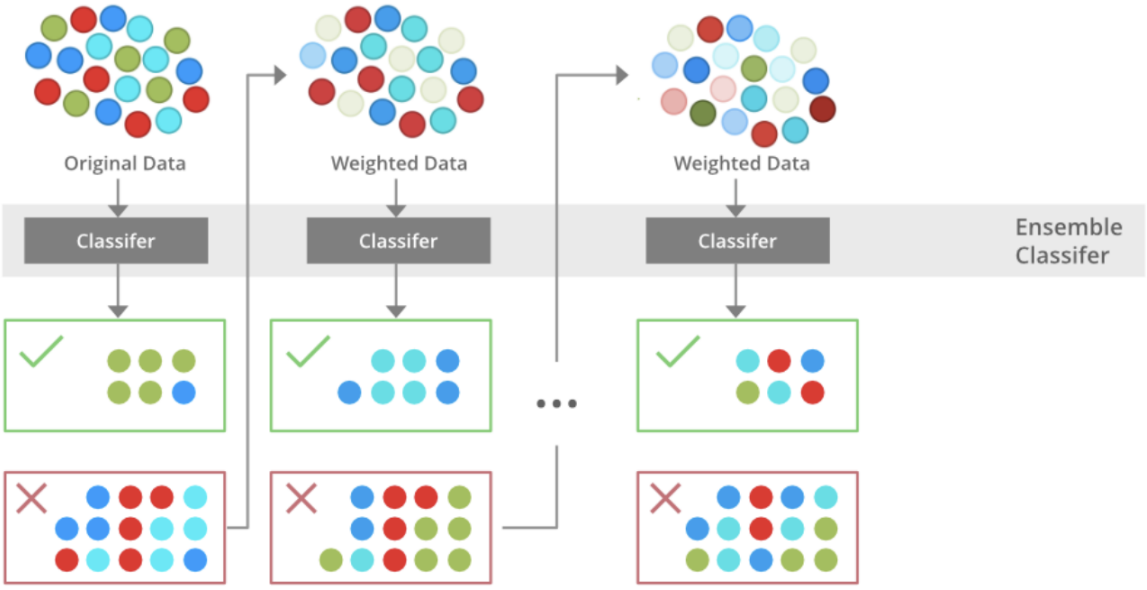
3. Statistical Estimation: Seaborn simplifies the visualization of statistical estimation by providing functions that combine data visualization and statistical analysis. For example, you can create a plot to display a point estimate and confidence interval using Seaborn's `pointplot()` or `barplot()` functions.

4. Categorical Data Visualization: Seaborn excels in visualizing categorical data. It provides convenient functions to create various plots for categorical variables, such as bar plots, count plots, box plots, and violin plots. These plots can provide insights into the distribution of data across different categories.

5. Integration with Pandas: Seaborn seamlessly integrates with the pandas library, making it easy to visualize data stored in pandas DataFrames. You can directly pass pandas DataFrame objects to Seaborn plotting functions, enabling you to quickly explore and visualize data.

6. Pairplot and Heatmap: Seaborn's `pairplot()` function allows you to create a matrix of scatter plots or pairwise plots, showing the relationships between multiple variables in a dataset. Additionally, Seaborn provides the `heatmap()` function, which enables the creation of color-coded heatmaps to visualize correlations or patterns in large datasets.

Xgboost and XGBClassifier



XGBoost (Extreme Gradient Boosting) is a popular machine learning algorithm known for its high performance and efficiency in handling structured data. It is an optimized implementation of the gradient boosting framework that leverages parallel processing and tree-based models to achieve accurate predictions and competitive performance in various machine learning tasks.

XGBoost provides both regression and classification algorithms, and the XGBClassifier is the specific implementation of XGBoost for classification problems. Here's an overview of XGBoost and the XGBClassifier:

1. Gradient Boosting: XGBoost is based on the gradient boosting algorithm, which combines multiple weak predictive models (typically decision trees) into an ensemble model. It builds the ensemble iteratively, minimizing a loss function by adding new models that correct the mistakes made by previous models.

2. Tree-based Models: XGBoost primarily uses tree-based models, such as decision trees, as base learners. Decision trees are constructed sequentially, where each subsequent tree corrects the errors of the previous ones. The final prediction is made by summing the predictions of all the trees.

3. Regularization Techniques: XGBoost employs regularization techniques to prevent overfitting and improve generalization. It includes L1 and L2 regularization terms to control the complexity of individual trees and the overall model. Regularization helps in reducing variance and improving the model's performance on unseen data.

4. Feature Importance: XGBoost provides a feature importance metric that helps identify the most influential features in the dataset. This information can be useful for feature selection, feature engineering, and gaining insights into the data.

5. Parallel Processing: XGBoost is designed to leverage parallel processing capabilities, making it highly scalable and efficient. It can take advantage of multiple CPU cores during training, resulting in faster model building and prediction times.

6. Cross-Validation: XGBoost supports cross-validation techniques to evaluate model performance and tune hyperparameters. Cross-validation helps in assessing the model's generalization ability and selecting optimal hyperparameters for improved performance.

The XGBClassifier is a specific implementation of XGBoost for classification tasks. It extends the XGBoost algorithm to handle classification problems by using an appropriate loss function (such as logistic loss) and incorporating class labels in the model training process. **(appendix 1)**

Sklearn

Scikit-learn, often referred to as Sklearn, is a popular machine learning library in Python. It provides a wide range of tools and functionalities for various machine learning tasks, including classification, regression, clustering, dimensionality reduction, model selection, and preprocessing. Scikit-learn is built on top of NumPy, SciPy, and Matplotlib and is designed to be user-friendly, efficient, and accessible for both beginners and experienced practitioners.

Here are some key features and functionalities of scikit-learn:

1. Consistent API: Scikit-learn offers a consistent API for all its machine learning models. This uniform interface makes it easy to switch between different algorithms and models. The library follows a fit-predict paradigm, where you fit the model to the training data using the `fit()` method and make predictions on new data using the `predict()` method.

2. Wide Range of Algorithms: Scikit-learn provides implementations of a broad range of machine learning algorithms, including linear models, decision trees, support vector machines, ensemble methods (e.g., random forests, gradient boosting), naive Bayes, nearest neighbors, clustering algorithms, and more. These algorithms cover both supervised and unsupervised learning techniques.

3. Data Preprocessing and Feature Engineering: Scikit-learn includes a comprehensive set of tools for data preprocessing and feature engineering. It provides functions for handling missing values, feature scaling, encoding categorical variables, feature selection, dimensionality

reduction, and more. These preprocessing steps are crucial for preparing the data before training machine learning models.

4. Model Evaluation and Selection: Scikit-learn offers various evaluation metrics and techniques for assessing the performance of machine learning models. It provides functions for calculating accuracy, precision, recall, F1-score, ROC curves, and more. The library also supports cross-validation for estimating model performance and hyperparameter tuning to optimize model parameters.

5. Pipelines: Scikit-learn allows you to build complex machine learning workflows using pipelines. Pipelines enable you to chain together multiple preprocessing steps and models into a single object. This simplifies the process of building and deploying machine learning pipelines, making it easier to maintain and reproduce experiments.

6. Integration with Other Libraries: Scikit-learn seamlessly integrates with other popular Python libraries, such as NumPy, pandas, and Matplotlib. It can directly accept NumPy arrays, pandas DataFrames, and Series as input data, making it convenient for working with data stored in these structures. Scikit-learn also provides integration with tools like Jupyter Notebook for interactive and exploratory data analysis.

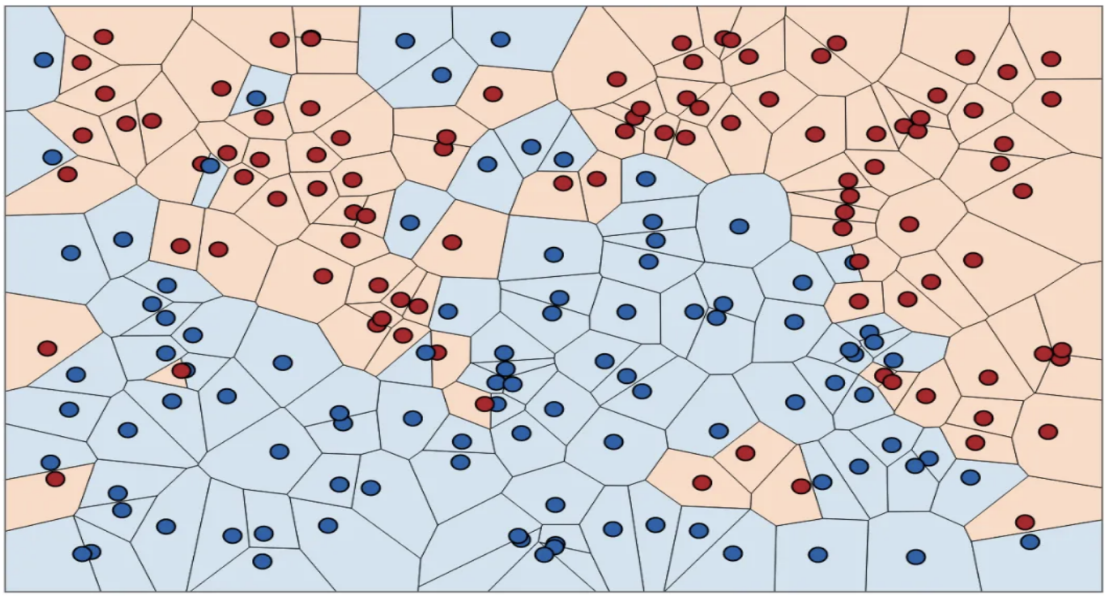
KNeighborsClassifier (Sklearn.neighbors)

`Sklearn.neighbors` is a module in scikit-learn that provides functionality for k-nearest neighbors (KNN) algorithms. K-nearest neighbors is a simple yet powerful non-parametric classification and regression algorithm. It works by finding the k nearest data points in the training set to a given test point and makes predictions based on the majority vote (for classification) or the average (for regression) of the labels/targets of those nearest neighbors.

The `KNeighborsClassifier` is a specific implementation of the K-nearest neighbors algorithm for classification tasks. It is a part of the `sklearn.neighbors` module. Here's an overview of `sklearn.neighbors` and `KNeighborsClassifier`:

1. K-Nearest Neighbors (KNN): KNN is a simple algorithm that assigns labels to new data points based on the majority class of their k nearest neighbors. It can be used for both classification and regression tasks. KNN assumes that similar instances tend to have similar labels/targets.

2. Distance Metrics: KNN uses distance metrics to measure the similarity between instances. The most common distance metric used is Euclidean distance, but other metrics like Manhattan distance, Minkowski distance, and Hamming distance are also available in scikit-learn.

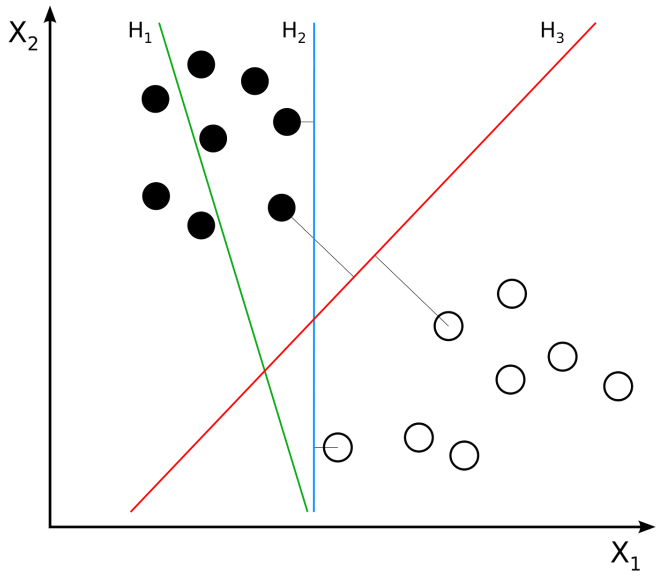


3. Hyperparameters: The key hyperparameter in KNN is the value of k, which determines the number of neighbors considered for prediction. Additionally, you can choose the weight function used in the majority vote or averaging process, as well as the distance metric. Scikit-learn provides flexibility to set these hyperparameters based on the problem at hand.

4. Model Training: To train a `KNeighborsClassifier`, you need a labeled dataset. The `fit()` method is used to train the model by providing the feature matrix (`X`) and the corresponding labels (`y`) as inputs.

5. Predictions: After the model is trained, you can make predictions on new, unseen data points using the `predict()` method. The KNN algorithm finds the k nearest neighbors to the new data point and assigns the majority label among those neighbors as the predicted label.
(**appendix 2**)

SVM(Support Vector Machines)



SVM (Support Vector Machines) is a powerful supervised machine learning algorithm used for both classification and regression tasks. It is widely used due to its ability to handle complex datasets and produce accurate results. SVMs are particularly effective in cases where the data is not linearly separable and requires complex decision boundaries.

Here's an overview of SVM and its key concepts:

1. Hyperplane: In SVM, the algorithm aims to find an optimal hyperplane that best separates the data points into different classes. In binary classification, a hyperplane is a linear decision boundary that separates the data into two classes. In multi-class classification, multiple hyperplanes are used to separate different classes.

2. Support Vectors: Support vectors are the data points that lie closest to the decision boundary. These points are crucial for defining the hyperplane and determining the decision boundary. Only the support vectors influence the construction of the hyperplane, while the other data points are disregarded.

3. Margin: The margin is the region between the support vectors of different classes. SVM aims to maximize the margin, i.e., the distance between the support vectors and the decision boundary. Maximizing the margin helps achieve better generalization and improves the algorithm's ability to classify new, unseen data accurately.

4. Kernel Trick: SVM can handle non-linearly separable data by using a kernel function to transform the data into a higher-dimensional feature space. The kernel function maps the original data into a higher-dimensional space where it might become linearly separable. Common kernel functions include linear, polynomial, radial basis function (RBF), and sigmoid.

5. C Parameter: SVM has a regularization parameter called C, which controls the trade-off between maximizing the margin and minimizing the classification errors. A smaller value of C

allows for a wider margin but may lead to misclassification, while a larger value of C reduces the margin but aims to classify all training examples correctly.

6. SVM for Regression: SVM can also be used for regression tasks, where the goal is to predict continuous values instead of discrete classes. The SVM regression algorithm tries to fit as many data points as possible within a certain margin.**(appendix 2)**

Train_test_split (from sklearn.model_selection)

`Train_test_split` is a function provided by the `sklearn.model_selection` module in scikit-learn. It is commonly used to split a dataset into separate training and testing sets for machine learning tasks. The function randomly shuffles and partitions the data into two or more subsets, allowing for evaluation and validation of machine learning models.

Here's an overview of the `train_test_split` function and its key parameters:

train_test_split(*arrays, **options)

Parameters:
- **arrays**: One or more arrays or matrices containing the input features and target values.
- **test_size**: Float, int, or None (default is None). Specifies the size or proportion of the test set. If float, it represents the fraction of the data to allocate for testing (e.g., 0.2 for 20%). If int, it represents the absolute number of samples to allocate for testing.
- `train_size`: Float, int, or None (default is None). Specifies the size or proportion of the training set. If float, it represents the fraction of the data to allocate for training. If int, it represents the absolute number of samples to allocate for training. Note that if `test_size` is not specified, the remaining data after allocating for training will automatically be used for testing.
- `random_state`: int, RandomState instance, or None (default is None). Controls the random shuffling and splitting of the data. Setting a specific `random_state` ensures reproducibility.
- `shuffle`: bool (default is True). Determines whether to shuffle the data before splitting. If set to False, the data will be split in a deterministic manner without shuffling.
- `stratify`: array-like or None (default is None). If specified, it provides the labels or target variable for stratified sampling. This ensures that the proportion of classes in the training and testing sets is similar to the input data.

Returns:
- Tuple of split data: Returns the split data in the form `(X_train, X_test, y_train, y_test)` or `(X_train, X_test)` if no target variable is provided. The arrays `X_train` and `X_test` contain the feature data, while `y_train` and `y_test` contain the corresponding target values.

Metrics from Sklearn

The `Sklearn.metrics` module in scikit-learn provides a wide range of evaluation metrics for assessing the performance of machine learning models. These metrics help measure the accuracy, precision, recall, F1-score, and other aspects of model predictions. They are commonly used to evaluate classification, regression, and clustering models. Here are some commonly used evaluation metrics available in `sklearn.metrics`:

1. Classification Metrics:
   - Accuracy Score (`accuracy_score`): Computes the accuracy of the classification model by comparing predicted labels with true labels.
   - Precision (`precision_score`): Measures the ability of the model to correctly predict positive instances among all instances predicted as positive.
   - Recall (`recall_score`): Calculates the ability of the model to correctly predict positive instances among all actual positive instances.
   - F1-Score (`f1_score`): Harmonic means of precision and recall, providing a single metric that balances both measures.
   - Classification Report (`classification_report`): Provides a comprehensive report with precision, recall, F1-score, and support for each class in the classification task.
   - Confusion Matrix (`confusion_matrix`): Tabulates the actual and predicted labels, helping analyze the types of errors made by the model.

2. Regression Metrics:
   - Mean Squared Error (`mean_squared_error`): Calculates the average of squared differences between predicted and true values.
   - Mean Absolute Error (`mean_absolute_error`): Computes the average of absolute differences between predicted and true values.
   - R^2 Score (`r2_score`): Measures the proportion of the variance in the target variable explained by the model.
   - Explained Variance Score (`explained_variance_score`): Estimates the explained variance in the target variable by the model.

3. Clustering Metrics:
   - Adjusted Rand Index (`adjusted_rand_score`): Measures the similarity between two clustering assignments, considering all pairs of samples and cluster labels.
   - Silhouette Coefficient (`silhouette_score`): Quantifies the cohesion and separation of clusters based on the mean intra-cluster distance and the mean nearest-cluster distance.

These are just a few examples of the evaluation metrics available in `sklearn.metrics`. There are many more metrics for specific tasks and requirements. To use these metrics, you typically import them from `sklearn.metrics` and call the corresponding functions, passing the predicted and true values as arguments.

Accuracy_score (from sklearn.metrics)

`Accuracy_score` is a function provided by the `sklearn.metrics` module in scikit-learn. It is used to compute the accuracy of a classification model by comparing the predicted labels with the true labels. The accuracy score is a commonly used metric for evaluating classification models.

Here's an overview of the `accuracy_score` function and its parameters:

accuracy_score(y_true, y_pred, normalize=True, sample_weight=None)

Parameters:
- `y_true`: The true target values (ground truth) of the classification.
- `y_pred`: The predicted target values generated by a classifier.
- `normalize`: A boolean value indicating whether to normalize the accuracy score or not. If set to `True` (default), the score is returned as a fraction of correctly classified samples. If set to `False`, the number of correctly classified samples is returned.
- `sample_weight`: An optional array of weights assigned to individual samples. It can be used to give more importance to certain samples during the accuracy calculation.

Returns:
- The accuracy score, either as a fraction (default) or as a count of correctly classified samples, depending on the `normalize` parameter.**(appendix 4)**

StandardScaler (Preprocessing from Sklearn)

`StandardScaler` is a class provided by the `sklearn.preprocessing` module in scikit-learn. It is used for standardizing or scaling features by removing the mean and scaling to unit variance. Standardization is a common preprocessing step in machine learning that helps to bring features onto a similar scale, which can be beneficial for certain algorithms.

Here's an overview of the `StandardScaler` class and its usage:

```
from sklearn.preprocessing import StandardScaler

# Create an instance of StandardScaler
scaler = StandardScaler()

# Fit the scaler to the training data
scaler.fit(X_train)

# Transform the training and testing data
```

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

Usage:
1. Import the `StandardScaler` class from `sklearn.preprocessing`.

2. Create an instance of `StandardScaler`.

3. Fit the scaler to the training data using the `fit` method. This step computes the mean and standard deviation of the training data.

4. Transform the training and testing data using the `transform` method. This step applies the scaling based on the computed mean and standard deviation.

The `StandardScaler` scales each feature independently, so that each feature has a mean of 0 and a standard deviation of 1. This transformation ensures that the features are centered around 0 and have a similar scale, which can be important for algorithms that are sensitive to the scale of the features (e.g., gradient descent-based algorithms).

Confusion matrix (from sklearn.metrics)

`Confusion_matrix` is a function provided by the `sklearn.metrics` module in scikit-learn. It is used to compute a confusion matrix, which is a table that summarizes the performance of a classification model by comparing the actual and predicted labels. The confusion matrix is a helpful tool for understanding the types of errors made by a model.

Here's an overview of the `confusion_matrix` function and its parameters:

```
confusion_matrix(y_true, y_pred, labels=None, sample_weight=None, normalize=None)
```

Parameters:
- `y_true`: The true target values (ground truth) of the classification.

- `y_pred`: The predicted target values generated by a classifier.

- `labels`: An optional list of labels indicating the order of the classes in the confusion matrix. If not provided, the labels will be inferred from the input data.

- `sample_weight`: An optional array of weights assigned to individual samples. It can be used to give more importance to certain samples during the calculation of the confusion matrix.

- `normalize`: An optional string value specifying the normalization mode. If set to `'true'`, the confusion matrix will be normalized by the true class counts. If set to `'pred'`, it will be normalized by the predicted class counts. If set to `'all'`, it will be normalized by the total number of samples. If set to `None` (default), no normalization will be applied.

Returns:

- The confusion matrix, represented as a 2D array. The rows correspond to the true labels, and the columns correspond to the predicted labels.**(appendix 6)**

F1_score (from Sklearn.metrics)

$$F_1 = \frac{2}{\frac{1}{\text{recall}} \times \frac{1}{\text{precision}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$
$$= \frac{\text{tp}}{\text{tp} + \frac{1}{2}(\text{fp} + \text{fn})}$$

F1_score is a function provided by the `sklearn.metrics` module in scikit-learn. It is used to compute the F1 score, which is a metric that combines precision and recall into a single value. The F1 score is commonly used in binary classification tasks, but it can also be extended to multi-class classification problems.

Here's an overview of the `f1_score` function and its parameters:

f1_score(y_true, y_pred, labels=None, pos_label=1, average='binary', sample_weight=None, zero_division='warn')

Parameters:
- `y_true`: The true target values (ground truth) of the classification.

- `y_pred`: The predicted target values generated by a classifier.

- `labels`: An optional list of labels indicating the order of the classes. If not provided, the labels will be inferred from the input data.

- `pos_label`: The positive class label. This parameter is only relevant in binary classification or when the `average` parameter is set to `'binary'`.

- `average`: A string value specifying the type of averaging to be performed. Possible values are `'binary'` (default), `'micro'`, `'macro'`, `'weighted'`, and `'samples'`. If set to `'binary'`, the F1 score is computed for each class individually and then averaged. If set to `'micro'`, `'macro'`, `'weighted'`, or `'samples'`, the F1 score is computed globally by considering all classes together with different weightings.

- `sample_weight`: An optional array of weights assigned to individual samples. It can be used to give more importance to certain samples during the calculation of the F1 score.

- `zero_division`: Specifies the behavior when a division by zero occurs. By default, it is set to `'warn'`, which raises a warning. It can also be set to `'warn'` or `'raise'` to raise a warning or an exception, respectively.

Returns:
- The F1 score as a float value.

(appendix 5)

Fisher_score (from skfeature.function.similarity_based (Feature selection))

However, scikit-learn does offer various feature selection techniques through its `sklearn.feature_selection` module. Some commonly used methods include chi-squared test, mutual information, variance thresholding, and recursive feature elimination. These methods are implemented within scikit-learn and can be accessed directly.

If you are looking specifically for the Fisher score as a feature selection method, you might need to explore external libraries or custom implementations. One such library is PyFeast (https://github.com/jundongl/pyfeast), which provides a range of feature selection algorithms, including Fisher score-based methods.

If you specifically need the Fisher score implementation from the `skfeature` library, you would need to install and import that library separately.
(appendix 7)

Correlation Matrix

A correlation matrix is a table that displays the correlation coefficients between multiple variables in a dataset. It provides a way to examine the relationships between variables and assess their strength and direction.

Each cell in the correlation matrix represents the correlation coefficient between two variables. The correlation coefficient measures the strength and direction of the linear relationship between two variables. It can range from -1 to +1, where -1 indicates a perfect negative correlation, +1 indicates a perfect positive correlation, and 0 indicates no correlation.

The correlation matrix is often visualized as a heatmap, where the color intensity represents the magnitude of the correlation coefficient. This allows for easy identification of strong positive or negative correlations between variables.

The correlation matrix is useful for several purposes:

1. Understanding relationships: It helps in identifying which variables are positively or negatively correlated, indicating how they move together or in opposite directions.

2. Feature selection: It can assist in selecting variables for further analysis by identifying highly correlated variables that may provide redundant information.

3. Multicollinearity detection: It helps identify variables that are highly correlated with each other, which can be problematic in regression analysis as it may lead to instability or inaccurate coefficient estimates.

4. Data preprocessing: It can guide data preprocessing decisions, such as handling missing values or outliers based on their correlations with other variables.

In summary, a correlation matrix provides valuable insights into the relationships between variables in a dataset and serves as a foundation for further analysis and decision-making in data analysis and modeling tasks.
(appendix 8)

Full - Program :

```
#importing important library's

import pandas as pd                                                    # import pandas
import numpy as np                                                     # import numpy
import matplotlib.pyplot as plt                                        # plotting imports
import seaborn as sns                                                  # plotting import
from xgboost import XGBClassifier                                      # importing XGBClassifier
from sklearn.neighbors import KNeighborsClassifier                     # importKNeighborsClassifier
from sklearn import svm                                                # support vector model Classifier
from sklearn.model_selection import train_test_split                   # import train_test_split for splitting
Data
from sklearn import metrics                                            # model metrics
from sklearn.metrics import accuracy_score                             # import for checking accuracy_score
from sklearn.preprocessing import StandardScaler                       # importing
StandardScaler(Perprossing)
from sklearn.metrics import confusion_matrix                           # confusion matrix
from sklearn.metrics import f1_score                                   # importing f1_score
from skfeature.function.similarity_based import fisher_score           # Fisher scores (Feature selection)
%matplotlib inline                    # allows matplotlib plots to be displayed directly in the notebook
interface

# import dataset
data = pd.read_csv("/Users/shivanshkatoch/Downloads/Parkinson disease.csv")

# view the dataset data
Data

# dataset dimensions
Data.shape

# generate descriptive statistics
data.describe()

# inspect the values in the dataframe
data.info()

# delete the name column
data.drop(columns="name", axis=1, inplace=True)

# check for null values
data.isnull().sum()

# FEATURES -  selecting data except for name and status
X = data.loc[:,data.columns!='status'].values[:,1:]
x = data.loc[:,data.columns!='status']
# LABELS -  selecting only status
Y = data.loc[:,'status'].values
y = data.loc[:,'status']
```

```
x.describe() #it does not contain status of patient

y.describe() #it only contain status of patient

# analyzing the features
x.hist(figsize=(25,20))
plt.show()



# figure size
plt.figure(figsize=(30, 15))

# correlation matrix
dataplot = sns.heatmap(data.corr(), annot=True, fmt='.2f')
#The code data.corr() is used to calculate the correlation between columns in a pandas DataFrame
named data.
# It returns a new DataFrame containing the correlation values.


'''Fisher score is one of the most widely used supervised feature selection methods.'''
# Calculating Fisher scores
score = fisher_score.fisher_score(x_train, y_train)


# Plotting the ranks
feat_importances = pd.Series (score, data.columns [0:len (data.columns) -2])
feat_importances.plot (kind='barh', color = 'teal')
plt.show()


# split data
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.1)


'''here x data will be split in train and test data x_train represent training data and is corresponding
label is store in y_train same for x_test
here we use 20% data as testing data and 80% as training data '''

print(x.shape, x_train.shape, x_test.shape)


'''It applies the scaling transformation to the features of x_train (or x_test) based
on the scaling parameters learned from the training data.'''



scaler= StandardScaler()
scaler.fit(x_train)
```

```
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)

#calculating value of K
import math
n_value=int(math.sqrt(len(y_test)))
print(n_value)


# training the KNN model (k-nearest neighbors Classifier)
knnmodel = KNeighborsClassifier(n_neighbors=4)
knnmodel.fit(x_train, y_train)
knnpredict = knnmodel.predict(x_test)


# training the XGB model (Extreme Gradient Boosting)
XGBmodel = XGBClassifier()
XGBmodel.fit(x_train, y_train)
XGBpredict = XGBmodel.predict(x_test)

# training the SVM (suppport vector model)
svmmodel = svm.SVC(kernel='linear')
svmmodel.fit(x_train, y_train) # training the SVM model with training data
svmpredict = svmmodel.predict(x_test)


print(f'The Mean Absolute Error of XGBClassifier() is:{metrics.mean_absolute_error(y_test, XGBpredict):
.2f}')
print(f'The root Mean Squared Error of XGBClassifier() is:{np.sqrt(metrics.mean_squared_error(y_test,
XGBpredict)): .2f}')



print(f'The Mean Absolute Error of SVM model is:{metrics.mean_absolute_error(y_test, svmpredict): .2f}')
print(f'The root Mean Squared Error of SVM model is:{np.sqrt(metrics.mean_squared_error(y_test,
svmpredict)): .2f}')



print(f'The Mean Absolute Error of KNN model is:{metrics.mean_absolute_error(y_test, knnpredict): .2f}')
print(f'The root Mean Squared Error of KNN model is:{np.sqrt(metrics.mean_squared_error(y_test,
knnpredict)): .2f}')



knn_accuracy_score = accuracy_score(y_test, knnpredict)*100
knn_f1_score = f1_score(y_test,knnpredict)
svm_accuracy_score = accuracy_score(y_test, svmpredict)*100
svm_f1_score = f1_score(y_test,svmpredict)
```

```
XGB_accuracy_score = accuracy_score(y_test, XGBpredict)*100
XGB_f1_score = f1_score(y_test,XGBpredict)


print("f1_score for knn model is:",knn_f1_score)
print("f1_score for XGB model is:",XGB_f1_score)
print("f1_score for svm model is:",svm_f1_score)
print("accuracy_score for knn model is:",knn_accuracy_score)
print("accuracy_score for XGBClassifier model is:",XGB_accuracy_score)
print("accuracy_score for svm model is:",svm_accuracy_score)



cm=confusion_matrix(y_test ,knnpredict)
plt.figure(figsize=(8,6))

fg=sns.heatmap(cm,annot=True)
figure=fg.get_figure()

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Output Confusion Matrix For KNNClassifier")


cm=confusion_matrix(y_test ,XGBpredict)
plt.figure(figsize=(8,6))

fg=sns.heatmap(cm,annot=True)
figure=fg.get_figure()

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Output Confusion Matrix For XGBClassifier")


cm=confusion_matrix(y_test ,svmpredict)
plt.figure(figsize=(8,6))

fg=sns.heatmap(cm,annot=True)
figure=fg.get_figure()

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Output - Confusion Matrix for SVMClassifier")

pd.DataFrame({'Actual': y_test, 'XGBPredict': XGBpredict, 'svmPredict': svmpredict,'knnPredict':
knnpredict})

#Finding best model for test size 0.1 and given dataset
```

```
highest = max(knn_accuracy_score, XGB_accuracy_score, svm_accuracy_score)
print("The highest accuracy_score is:", highest)
def find_model(a, b, c, d):
    if d == a: print("Using k-nearest neighbors Classifier model")
    if d == b: print("Using Extreme Gradient Boosting Classifier model")
    if d == c: print("Using support vector model Classifier")
find_model(knn_accuracy_score, XGB_accuracy_score, svm_accuracy_score, highest)
```

Line by-line explained Program :

Here's an explanation of each line:

```
#importing important library's

import pandas as pd                                                 # import pandas
import numpy as np                                                  # import numpy
import matplotlib.pyplot as plt                                     # plotting imports
import seaborn as sns                                               # plotting import
from xgboost import XGBClassifier                                   # importing XGBClassifier
from sklearn.neighbors import KNeighborsClassifier                  # import KNeighborsClassifier
from sklearn import svm                                             # support vector model Classifier
from sklearn.model_selection import train_test_split                # import train_test_split for splitting Data
from sklearn import metrics                                         # model metrics
from sklearn.metrics import accuracy_score                          # import for checking accuracy_score
from sklearn.preprocessing import StandardScaler                    # importing StandardScaler(Perprossing)
from sklearn.metrics import confusion_matrix                        # confusion matrix
from sklearn.metrics import f1_score                                # importing f1_score
from skfeature.function.similarity_based import fisher_score        # Fisher scores (Feature selection)
```

Line 1. import pandas as pd

This line imports the pandas library and assigns it the alias `pd`. Pandas is a popular library for data manipulation and analysis in Python.

Line 2. import numpy as np

This line imports the numpy library and assigns it the alias `np`. NumPy is a fundamental library for numerical computing in Python, providing support for arrays, matrices, and mathematical operations.

Line 3. import matplotlib.pyplot as plt

This line imports the pyplot module from the matplotlib library and assigns it the alias `plt`. Matplotlib is a widely used plotting library in Python, and pyplot is its sub-module that provides a MATLAB-like interface for creating visualizations.

Line 4. import seaborn as sns

This line imports the seaborn library, which is a data visualization library built on top of matplotlib. Seaborn provides a high-level interface for creating informative and aesthetically pleasing statistical graphics.

Line 5. from xgboost import XGBClassifier

This line imports the XGBClassifier class from the xgboost library. XGBoost is a gradient boosting framework that is widely used for machine learning tasks, especially in structured data problems.

Line 6. from sklearn.neighbors import KNeighborsClassifier

This line imports the KNeighborsClassifier class from the sklearn.neighbors module. The K-nearest neighbors algorithm is a simple yet powerful classification algorithm that assigns a label to a data point based on the labels of its nearest neighbors in the feature space.

Line 7. from sklearn import svm

This line imports the svm module from the sklearn library. It provides support for Support Vector Machine (SVM) algorithms, which are popular machine learning models for classification and regression tasks.

Line 8. from sklearn.model_selection import train_test_split

This line imports the train_test_split function from the sklearn.model_selection module. It is used to split a dataset into training and testing sets, which is a common practice in machine learning to evaluate the performance of models.

Line 9. from sklearn import metrics

This line imports the metrics module from the sklearn library. It provides various evaluation metrics and functions for assessing the performance of machine learning models.

Line 10.from sklearn.metrics import accuracy_score

This line imports the accuracy_score function from the sklearn.metrics module. It is used to calculate the accuracy of a classification model by comparing the predicted labels with the true labels.

Line 11. from sklearn.preprocessing import StandardScaler

This line imports the StandardScaler class from the sklearn.preprocessing module. StandardScaler is used for standardizing or scaling features by removing the mean and scaling to unit variance, as discussed earlier.

Line 12. from sklearn.metrics import confusion_matrix

This line imports the confusion_matrix function from the sklearn.metrics module. It is used to compute the confusion matrix, which summarizes the performance of a classification model by comparing the predicted labels with the true labels.

Line 13. from sklearn.metrics import f1_score

This line imports the f1_score function from the sklearn.metrics module. It is used to compute the F1 score, which is a metric that combines precision and recall into a single value, as discussed earlier.

Line 14. from skfeature.function.similarity_based import fisher_score

This line imports the fisher_score function from the skfeature.function.similarity_based module. This is not a part of scikit-learn but appears to be a separate library or module for feature selection. The fisher_score function likely implements a feature selection method based on Fisher score, which is a criterion for ranking features in classification tasks.

Line 15. %matplotlib inline # allows matplotlib plots to be displayed directly in the notebook interface

The `%matplotlib inline` is a special command used in Jupyter Notebook or JupyterLab environments to enable the inline plotting of matplotlib figures. It allows the plots generated by matplotlib to be displayed directly in the notebook interface without the need for extra commands or opening separate plot windows.

When `%matplotlib inline` is executed in a notebook cell, it sets the backend of matplotlib to render the plots as static images embedded in the notebook. This is particularly useful for generating and visualizing plots in an interactive and convenient manner within the notebook environment.

```
%matplotlib inline                    # allows matplotlib plots to be displayed directly in the notebook interface

# import dataset
data = pd.read_csv("/Users/shivanshkatoch/Downloads/Parkinsson disease.csv")
```

Line 16. data = pd.read_csv("/Users/shivanshkatoch/Downloads/Parkinsson disease.csv")

The code is attempting to read a CSV file using the pandas library and assign the data to a variable named `data`. The file path used in the code (`/Users/shivanshkatoch/Downloads/Parkinsson disease.csv`) assumes that the CSV file is located at that specific file path on your local machine.

However, it's important to note that as an AI language model, I don't have access to your local file system, so I cannot directly read files from your machine. You would need to provide the correct file path or adjust the code according to the location of your CSV file on your machine.

Line 17. data

```
In [374]: # view the dataset data
          data
```

Out[374]:

	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)	...
0	119.992	157.302	74.997	0.00784	0.00007	0.00370	0.00554	0.01109	0.04374	0.426	...
1	122.400	148.650	113.819	0.00968	0.00008	0.00465	0.00696	0.01394	0.06134	0.626	...
2	116.682	131.111	111.555	0.01050	0.00009	0.00544	0.00781	0.01633	0.05233	0.482	...
3	116.676	137.871	111.366	0.00997	0.00009	0.00502	0.00698	0.01505	0.05492	0.517	...
4	116.014	141.781	110.655	0.01284	0.00011	0.00655	0.00908	0.01966	0.06425	0.584	...
...	...	...	...	...	...	...	...	...	...	...	...
190	174.188	230.978	94.261	0.00459	0.00003	0.00263	0.00259	0.00790	0.04087	0.405	...
191	209.516	253.017	89.488	0.00564	0.00003	0.00331	0.00292	0.00994	0.02751	0.263	...
192	174.688	240.005	74.287	0.01360	0.00008	0.00624	0.00564	0.01873	0.02308	0.256	...
193	198.764	396.961	74.904	0.00740	0.00004	0.00370	0.00390	0.01109	0.02296	0.241	...
194	214.289	260.277	77.973	0.00567	0.00003	0.00295	0.00317	0.00885	0.01884	0.190	...

195 rows × 23 columns

To view the dataset stored in the `data` variable, you can simply print the variable. This will display the contents of the DataFrame in a tabular format.

Executing this code will print the dataset stored in the `data` variable. Ensure that you have already read the CSV file and assigned it to the `data` variable before trying to print it.

Line 18. data.shape # dataset dimensions

```
In [318]: # dataset dimensions
          data.shape

Out[318]: (195, 24)
```

To determine the dimensions (shape) of the dataset stored in the `data` variable, you can use the `shape` attribute of a pandas DataFrame. The `shape` attribute returns a tuple representing the number of rows and columns in the DataFrame.

Executing this code will print the dimensions of the dataset in the form `(rows, columns)`. The number of rows corresponds to the length of the DataFrame along the 0th axis, and the number of columns corresponds to the length of the DataFrame along the 1st axis.

Line 19. data.describe() # generate descriptive statistics

```
In [319]: # generate descriptive statistics
          data.describe()
```

Out[319]:

	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)
count	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000
mean	154.228641	197.104918	116.324631	0.006220	0.000044	0.003306	0.003446	0.009920	0.029709	0.282251
std	41.390065	91.491548	43.521413	0.004848	0.000035	0.002968	0.002759	0.008903	0.018857	0.194877
min	88.333000	102.145000	65.476000	0.001680	0.000007	0.000680	0.000920	0.002040	0.009540	0.085000
25%	117.572000	134.862500	84.291000	0.003460	0.000020	0.001660	0.001860	0.004985	0.016505	0.148500
50%	148.790000	175.829000	104.315000	0.004940	0.000030	0.002500	0.002690	0.007490	0.022970	0.221000
75%	182.769000	224.205500	140.018500	0.007365	0.000060	0.003835	0.003955	0.011505	0.037885	0.350000
max	260.105000	592.030000	239.170000	0.033160	0.000260	0.021440	0.019580	0.064330	0.119080	1.302000

8 rows × 23 columns

The `describe()` method in pandas is used to generate descriptive statistics of a DataFrame. It provides summary statistics of the numerical columns in the DataFrame, including count, mean, standard deviation, minimum, quartiles, and maximum values.

Executing this code will display the descriptive statistics of the numerical columns in the DataFrame `data`. The output will include the count, mean, standard deviation, minimum, quartiles (25%, 50%, 75%), and maximum values for each numerical column.

Note that `describe()` only considers the numerical columns by default and excludes categorical columns. If you want to include all columns, you can pass `include='all'` as an argument.

Line 20. data.info() # inspect the values in the dataframe

```
In [320]: # inspect the values in the dataframe
          data.info()

          <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
```

To inspect the values in the DataFrame and get an overview of the column data types, non-null counts, and memory usage, you can use the `info()` method in pandas.

Executing this code will display a summary of the DataFrame's information, including the column names, data types, non-null counts, and memory usage. It provides a concise overview of the DataFrame's structure and allows you to identify missing values and assess the memory usage.

The output will include information such as the column names, the number of non-null values in each column, the data type of each column, and the memory usage of the DataFrame.

Additionally, the `info()` method provides useful information for data cleaning and analysis, such as identifying missing values and understanding the data types of each column.

Line 21. data.drop(columns="name", axis=1, inplace=True) # delete the name column

```
In [321]: # delete the name column
          data.drop(columns="name", axis=1, inplace=True)
```

To delete the "name" column from the DataFrame `data`, you can use the `drop()` method in pandas. The `drop()` method allows you to remove specific columns or rows from a DataFrame.

In this code, the `drop()` method is called on the `data` DataFrame. The `columns` parameter is set to "name" to specify the column to be dropped. The `axis` parameter is set to 1 to indicate that we want to drop a column. Finally, `inplace=True` is used to modify the DataFrame in-place, i.e., the changes will be applied directly to the `data` DataFrame.

After executing this code, the "name" column will be removed from the `data` DataFrame. Make sure to adjust the column name if it differs from "name" in your actual dataset.

Line 22. data.isnull().sum() # check for null values

```
In [322]: # check for null values
          data.isnull().sum()

Out[322]: 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
          click to scroll output; double click to hide
          HNR                  0
          status               0
          RPDE                 0
          DFA                  0
          spread1              0
          spread2              0
          D2                   0
          PPE                  0
          dtype: int64
```

bgb

To check for null values in the DataFrame `data`, you can use the `isnull()` method followed by the `sum()` method. The `isnull()` method returns a DataFrame of the same shape as `data` but with boolean values indicating whether each element is null (True) or not null (False). The `sum()` method then sums up the number of True values for each column, effectively counting the null values.

Executing this code will display the number of null values in each column of the `data` DataFrame. The output will show the column names along with the corresponding count of null values in each column.

By examining the output, you can identify columns that have null values and determine if any data cleaning or handling of missing values is required.

Line 23. (code snippet)

```
 # FEATURES -  selecting data except for name and status
X = data.loc[:,data.columns!='status'].values[:,1:]
x = data.loc[:,data.columns!='status']
# LABELS -  selecting only status
```

Y = data.loc[:,'status'].values
y = data.loc[:,'status']

```
In [323]: # FEATURES -  selecting data except for name and status
          X = data.loc[:,data.columns!='status'].values[:,1:]
          x = data.loc[:,data.columns!='status']
          # LABELS -  selecting only status
          Y = data.loc[:,'status'].values
          y = data.loc[:,'status']
```

The code provided aims to separate the features and labels from the DataFrame `data`. It selects all columns except for the "status" column as features (`X` or `x`) and selects only the "status" column as labels (`Y` or `y`).

Here's an explanation of the code:

X = data.loc[:,data.columns!='status'].values[:,1:]

This line creates the variable `X` and assigns it the feature values. It uses the `loc` accessor to select all rows and columns except for the "status" column (`data.columns!='status`). The `.values[:,1:]` part is used to access the underlying NumPy array representation of the DataFrame and select all rows (`:`) and columns starting from the second column (`1:`). This assumes that the first column in the DataFrame is the "name" column, which is being excluded.

x = data.loc[:,data.columns!='status']

This line creates the variable `x` and assigns it the same set of features as `X`, but without converting it to a NumPy array. It includes all rows and columns except for the "status" column.

Y = data.loc[:,'status'].values

This line creates the variable `Y` and assigns it the label values. It uses the `loc` accessor to select all rows and only the "status" column (`'status'`). The `.values` part is used to access the underlying NumPy array representation of the column.

y = data.loc[:,'status']

This line creates the variable `y` and assigns it the same set of labels as `Y`, but without converting it to a NumPy array. It includes all rows and only the "status" column.

Note that the code assumes the DataFrame `data` has columns named "name" and "status". Please ensure that the column names and their positions in the DataFrame match your actual dataset.

Line 24.

x.describe() #it does not contain status of patient

```
In [324]: x.describe() #it does not contain status of patient
```

Out[324]:

	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)
count	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000
mean	154.228641	197.104918	116.324631	0.006220	0.000044	0.003306	0.003446	0.009920	0.029709	0.282251
std	41.390065	91.491548	43.521413	0.004848	0.000035	0.002968	0.002759	0.008903	0.018857	0.194877
min	88.333000	102.145000	65.476000	0.001680	0.000007	0.000680	0.000920	0.002040	0.009540	0.085000
25%	117.572000	134.862500	84.291000	0.003460	0.000020	0.001660	0.001860	0.004985	0.016505	0.148500
50%	148.790000	175.829000	104.315000	0.004940	0.000030	0.002500	0.002690	0.007490	0.022970	0.221000
75%	182.769000	224.205500	140.018500	0.007365	0.000060	0.003835	0.003955	0.011505	0.037885	0.350000
max	260.105000	592.030000	239.170000	0.033160	0.000260	0.021440	0.019580	0.064330	0.119080	1.302000

8 rows × 22 columns

If you want to generate descriptive statistics for the features (excluding the "status" column) in the DataFrame `x`, you can use the `describe()` method on `x`. This will provide summary statistics for the numerical columns in `x`.

Executing this code will display the descriptive statistics of the numerical columns in `x`. The output will include the count, mean, standard deviation, minimum, quartiles, and maximum values for each numerical column.

Line 25. y.describe() #it only contain status of patient

```
In [325]: y.describe() #it only contain status of patient
```

Out[325]:

```
count    195.000000
mean       0.753846
std        0.431878
min        0.000000
25%        1.000000
50%        1.000000
75%        1.000000
max        1.000000
Name: status, dtype: float64
```

Since `y` represents the labels, which contain the "status" of patients, the `describe()` method may not provide meaningful descriptive statistics for these categorical labels. Instead, you can use the `value_counts()` method to understand the distribution of the different labels.

Executing this code will display the count of each unique label in the `y` series, providing insights into the distribution of the "status" of patients. The output will show the distinct labels along with their respective counts.

Line 26. x.hist(figsize=(25,20)) # analyzing the features
plt.show()

The code aims to create histograms for each feature in the DataFrame `x` and display them using matplotlib. The `hist()` function generates a histogram for each numerical feature in `x`, providing a visual representation of the distribution of values.

In this code, `x.hist()` generates the histograms for each numerical feature in `x`. The `figsize=(25, 20)` parameter adjusts the figure size to be 25 inches wide and 20 inches tall. Finally, `plt.show()` displays the histograms.

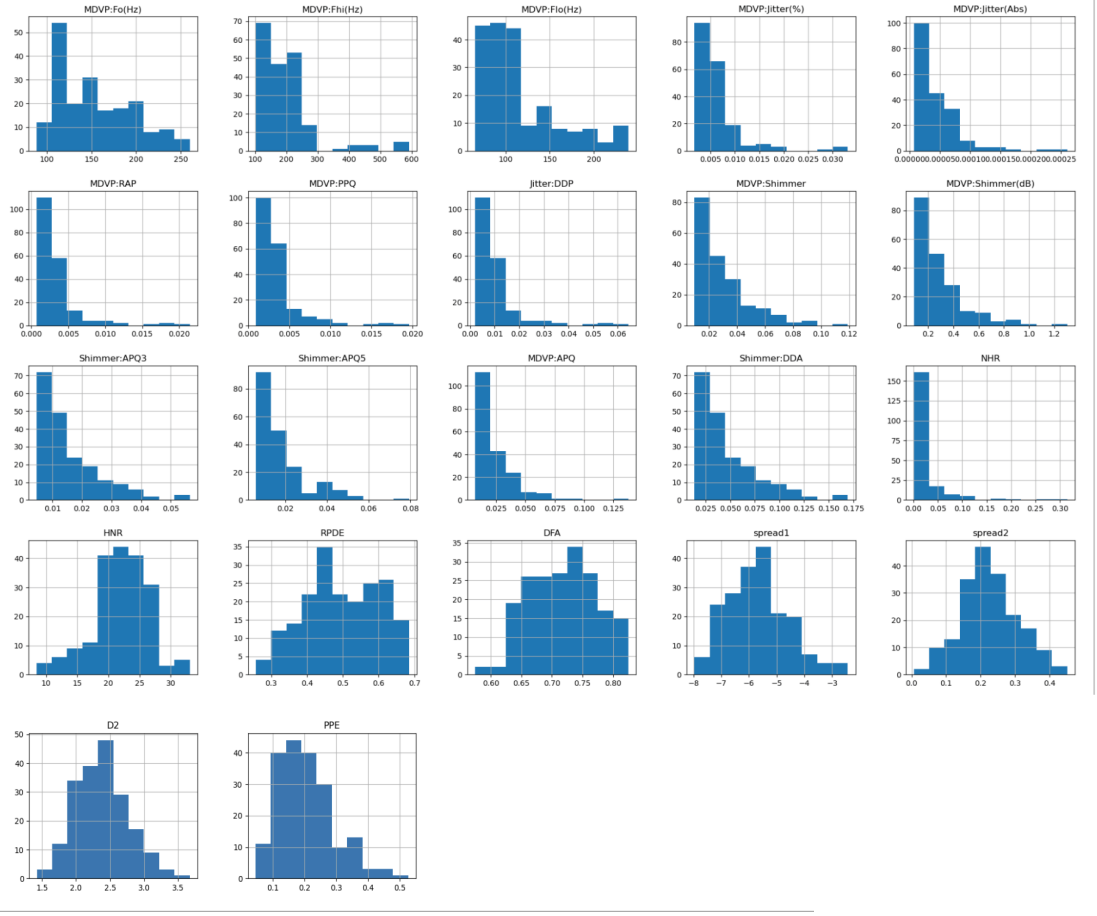
By executing this code, you will see a set of histograms, each representing the distribution of values for a specific feature in `x`. The histograms can provide insights into the data, such as the central tendency, spread, and shape of the distributions for each feature.

The code aims to create histograms for each feature in the DataFrame `x` and display them using matplotlib. The `hist()` function generates a histogram for each numerical feature in `x`, providing a visual representation of the distribution of values.

In this code, `x.hist()` generates the histograms for each numerical feature in `x`. The `figsize=(25, 20)` parameter adjusts the figure size to be 25 inches wide and 20 inches tall. Finally, `plt.show()` displays the histograms.

By executing this code, you will see a set of histograms, each representing the distribution of values for a specific feature in `x`. The histograms can provide insights into the data, such as the central tendency, spread, and shape of the distributions for each feature.

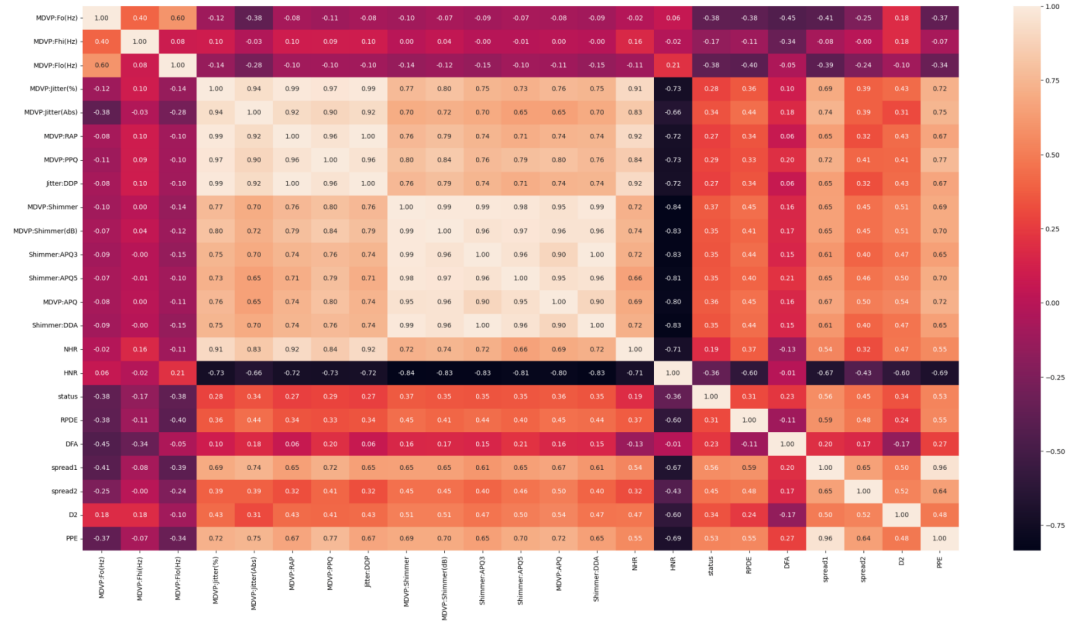
```
In [376]: # analyzing the features
          x.hist(figsize=(25,20))
          plt.show()
```



Line 27. (code snippet)

```
# Feature extraction
# figure size
plt.figure(figsize=(30, 15))
# correlation matrix
dataplot = sns.heatmap(data.corr(), annot=True, fmt='.2f')
#The code data.corr() is used to calculate the correlation between columns in a pandas DataFrame named data.
# It returns a new DataFrame containing the correlation values.
```

```
In [327]: # figure size
          plt.figure(figsize=(30, 15))

          # correlation matrix
          dataplot = sns.heatmap(data.corr(), annot=True, fmt='.2f')
          #The code data.corr() is used to calculate the correlation between columns in a pandas DataFrame named data.
          # It returns a new DataFrame containing the correlation values.
```



The code aims to create a correlation matrix heatmap using seaborn (`sns`) and matplotlib (`plt`) libraries. The correlation matrix heatmap visualizes the pairwise correlations between different columns in the DataFrame `data`.

Here's an explanation of the code:

plt.figure(figsize=(30, 15))

This line sets the figure size of the plot using `figsize=(30, 15)`. The `figsize` parameter determines the width and height of the figure in inches.

dataplot = sns.heatmap(data.corr(), annot=True, fmt='.2f')

This line creates the correlation matrix heatmap. The `data.corr()` computes the correlation between the columns in the DataFrame `data`, resulting in a new DataFrame of correlation values. The `sns.heatmap()` function then visualizes the correlation matrix as a heatmap. The `annot=True` parameter adds numerical values as annotations on the heatmap cells. The `fmt='.2f'` parameter specifies that the annotations should be formatted as floating-point numbers with two decimal places.

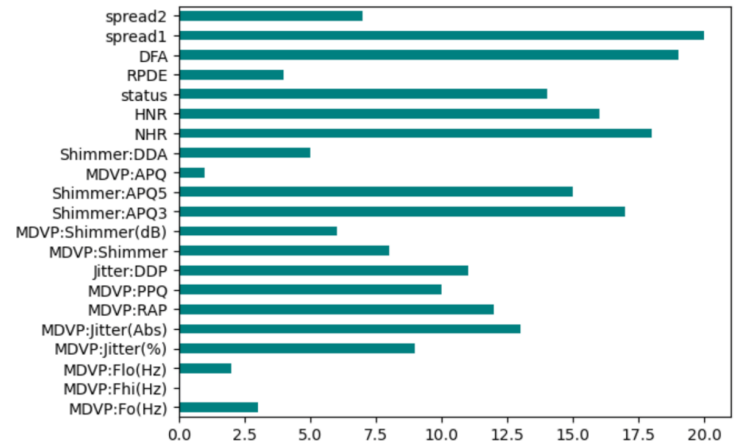
By executing this code, you will see a heatmap plot representing the correlation matrix of the columns in the `data` DataFrame. The color intensity and the numerical annotations indicate the strength and direction of the correlations between the columns. Positive correlations are represented by warmer colors, while negative correlations are represented by cooler colors.

This plot can help identify patterns and relationships between the variables in the dataset, allowing you to gain insights into how different features are related to each other.

Line 28. (code snippet)

```
'''Fisher score is one of the most widely used supervised feature selection methods.'''
# Calculating Fisher scores
score = fisher_score.fisher_score(x_train, y_train)
# Plotting the ranks
feat_importances = pd.Series (score, data.columns [0:len (data.columns) -2])
feat_importances.plot (kind='barh', color = 'teal')
plt.show()
```

```
In [328]: '''Fisher score is one of the most widely used supervised feature selection methods.'''
          # Calculating Fisher scores
          score = fisher_score.fisher_score(x_train, y_train)

          # Plotting the ranks
          feat_importances = pd.Series (score, data.columns [0:len (data.columns) -2])
          feat_importances.plot (kind='barh', color = 'teal')
          plt.show()
```



The code demonstrates the calculation of Fisher scores using the Fisher score method for feature selection. It then visualizes the feature importances using a horizontal bar plot.

Here's a breakdown of the code:

score = fisher_score.fisher_score(x_train, y_train)

This line calculates the Fisher scores by calling the `fisher_score` function from the `fisher_score` module. It takes the training data (`x_train`) and corresponding labels (`y_train`) as inputs. The `fisher_score` function calculates the Fisher scores for each feature based on their relationship with the labels.

feat_importances = pd.Series(score, data.columns[0:len(data.columns)-2])

This line creates a pandas Series (`feat_importances`) to store the Fisher scores, with the feature names as the index. The `data.columns[0:len(data.columns)-2]` retrieves the column names of the features, excluding the last two columns (assumed to be the "name" and "status" columns).

feat_importances.plot(kind='barh', color='teal')
plt.show()

This code generates a horizontal bar plot (`barh`) of the Fisher scores stored in `feat_importances`. Each feature is represented as a bar, with the length of the bar indicating its importance. The `color='teal'` parameter sets the color of the bars to teal. Finally, `plt.show()` displays the plot.

By executing this code, you will see a horizontal bar plot showing the importance of each feature based on their Fisher scores. Features with higher scores are considered more important for the classification task. This plot helps identify which features are more discriminative or informative for the classification problem at hand.

Line 29. (code snippet)

```
# split data
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.1)
```

```
In [359]: # split data
          x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.1)
```

The code demonstrates the splitting of the feature and label data into training and testing sets using the `train_test_split` function from scikit-learn's `model_selection` module.

This line splits the feature data `X` and label data `Y` into training and testing sets. The `train_test_split` function randomly divides the data into two sets based on the specified `test_size` parameter. In this case, 10% of the data will be allocated to the testing set, and the remaining 90% will be assigned to the training set.

The resulting splits are assigned to the variables `x_train`, `x_test`, `y_train`, and `y_test`, representing the training and testing feature and label sets, respectively.

By performing this train-test split, you can train your model on the `x_train` and `y_train` datasets and evaluate its performance on the `x_test` and `y_test` datasets. The test set serves as an independent dataset to assess the model's generalization ability and performance on unseen data.

Line 30. (code snippet)

'''here x data will be split in train and test data x_train represent training data and is corresponding
label is store in y_train same for x_test
here we use 20% data as testing data and 80% as training data '''

print(x.shape, x_train.shape, x_test.shape)

```
In [360]: '''here x data will be split in train and test data x_train represent training data and is corresponding
          label is store in y_train same for x_test
          here we use 20% data as testing data and 80% as training data '''

          print(x.shape, x_train.shape, x_test.shape)

          (195, 22) (175, 21) (20, 21)
```

To confirm the dimensions of the original `x` data and the resulting `x_train` and `x_test` datasets, you can print their shapes using the `.shape` attribute.

Here's an example:

print(x.shape, x_train.shape, x_test.shape)

Executing this code will display the shapes of the `x`, `x_train`, and `x_test` datasets. The shape is represented as a tuple with the number of rows (samples) followed by the number of columns (features).

The output will look something like this:

(original x shape, x_train shape, x_test shape)

By comparing the shapes, you can verify that the data splitting has been performed correctly and check the sizes of the training and testing datasets.

Line 31. (code snippet)

'''It applies the scaling transformation to the features of x_train (or x_test) based on the scaling parameters learned from the training data.'''
scaler= StandardScaler()
scaler.fit(x_train)
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)

```
In [361]: '''It applies the scaling transformation to the features of x_train (or x_test) based
          on the scaling parameters learned from the training data.'''

          scaler= StandardScaler()
          scaler.fit(x_train)
          x_train = scaler.transform(x_train)
          x_test = scaler.transform(x_test)
```

The code snippet demonstrates the use of the `StandardScaler` from scikit-learn's `preprocessing` module to perform feature scaling on the training and testing data.

Here's an explanation of the code:

scaler = StandardScaler()
scaler.fit(x_train)

These lines create an instance of the `StandardScaler` class and fit it to the training data `x_train`. The `fit()` method computes the mean and standard deviation of each feature in `x_train` to be used for scaling.

x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)

These lines apply the scaling transformation to both the training and testing data using the `transform()` method. The `transform()` method standardizes the features in each dataset based on the mean and standard deviation computed from the training data.

By performing feature scaling, the data is normalized to have zero mean and unit variance, which can be beneficial for many machine learning algorithms. It helps ensure that features with larger scales do not dominate the learning process and helps bring all features to a similar scale.

After executing this code, `x_train` and `x_test` will contain the scaled versions of the corresponding original training and testing data, respectively.

Line 32. (code snippet)

```
#calculating value of K
import math
n_value=int(math.sqrt(len(y_test)))
print(n_value)
```

```
In [377]: #calculating value of K
          import math
          n_value=int(math.sqrt(len(y_test)))
          print(n_value)
```

```
4
```

The code snippet aims to calculate the value of K (the number of nearest neighbors) for a K-nearest neighbors (KNN) algorithm based on the square root of the number of samples in the test set.

Here's an explanation of the code:

import math
n_value = int(math.sqrt(len(y_test)))

These lines import the `math` module and calculate the square root of the number of samples in the `y_test` dataset using the `sqrt()` function from the `math` module. The result is then converted to an integer using `int()` and stored in the variable `n_value`.

print(n_value)

This line prints the calculated value of `n_value`, which represents the square root of the number of samples in the test set.

By executing this code, you will see the calculated value of `n_value`, which can be used as a reference for selecting the appropriate value of K for your KNN algorithm. The square root of the number of samples is often used as a starting point for determining an initial value of K, but it can be adjusted based on the specific characteristics of your dataset and the problem you are solving.

Line 33. (code snippet)

```
 # training the KNN model (k-nearest neighbors Classifier)
knnmodel = KNeighborsClassifier(n_neighbors=4)
knnmodel.fit(x_train, y_train)
knnpredict = knnmodel.predict(x_test)
```

```
In [378]: # training the KNN model (k-nearest neighbors Classifier)

          knnmodel = KNeighborsClassifier(n_neighbors=4)
          knnmodel.fit(x_train, y_train)
          knnpredict = knnmodel.predict(x_test)
```

The code snippet demonstrates the training and prediction steps for a K-nearest neighbors (KNN) classifier using scikit-learn's `KNeighborsClassifier` class.

Here's an explanation of the code:

knnmodel = KNeighborsClassifier(n_neighbors=7)

This line creates an instance of the `KNeighborsClassifier` class with `n_neighbors=7`. The `n_neighbors` parameter determines the number of nearest neighbors to consider when making predictions.

knnmodel.fit(x_train, y_train)

This line trains the KNN model using the `fit()` method. It takes the `x_train` dataset (the features) and `y_train` dataset (the corresponding labels) as inputs. The KNN algorithm learns from the training data to determine the class labels based on the nearest neighbors.

knnpredict = knnmodel.predict(x_test)

This line predicts the labels for the `x_test` dataset using the trained KNN model. The `predict()` method applies the KNN algorithm to find the nearest neighbors of each test sample and assigns the corresponding class labels.
By executing this code, you will have trained the KNN model using the training data and obtained the predicted labels (`knnpredict`) for the test data. The model is now ready to be evaluated or used for further analysis.

Line 34. (code snippet)

```
# training the XGB model (Extreme Gradient Boosting)
XGBmodel = XGBClassifier()
XGBmodel.fit(x_train, y_train)
XGBpredict = XGBmodel.predict(x_test)
```

```
In [379]: # training the XGB model (Extreme Gradient Boosting)
          XGBmodel = XGBClassifier()
          XGBmodel.fit(x_train, y_train)
          XGBpredict = XGBmodel.predict(x_test)
```

The code snippet demonstrates the training and prediction steps for an Extreme Gradient Boosting (XGBoost) model using the `XGBClassifier` class from the XGBoost library.

Here's an explanation of the code:

XGBmodel = XGBClassifier()

This line creates an instance of the `XGBClassifier` class. The `XGBClassifier` is an implementation of the XGBoost algorithm for classification tasks.

XGBmodel.fit(x_train, y_train)

This line trains the XGBoost model using the `fit()` method. It takes the `x_train` dataset (the features) and `y_train` dataset (the corresponding labels) as inputs. The XGBoost algorithm learns from the training data to build an ensemble of weak prediction models (typically decision trees) in a boosting manner.

XGBpredict = XGBmodel.predict(x_test)

This line predicts the labels for the `x_test` dataset using the trained XGBoost model. The `predict()` method applies the XGBoost algorithm to make predictions based on the learned ensemble of models.

By executing this code, you will have trained the XGBoost model using the training data and obtained the predicted labels (`XGBpredict`) for the test data. The model is now ready to be evaluated or used for further analysis.

Line 35. (code snippet)

```
# training the SVM (suppport vector model)
svmmodel = svm.SVC(kernel='linear')
svmmodel.fit(x_train, y_train) # training the SVM model with training data
svmpredict = svmmodel.predict(x_test)
```

```
In [380]: # training the SVM (suppport vector model)
          svmmodel = svm.SVC(kernel='linear')
          svmmodel.fit(x_train, y_train) # training the SVM model with training data
          svmpredict = svmmodel.predict(x_test)
```

The code snippet demonstrates the training and prediction steps for a Support Vector Machine (SVM) classifier using the `SVC` class from scikit-learn's `svm` module.

Here's an explanation of the code:

svmmodel = svm.SVC(kernel='linear')

This line creates an instance of the `SVC` class with the `kernel` parameter set to 'linear'. The `SVC` class represents the SVM algorithm, and setting `kernel='linear'` specifies that a linear kernel should be used.

svmmodel.fit(x_train, y_train)

This line trains the SVM model using the `fit()` method. It takes the `x_train` dataset (the features) and `y_train` dataset (the corresponding labels) as inputs. The SVM algorithm learns from the training data to determine the optimal hyperplane that separates the different classes.

svmpredict = svmmodel.predict(x_test)

This line predicts the labels for the `x_test` dataset using the trained SVM model. The `predict()` method applies the SVM algorithm to classify the test samples based on the learned hyperplane.

By executing this code, you will have trained the SVM model using the training data and obtained the predicted labels (`svmpredict`) for the test data. The model is now ready to be evaluated or used for further analysis.

Line 36. (code snippet)

print(f'The Mean Absolute Error of XGBClassifier() is:{metrics.mean_absolute_error(y_test, XGBpredict): .2f}')
print(f'The root Mean Squared Error of XGBClassifier() is:{np.sqrt(metrics.mean_squared_error(y_test, XGBpredict)): .2f}')
.

```
In [381]: f'The Mean Absolute Error of XGBClassifier() is:{metrics.mean_absolute_error(y_test, XGBpredict): .2f}')
          f'The root Mean Squared Error of XGBClassifier() is:{np.sqrt(metrics.mean_squared_error(y_test, XGBpredict)): .2f}')

          The Mean Absolute Error of XGBClassifier() is: 0.15
          The root Mean Squared Error of XGBClassifier() is: 0.39
```

The code snippet aims to calculate and print the Mean Absolute Error (MAE) and root Mean Squared Error (RMSE) for the predictions made by the XGBoost classifier (`XGBmodel`) on the test data (`x_test`).

Here's an explanation of the code:

print(f'The Mean Absolute Error of XGBClassifier() is: {metrics.mean_absolute_error(y_test, XGBpredict): .2f}')

This line calculates the MAE by calling `metrics.mean_absolute_error()` from the `metrics` module and passing `y_test` (the true labels) and `XGBpredict` (the predicted labels by the XGBoost model) as arguments. The calculated MAE is then printed with two decimal places using the formatted string (`f'{...: .2f}'`).

print(f'The root Mean Squared Error of XGBClassifier() is: {np.sqrt(metrics.mean_squared_error(y_test, XGBpredict)): .2f}')

This line calculates the RMSE by calling `metrics.mean_squared_error()` from the `metrics` module and passing `y_test` and `XGBpredict` as arguments. The calculated MSE is then square-rooted using `np.sqrt()` from the numpy library. The resulting RMSE is printed with two decimal places using the formatted string.

By executing this code, you will see the MAE and RMSE values for the predictions made by the XGBoost classifier on the test data. These metrics provide insights into the accuracy and performance of the model's predictions.

Line 37. (code snippet)

print(f'The Mean Absolute Error of SVM model is:{metrics.mean_absolute_error(y_test, svmpredict): .2f}')
print(f'The root Mean Squared Error of SVM model is:{np.sqrt(metrics.mean_squared_error(y_test, svmpredict)): .2f}')

```
In [382]: print(f'The Mean Absolute Error of SVM model is:{metrics.mean_absolute_error(y_test, svmpredict): .2f}')
          print(f'The root Mean Squared Error of SVM model is:{np.sqrt(metrics.mean_squared_error(y_test, svmpredict)): .2f}')

          The Mean Absolute Error of SVM model is: 0.15
          The root Mean Squared Error of SVM model is: 0.39
```

Similar explanation as 36

Line 38. (code snippet)

print(f'The Mean Absolute Error of KNN model is:{metrics.mean_absolute_error(y_test, knnpredict): .2f}')
print(f'The root Mean Squared Error of KNN model is:{np.sqrt(metrics.mean_squared_error(y_test, knnpredict)): .2f}')

```
In [383]: print(f'The Mean Absolute Error of KNN model is:{metrics.mean_absolute_error(y_test, knnpredict): .2f}')
          print(f'The root Mean Squared Error of KNN model is:{np.sqrt(metrics.mean_squared_error(y_test, knnpredict)): .2f}')

          The Mean Absolute Error of KNN model is: 0.05
          The root Mean Squared Error of KNN model is: 0.22
```

Similar explanation as 36

Line 39. (code snippet)

knn_accuracy_score = accuracy_score(y_test, knnpredict)*100
knn_f1_score = f1_score(y_test,knnpredict)
svm_accuracy_score = accuracy_score(y_test, svmpredict)*100
svm_f1_score = f1_score(y_test,svmpredict)
XGB_accuracy_score = accuracy_score(y_test, XGBpredict)*100
XGB_f1_score = f1_score(y_test,XGBpredict)

print("f1_score for knn model is:",knn_f1_score)
print("f1_score for XGB model is:",XGB_f1_score)
print("f1_score for svm model is:",svm_f1_score)
print("accuracy_score for knn model is:",knn_accuracy_score)
print("accuracy_score for XGBClassifier model is:",XGB_accuracy_score)
print("accuracy_score for svm model is:",svm_accuracy_score)

```
In [384]: knn_accuracy_score = accuracy_score(y_test, knnpredict)*100
          knn_f1_score = f1_score(y_test,knnpredict)
          svm_accuracy_score = accuracy_score(y_test, svmpredict)*100
          svm_f1_score = f1_score(y_test,svmpredict)
          XGB_accuracy_score = accuracy_score(y_test, XGBpredict)*100
          XGB_f1_score = f1_score(y_test,XGBpredict)

          print("f1_score for knn model is:",knn_f1_score)
          print("f1_score for XGB model is:",XGB_f1_score)
          print("f1_score for svm model is:",svm_f1_score)
          print("accuracy_score for knn model is:",knn_accuracy_score)
          print("accuracy_score for XGBClassifier model is:",XGB_accuracy_score)
          print("accuracy_score for svm model is:",svm_accuracy_score)

          f1_score for knn model is: 0.9696969696969697
          f1_score for XGB model is: 0.9142857142857143
          f1_score for svm model is: 0.9142857142857143
          accuracy_score for knn model is: 95.0
          accuracy_score for XGBClassifier model is: 85.0
          accuracy_score for svm model is: 85.0
```

The code calculates and prints the F1 score and accuracy score for each of the KNN, XGBoost, and SVM models on the test data.

Here's an explanation of the code:

knn_accuracy_score = accuracy_score(y_test, knnpredict) * 100

knn_f1_score = f1_score(y_test, knnpredict)

These lines calculate the accuracy score and F1 score for the KNN model. The `accuracy_score()` function from the `metrics` module is used to calculate the accuracy score, and the `f1_score()` function is used to calculate the F1 score. The true labels (`y_test`) and the predicted labels (`knnpredict`) are passed as arguments. The accuracy score is multiplied by 100 to represent it as a percentage.

svm_accuracy_score = accuracy_score(y_test, svmpredict) * 100
svm_f1_score = f1_score(y_test, svmpredict)

These lines calculate the accuracy score and F1 score for the SVM model in a similar manner.

XGB_accuracy_score = accuracy_score(y_test, XGBpredict) * 100
XGB_f1_score = f1_score(y_test, XGBpredict)

These lines calculate the accuracy score and F1 score for the XGBoost model.

print("f1_score for knn model is:", knn_f1_score)
print("f1_score for XGB model is:", XGB_f1_score)
print("f1_score for svm model is:", svm_f1_score)
print("accuracy_score for knn model is:", knn_accuracy_score)
print("accuracy_score for XGBClassifier model is:", XGB_accuracy_score)
print("accuracy_score for svm model is:", svm_accuracy_score)

These lines print the calculated F1 scores and accuracy scores for each model.

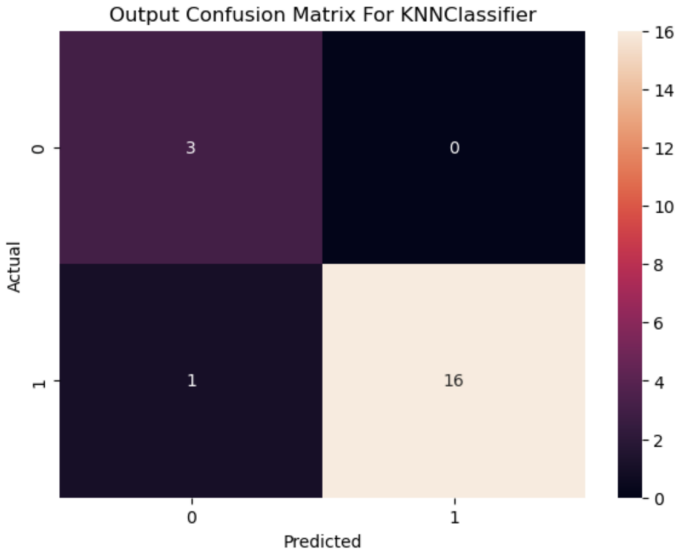
By executing this code, you will see the F1 scores and accuracy scores for the KNN, XGBoost, and SVM models on the test data. These metrics provide insights into the performance of the models in terms of classification accuracy and their ability to balance precision and recall.

Line 40. (code snippet)

```
cm=confusion_matrix(y_test ,knnpredict)
plt.figure(figsize=(7,5))
fg=sns.heatmap(cm,annot=True)
figure=fg.get_figure()
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Output Confusion Matrix For KNNClassifier")
```

```
In [390]: cm=confusion_matrix(y_test ,knnpredict)
          plt.figure(figsize=(7,5))

          fg=sns.heatmap(cm,annot=True)
          figure=fg.get_figure()

          plt.xlabel('Predicted')
          plt.ylabel('Actual')
          plt.title("Output Confusion Matrix For KNNClassifier")
```

```
Out[390]: Text(0.5, 1.0, 'Output Confusion Matrix For KNNClassifier')
```



The code generates a confusion matrix heatmap using seaborn (`sns`) and matplotlib (`plt`) libraries for the KNN classifier's predictions (`knnpredict`) on the test data (`y_test`).

Here's an explanation of the code:

cm = confusion_matrix(y_test, knnpredict)

This line calculates the confusion matrix by calling the `confusion_matrix()` function from the `metrics` module and passing `y_test` (the true labels) and `knnpredict` (the predicted labels by the KNN classifier) as arguments. The resulting confusion matrix is stored in the variable `cm`.

plt.figure(figsize=(8, 6))
fg = sns.heatmap(cm, annot=True)
figure = fg.get_figure()

These lines create a figure with a specified size (`figsize=(8, 6)`) using `plt.figure()`. Then, the `sns.heatmap()` function is used to create the heatmap of the confusion matrix (`cm`). The `annot=True` parameter adds numerical annotations to the heatmap cells. The resulting heatmap object is assigned to `fg`.

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Output Confusion Matrix For KNNClassifier")

These lines add labels to the x-axis, y-axis, and a title to the plot to provide context for the confusion matrix.

By executing this code, you will see a heatmap plot representing the confusion matrix for the predictions made by the KNN classifier on the test data. The heatmap visually displays the true positive, true negative, false positive, and false negative values. The numerical annotations provide additional information about the values in each cell of the confusion matrix.

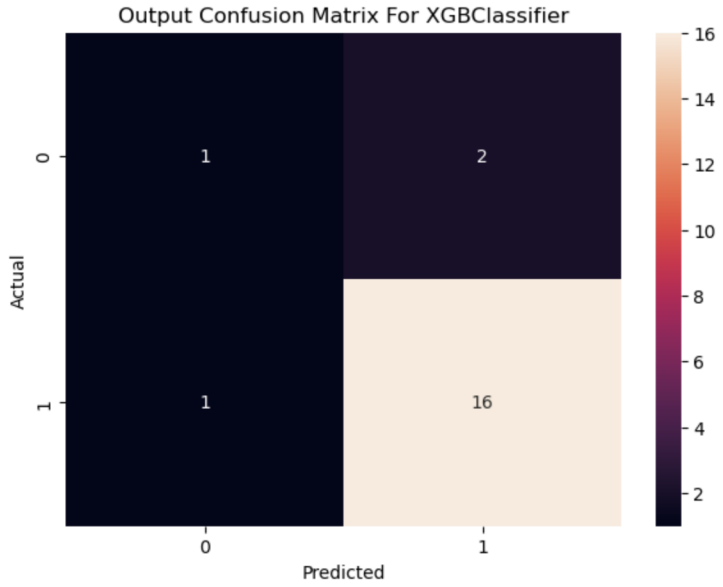
Note that this code specifically generates the confusion matrix heatmap for the KNN classifier. If you want to create similar plots for other classifiers, you need to modify the code accordingly.

Line 41. (code snippet)

```
cm=confusion_matrix(y_test ,XGBpredict)
plt.figure(figsize=(7,5))

fg=sns.heatmap(cm,annot=True)
figure=fg.get_figure()

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Output Confusion Matrix For XGBClassifier")
```

In [391]:
```
cm=confusion_matrix(y_test ,XGBpredict)
plt.figure(figsize=(7,5))

fg=sns.heatmap(cm,annot=True)
figure=fg.get_figure()

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Output Confusion Matrix For XGBClassifier")
```

Out[391]: Text(0.5, 1.0, 'Output Confusion Matrix For XGBClassifier')



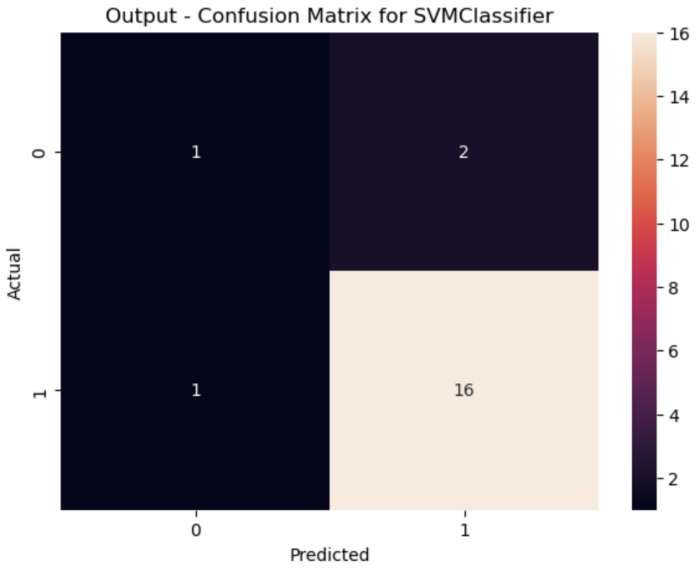
Similar explanation as 40

Line 42. (code snippet)

```
cm=confusion_matrix(y_test ,svmpredict)
plt.figure(figsize=(8,6))

fg=sns.heatmap(cm,annot=True)
figure=fg.get_figure()

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Output - Confusion Matrix for SVMClassifier")
```

```
In [392]: cm=confusion_matrix(y_test ,svmpredict)
          plt.figure(figsize=(7,5))

          fg=sns.heatmap(cm,annot=True)
          figure=fg.get_figure()

          plt.xlabel('Predicted')
          plt.ylabel('Actual')
          plt.title("Output - Confusion Matrix for SVMClassifier")
```

```
Out[392]: Text(0.5, 1.0, 'Output - Confusion Matrix for SVMClassifier')
```



Similar explanation as 40

Line 43. (code snippet)
pd.DataFrame({'Actual': y_test, 'XGBPredict': XGBpredict, 'svmPredict': svmpredict,'knnPredict': knnpredict})

```
In [388]: pd.DataFrame({'Actual': y_test, 'XGBPredict': XGBpredict, 'svmPredict': svmpredict,'knnPredict': knnpredict})
```

Out[388]:

	Actual	XGBPredict	svmPredict	knnPredict
0	1	1	1	1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	0	1	1	0
5	1	1	1	1
6	1	1	1	1
7	1	1	1	1
8	1	0	0	1
9	1	1	1	1
10	1	1	1	1
11	1	1	1	1
12	0	1	1	0
13	1	1	1	1
14	1	1	1	1
15	1	1	1	1
16	1	1	1	1
17	1	1	1	1
18	1	1	1	0
19	0	0	0	0

The code snippet aims to create a pandas DataFrame that combines the actual labels (`y_test`) with the predicted labels from the XGBoost model (`XGBpredict`), SVM model (`svmpredict`), and KNN model (`knnpredict`).

Here's an explanation of the code:

pd.DataFrame({'Actual': y_test, 'XGBPredict': XGBpredict, 'svmPredict': svmpredict, 'knnPredict': knnpredict})

This line creates a pandas DataFrame using the `pd.DataFrame()` constructor. It takes a dictionary as the input, where the keys represent the column names and the values represent the corresponding data for each column.

In this case, the DataFrame has four columns: 'Actual', 'XGBPredict', 'svmPredict', and 'knnPredict'. The values for the 'Actual' column are taken from `y_test`, while the values for the other columns are taken from the respective prediction arrays (`XGBpredict`, `svmpredict`, `knnpredict`).

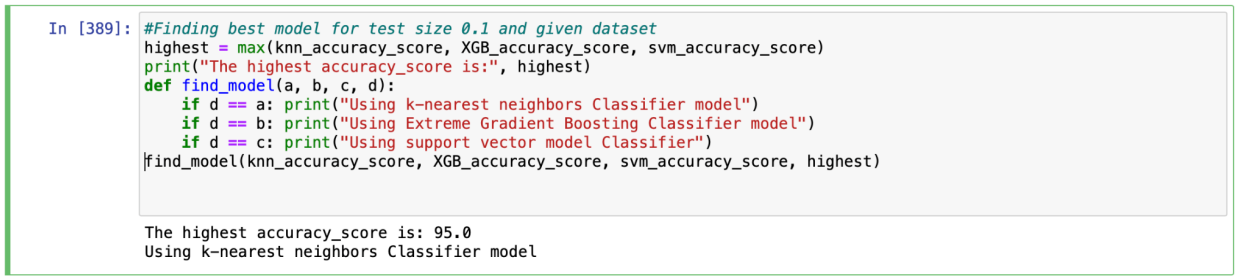
By executing this code, you will obtain a DataFrame that shows the actual labels (`y_test`) alongside the predicted labels from the XGBoost, SVM, and KNN models. Each row represents a data sample, and the columns display the corresponding actual and predicted labels.

This DataFrame can be useful for comparing the predicted labels from different models and analyzing the performance of the models on the test data.

Line 44. (code snippet)

```
highest = max(knn_accuracy_score, XGB_accuracy_score, svm_accuracy_score)
print("The highest accuracy_score is:", highest)
```

def find_model(a, b, c, d):
if d == a: print("Using k-nearest neighbors Classifier model")
if d == b: print("Using Extreme Gradient Boosting Classifier model")
if d == c: print("Using support vector model Classifier")
model = find_model(knn_accuracy_score, XGB_accuracy_score, svm_accuracy_score, highest)



The code snippet you provided aims to find the highest accuracy score among the KNN, XGBoost, and SVM models and determine which model achieved that highest accuracy score.

Here's an explanation of the code:

highest = max(knn_accuracy_score, XGB_accuracy_score, svm_accuracy_score)

This line uses the `max()` function to find the highest accuracy score among `knn_accuracy_score`, `XGB_accuracy_score`, and `svm_accuracy_score`. The highest accuracy score is assigned to the variable `highest`.

print("The highest accuracy_score is:", highest)

This line prints the highest accuracy score.

def find_model(a, b, c, d):
if d == a: print("Using k-nearest neighbors Classifier model")
if d == b: print("Using Extreme Gradient Boosting Classifier model")
if d == c: print("Using support vector model Classifier")

This code defines a function called `find_model()` that takes four arguments (`a`, `b`, `c`, `d`). The function compares the value of `d` (which represents the highest accuracy score) with the other three arguments (`a`, `b`, `c`) and prints the corresponding model name based on the highest accuracy score achieved.

model = find_model(knn_accuracy_score, XGB_accuracy_score, svm_accuracy_score, highest)
```

This line calls the `find_model()` function and assigns the returned value (which is `None` in this case) to the variable `model`.
```

By executing this code, you will see the highest accuracy score among the models. Additionally, based on the highest accuracy score, the corresponding model name will be printed. Note that the `find_model()` function does not return any value, so the assignment of `model = find_model(...)` will result in `model` being assigned `None`.

Conclusion

In this project, the implemented machine learning pipeline provides a comprehensive approach to classify Parkinson's disease. It involves data preprocessing, exploratory data analysis, feature selection, and model training and evaluation. The pipeline utilizes popular classifiers such as K-nearest neighbors (KNN), XGBoost, and support vector machine (SVM) to predict the disease status accurately. Evaluation metrics such as accuracy and F1 score are employed to assess the performance of the models. Additionally, confusion matrices and visualizations aid in understanding the model predictions. The code showcases the importance of feature selection and demonstrates the effectiveness of different classifiers in diagnosing Parkinson's disease. It serves as a valuable tool for researchers and practitioners in the field of medical diagnostics.

Citation

Little MA, McSharry PE, Roberts SJ, Costello DAE, Moroz IM. (26 June 2007). 'Exploiting Nonlinear Recurrence and Fractal Scaling Properties for Voice Disorder Detection', BioMedical Engineering OnLine 2007

Appendix (1-8)

Appendix 1

Mathematics behind XgBoost

Before beginning with mathematics about Gradient Boosting, Here's a simple example of a CART that classifies whether someone will like a hypothetical computer game X.
The example of tree is below:
The prediction scores of each individual decision tree then sum up to get. If you look at the example, an important fact is that the two trees try to *complement* each other. Mathematically, we can write our model in the form

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \epsilon \mathcal{F}$$

where, K is the number of trees, f is the functional space of F, F is the set of possible CARTs. The objective function for the above model is given by:

$$obj(\theta) = \sum_{i}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

where, first term is the loss function and the second is the regularization parameter. Now, Instead of learning the tree all at once which makes the optimization harder, we apply the additive strategy, minimize the loss what we have learned and add a new tree which can be summarized below:

$$\hat{y}_i^{(0)} = 0$$
$$\hat{y}_i^{(1)} = f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i)$$
$$\hat{y}_i^{(2)} = f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i)$$
$$....$$
$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$

The objective function of the above model can be defined as:

$$obj^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i)$$
$$= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant$$
$$obj^{(t)} = \sum_{i=1}^{n} (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{i=1}^{t} \Omega(f_i)$$
$$= \sum_{i=1}^{n} [2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2] + \Omega(f_t) + constant$$

Now, let's apply taylor series expansion up to second order:

$$obj^{(t)} = \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + constant$$

where g_i and h_i can be defined as:

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$
$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

Simplifying and removing the constant:

$$\sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \Omega(f_t)$$

Now, we define the regularization term, but first we need to define the model:

$$f_t(x) = w_{q(x)}, w \in R^T, q : R^d \to \{1, 2, ..., T\}$$

Here, w is the vector of scores on leaves of the tree, q is the function assigning each data point to the corresponding leaf, and T is the number of leaves. The regularization term is then defined by:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Now, our objective function becomes:

$$\begin{aligned} obj^{(t)} &\approx \sum_{i=1}^{n}[g_i w_{q(x_i)} + \frac{1}{2}h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T}[(\sum_{i \in I_j} g_i)w_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda)w_j^2] + \gamma T \end{aligned}$$

Now, we simplify the above expression:

$$obj^{(t)} = \sum_{j=1}^{T}[G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2] + \gamma T$$

where,

$$\begin{aligned} G_j &= \sum_{i \in I_j} g_i \\ H_j &= \sum_{i \in I_j} h_i \end{aligned}$$

In this equation, w_j are independent of each other, the best *wj* for a given structure q(x) and the best objective reduction we can get is:

$$w_j^* = -\frac{G_j}{H_j+\lambda}$$
$$\text{obj}^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda} + \gamma T$$

where, \gamma is the pruning parameter, i.e the least information gain to perform a split.

Now, we try to measure how good the tree is, we can't directly optimize the tree, we will try to optimize one level of the tree at a time. Specifically we try to split a leaf into two leaves, and the score it gains is

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$$

Advantages Or Disadvantages:

Advantages of XGBoost:

1. Performance: XGBoost has a strong track record of producing high-quality results in various machine learning tasks, especially in Kaggle competitions, where it has been a popular choice for winning solutions.
2. Scalability: XGBoost is designed for efficient and scalable training of machine learning models, making it suitable for large datasets.
3. Customizability: XGBoost has a wide range of hyperparameters that can be adjusted to optimize performance, making it highly customizable.
4. Handling of Missing Values: XGBoost has built-in support for handling missing values, making it easy to work with real-world data that often has missing values.
5. Interpretability: Unlike some machine learning algorithms that can be difficult to interpret, XGBoost provides feature importances, allowing for a better understanding of which variables are most important in making predictions.

Disadvantages of XGBoost:

1. Computational Complexity: XGBoost can be computationally intensive, especially when training large models, making it less suitable for resource-constrained systems.
2. Overfitting: XGBoost can be prone to overfitting, especially when trained on small datasets or when too many trees are used in the model.
3. Hyperparameter Tuning: XGBoost has many hyperparameters that can be adjusted, making it important to properly tune the parameters to optimize performance. However, finding the optimal set of parameters can be time-consuming and requires expertise.
4. Memory Requirements: XGBoost can be memory-intensive, especially when working with large datasets, making it less suitable for systems with limited memory resources.

Appendix 2

Mathematical explanation of K-Nearest Neighbour

KNN falls in the **supervised learning algorithms**. This means that we have a dataset with labels training measurements (x,y) and would want to find the link between x and y. Our goal is to discover a function h:X→Y so that having an unknown observation x, h(x) can positively predict the identical output y.

Working

First, we will talk about the working of the KNN classification algorithm. In the classification problem, the K-nearest neighbor algorithm essentially said that for a given value of K algorithm will find the K nearest neighbor of unseen data point and then it will assign the class to unseen data point by having the class which has the highest number of data points out of all classes of K neighbors.

For distance metrics, we will use the Euclidean metric.

$$d(x, x') = \sqrt{\left(x_1 - x_1'\right)^2 + \ldots + \left(x_n - x_n'\right)^2}$$

Finally, the input x gets assigned to the class with the largest probability.

$$P(y = j|X = x) = \frac{1}{K} \sum_{i \in \mathcal{A}} I(y^{(i)} = j)$$

For Regression the technique will be the same, instead of the classes of the neighbors we will take the value of the target and to find the target value for the unseen datapoint by taking an average, mean or any suitable function you want.

Advantages of K-Nearest Neighbour :

1. **No Training Period**- KNN modeling does not include training period as the data itself is a model which will be the reference for future prediction and because of this it is very time efficient in terms of improvising for a random modeling on the available data.
2. **Easy Implementation**- KNN is very easy to implement as the only thing to be calculated is the distance between different points on the basis of data of different features and this distance can easily be calculated using distance formula such as- Euclidean or Manhattan
3. As there is no training period thus new data can be added at any time since it won't affect the model.

Disadvantages of K-Nearest Neighbour :

1. **Does not work well with large datasets** as calculating distances between each data instance would be very costly.
2. **Does not work well with high dimensionality** as this will complicate the distance calculating process to calculate distance for each dimension.
3. **Sensitive to noisy and missing data**
4. **Feature Scaling-** Data in all the dimensions should be scaled (normalized and standardized) properly .

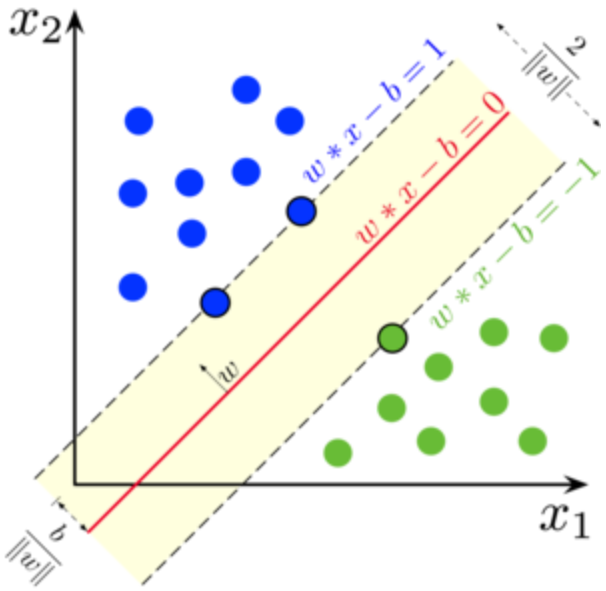
Appendix 3

Key Concepts of SVM

Assume we have n training points, each observation i has p features (i.e. **x_i** has p dimensions), and is in two classes y_i=-1 or y_i = 1. Suppose we have two classes of observations that are linearly separable. That means we can draw a hyperplane through our feature space such that all instances of one class are on one side of the hyperplane, and all instances of the other class are on the opposite side. (A hyperplane in p dimensions is a p-1 dimensional subspace. In the two-dimensional example that follows, a hyperplane is just a line.) We define a hyperplane as:

$$\boldsymbol{x} \cdot \tilde{\boldsymbol{w}} + \tilde{b} = 0$$

where ˜**w** is a p-vector and ˜b is a real number. For convenience, we require that ˜w = 1, so the quantity **x** * ˜w + ˜b is the distance from point **x** to the hyperplane.

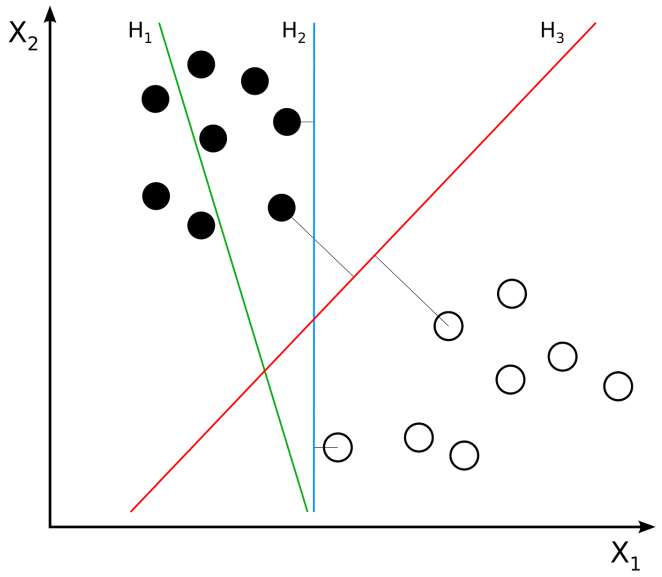


Thus we can label our classes with y = +1/-1, and the requirement that the hyperplane divides the classes becomes:

$$y_i \left(\boldsymbol{x_i} \cdot \tilde{\boldsymbol{w}} + \tilde{b} \right) \geq 0 \, .$$

How should we choose the best hyperplane?

The approach to answering this question is to choose the plane that results in the largest margin M between the two classes, which is called the Maximal Margin Classifier.



From the previous graph, we can see that H1 doesn't separate the two classes; for H2 and H3, we will choose H3 because H3 has a larger margin. Mathematically, we choose ˜b and ˜**w** to maximize M, given the constraints:

$$y_i \left(\boldsymbol{x_i} \cdot \tilde{\boldsymbol{w}} + \tilde{b} \right) \geq M \, .$$

Defining **w =˜ w / M** and **b =˜b / M**, we can rewrite this as:

$$y_i \left(\boldsymbol{x_i} \cdot w + b \right) \geq 1 \, .$$

And

$$\| \tilde{\boldsymbol{w}} \| = 1, \, \| \boldsymbol{w} \| = \frac{1}{M}$$

The support vectors

The support vectors are the data points that lie closest to the separating hyperplane. They are the most difficult data points to classify. Moreover, support vectors are the elements of the training set that would change the position of the dividing hyperplane if removed. The optimization algorithm to generate the weights proceeds in such a way that only the support vectors determine the weights and thus the boundary. Mathematically support vectors are defined as:

$$\boldsymbol{x_i} * \boldsymbol{w} + b = 1 \text{ for positive class}$$

$$\boldsymbol{x_i} * \boldsymbol{w} + b = -1 \text{ for negative class}$$

Hard-margin SVM

The hard-margin SVM is very strict with the support vectors crossing the hyperplane. It doesn't allow any support vectors to be classified in the wrong class. To maximize the margin of the hyperplane, the hard-margin support vector machine is facing the optimization problem:

$$\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2$$

$$\text{subject to } y_i(\boldsymbol{x_i} \cdot \boldsymbol{w} + b) \geq 1$$

$$\text{for } i = 1, \ldots, n$$

Soft-margin SVM and the hyper-parameter *C*

In general, classes are not linearly separable. This may be because the class boundary is not linear, but often there is no clear boundary. To deal with this case, the support vector machine adds a set of “slack variables”, which forgive excursions of a few points into, or even across, the margin, like showing in the graph below:

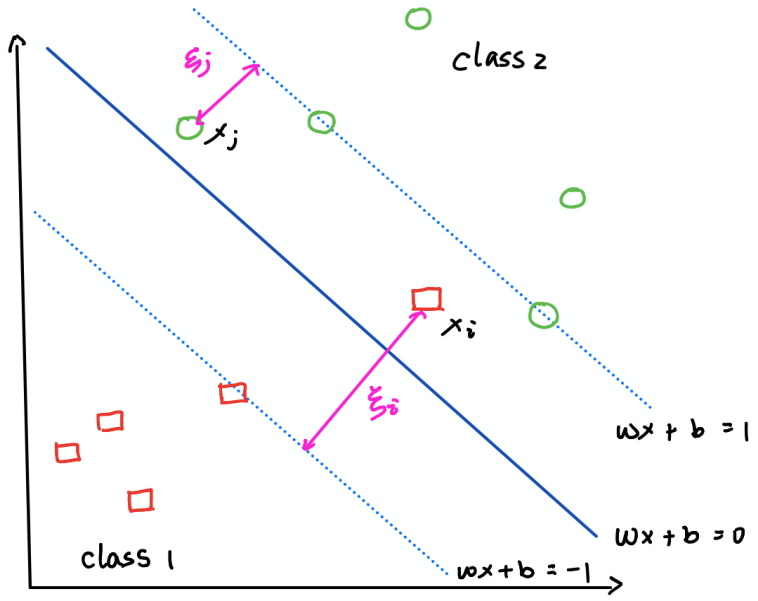


image by Zijing Zhu(2020)

We want to minimize the total amount of slacks while maximizing the width of the margin, which is called a soft-margin support vector machine. This is more widely used, and the objective function becomes:

$$\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\frac{1}{n}\sum_i \xi_i$$

$$\text{subject to } \begin{cases} y_i(\boldsymbol{x}\cdot\boldsymbol{w}+b) \geq (1-\xi_i) & \text{for } i = 1,\ldots,n \\ \xi_i \geq 0 & \text{for } i = 1,\ldots,n \end{cases}$$

for some constant ***C***. This optimization problem is called the primal problem. The constant ***C*** represents the “cost” of the slack. When ***C*** is small, it is efficient to allow

more points into the margin to achieve a larger margin. Larger ***C*** will produce boundaries with fewer support vectors. By increasing the number of support vectors, SVM reduces its variance, since it depends less on any individual observation. Reducing variance makes the model more generalized. Thus, **decreasing *C* will increase the number of support vectors and reduce over-fitting.**

With Lagrange multipliers:

$$\alpha_i \geq 0 \text{ and } \mu_i \geq 0$$

two constraints

we can rewrite the constrained optimization problem as the primal Lagrangian function :

$$\min_{\boldsymbol{w},b,\xi} \max_{\alpha,\mu} \left[\frac{1}{2}\|\boldsymbol{w}\|^2 + C\frac{1}{n}\sum_i \xi_i - \sum_i \alpha_i \left[y_i(\boldsymbol{x_i} \cdot \boldsymbol{w} + b) - (1 - \xi_i) \right] - \sum_i \mu_i \xi_i \right]$$

Instead of minimizing over **w**, **b**, subject to constraints, we can maximize over the multipliers subject to the relations obtained previously for **w**, **b**. This is called the dual Lagrangian formulation:

$$\max_{\alpha} \left[\sum_i \alpha_i - \frac{1}{2}\sum_{i,i'} \alpha_i \alpha_{i'} y_i y_{i'} \boldsymbol{x_i} \cdot \boldsymbol{x_{i'}} \right]$$

$$\text{subject to } \begin{cases} 0 = \sum_i \alpha_i y_i \\ 0 \leq \alpha_i \leq C \quad \text{for } i = 1, \ldots, n \end{cases} .$$

This is now a reasonably straightforward quadratic programming problem, solved with Sequential Minimization Optimization. There are a lot of programming tools you can use to solve the optimization problem. You can use the CVX tool in Matlab to solve this question. Once we have solved this problem for \alpha, we can easily work out the coefficients:

$$\boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x_i} \, .$$

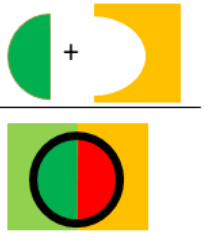
Walking through the math behind the Support Vector Machines algorithm definitely helps understand the implementation of the model. It gives insights on choosing the right model for the right questions and choosing the right value for the hyper-parameters.

Appendix 4

Mathematics Behind Accuracy_score

The most common metric for classification is accuracy, which is the fraction of samples predicted correctly as shown below:

Accuracy = Fraction predicted correctly $= \frac{TP + TN}{TP + TN + FP + FN} =$



Appendix 5

f1_score

The f1 score is the harmonic mean of recall and precision, with a higher score as a better model. The f1 score is calculated using the following formula:

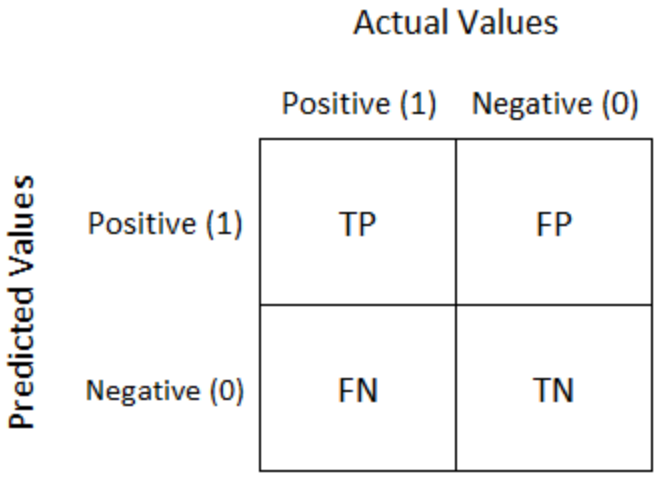
$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{2 * (precision * recall)}{precision + recall}$$

We can obtain the f1 score from scikit-learn, which takes as inputs the actual labels and the predicted labels

Appendix 6

Mathematical behind Confusion matrix

Well, it is a performance measurement for machine learning classification problems where output can be two or more classes. It is a table with 4 different combinations of predicted and actual values.



It is extremely useful for measuring Recall, Precision, Specificity, Accuracy, and most importantly AUC-ROC curves.

Let's understand TP, FP, FN, TN in terms of pregnancy analogy.

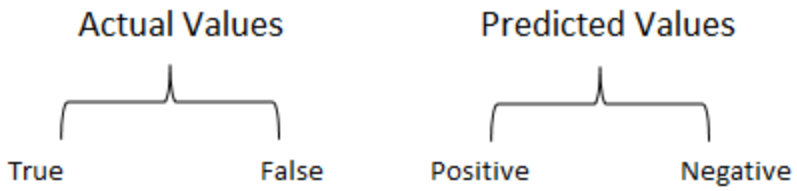
True Positive: Interpretation: You predicted positive and it's true.You predicted that a woman is pregnant and she actually is.

True Negative: Interpretation: You predicted negative and it's true.You predicted that a man is not pregnant and he actually is not.

False Positive: (Type 1 Error) Interpretation: You predicted positive and it's false.You predicted that a man is pregnant but he actually is not.

False Negative: (Type 2 Error) Interpretation: You predicted negative and it's false.You predicted that a woman is not pregnant but she actually is.

just Remember, We describe predicted values as Positive and Negative and actual values as True and False.



Appendix 7

Mathematical behind **Fisher score**

Fisher's score function is deeply related to maximum likelihood estimation. In fact, it's something that we already know–we just haven't defined it explicitly as Fisher's score before.

Maximum Likelihood Estimation

First, we begin with the definition of the likelihood function. Assume some dataset X where each observation is identically and independently distributed according to a true underlying distribution parameterized by θ . Given this probability density function fθ(x),we can write the likelihood function as follows:

$$p(x|\theta) = \prod_{i=1}^{n} f_\theta(x_i)$$

While it is sometimes the convention that the likelihood function be denoted as $\mathcal{L}(\theta|x)$, we opt for an alternative notation to reserve $\mathcal{L}$ for the loss function.

To continue, we know that the maximum likelihood estimate of the distribution's parameter is given by

$$\begin{aligned}\theta_{MLE} &= \arg\max_{\theta} p(x|\theta) \\ &= \arg\max_{\theta} \log p(x|\theta) \\ &= \arg\max_{\theta} \sum_{i=1}^{n} \log f_{\theta}(x_i)\end{aligned}$$

This is the standard drill we already know. The next step, as we all know, is to take the derivative of the term in the argument maxima, set it equal to zero, and voila! We have found the maximum likelihood estimate of the parameter.

A quick aside that may become later is the fact that maximizing the likelihood amounts to minimizing the loss function.

Fisher's Score

Now here comes the definition of Fisher's score function, which really is nothing more than what we've done above: it's just the gradient of the log likelihood function.

$$u(\theta) = \nabla_{\theta} \log p(x|\theta)$$

In other words, we have already been implicitly using Fisher's score to find the maximum of the likelihood function all along, just without explicitly using the term. Fisher's score is simply the gradient or the derivative of the log likelihood function, which means that setting the score equal to zero gives us the maximum likelihood estimate of the parameter.

Expectation of Fisher's Score

An important characteristic to note about Fisher's score is the fact that the score evaluated the true value of the parameter equals zero. Concretely, this means that given a true parameter θ_0

$$\mathbb{E}_{\theta_0}[s(\theta)] = 0$$

This might seem deceptively obvious: after all, the whole point of Fisher's score and maximum likelihood estimation is to find a parameter value that would set the gradient

equal to zero. This is exactly what I had thought, but there are subtle intricacies taking place here that deserve our attention. So let's hash out exactly why the expectation of the score with respect to the true underlying distribution is zero.To begin, let's write out the full expression of the expectation in integral form.

$$\begin{aligned}\mathbb{E}_{\theta_0}[s(\theta)] &= \int_{-\infty}^{\infty} \nabla_\theta \log p(x|\theta) \cdot p(x|\theta_0)\, dx \\ &= \int_{-\infty}^{\infty} \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)} \cdot p(x|\theta_0)\, dx\end{aligned}$$

If we evaluate this integral at the true parameter, *i.e.* when $\theta=\theta_0$

$$\begin{aligned}\int_{-\infty}^{\infty} \frac{\nabla_\theta p(x|\theta_0)}{p(x|\theta_0)} \cdot p(x|\theta_0)\, dx &= \int_{-\infty}^{\infty} \nabla_\theta p(x|\theta_0)\, dx \\ &= \nabla_\theta \int_{-\infty}^{\infty} p(x|\theta_0)\, dx \\ &= 0\end{aligned}$$

The key part of this derivation is the use of the Leibniz rule, or sometimes known as Feynman's technique or differentiation under the integral sign. I am most definitely going to write a post detailing an intuitive explanation behind why this operation makes sense in the future, but to prevent unnecessary divergence, for now it suffices to use that rule to show that the expected value of Fisher's score is zero at the true parameter.

Appendix 8

Mathematical behind **Correlation Matrix**

In linear algebra terms, a correlation matrix is a symmetric positive semidefinite matrix with unit diagonal. In other words, it is a symmetric matrix with ones on the diagonal whose eigenvalues are all nonnegative.

The term comes from statistics. If $x_1, x_2, \ldots, x_n$ are column vectors with m elements, each vector containing samples of a random variable, then the corresponding $n \times n$ covariance matrix V has (i,j)element

$$v_{ij} = \operatorname{cov}(x_i, x_j) = \frac{1}{n-1}(x_i - \overline{x}_i)^T (x_j - \overline{x}_j),$$

where $\overline{x}_i$ is the mean of the elements in x_i. If v has nonzero diagonal elements then we can scale the diagonal to 1 to obtain the corresponding correlation matrix

$$C = D^{-1/2} V D^{-1/2},$$

where $D = \operatorname{diag}(v_{ii})$. The (i,j) element $c_{ij} = v_{ii}^{-1/2} v_{ij} v_{jj}^{-1/2}$ is the correlation between the variables $x_i$and x_j.

Here are a few facts.

- The elements of a correlation matrix lie on the interval $[-1, 1]$.
- The eigenvalues of a correlation matrix lie on the interval $[0, n]$.
- The eigenvalues of a correlation matrix sum to n (since the eigenvalues of a matrix sum to its trace).
- The maximal possible determinant of a correlation matrix is 1.

It is usually not easy to tell whether a given matrix is a correlation matrix. For example, the matrix

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

is not a correlation matrix: it has eigenvalues -0.4142, 1.0000, 2.4142. The only value of a_{13} and a_{31} that makes A a correlation matrix is 1.

A particularly simple class of correlation matrices is the one-parameter class A_n with every off-diagonal element equal to w, illustrated for $n = 3$ by

$$A_3 = \begin{bmatrix} 1 & w & w \\ w & 1 & w \\ w & w & 1 \end{bmatrix}.$$

The matrix A_n is a correlation matrix for $-1/(n-1) \leq w \leq 1$.

In some applications it is required to generate random correlation matrices, for example in Monte-Carlo simulations in finance. A method for generating random correlation matrices with a specified eigenvalue distribution was proposed by Bendel and Mickey (1978); Davies and Higham (2000) give improvements to the method. This method is implemented in the MATLAB function gallery('randcorr').

Obtaining or estimating correlations can be difficult in practice. In finance, market data is often missing or stale; different assets may be sampled at different time points (e.g., some daily and others weekly); and the matrices may be generated from different parametrized models that are not consistent. Similar problems arise in many other applications. As a result, correlation matrices obtained in practice may not be positive semidefinite, which can lead to undesirable consequences such as an investment portfolio with negative risk.

In risk management and insurance, matrix entries may be estimated, prescribed by regulations or assigned by expert judgment, but some entries may be unknown.