HEALTHCARE ANALYTICS FOR DISEASE PREDICTION PROJECT DOCUMENTATION

Project Scope:

- Analyze healthcare data to predict the likelihood of a disease.
- Utilize machine learning algorithms for classification.
- Provide insights for early intervention and prevention.

Module 1	Data Collection and Preprocessing
Module 2	Exploratory Data Analysis
Module 3	Disease Prediction Model
Module 4	Model Evaluation
Module 5	Documentation

Module 1: Data Collection and Preprocessing

Objective

In liver cirrhosis, there are mainly 4 stages which are as follows:

a. Stage 1 : Normal
b. Stage 2 : Fatty Liver
c. Stage 3 : Liver Fibrosis
d. Stage 4 : Liver Cirrhosis

The primary target is to predict the stage of the liver cirrhosis disease. The dataset consists of both numerical as well as categorical features.

Data Sources:
- Collect relevant healthcare data from sources such as electronic health records (EHRs), medical databases, or public health datasets.

Dataset: cirrhosis.csv

Context:

Cirrhosis is a late stage of scarring (fibrosis) of the liver caused by many forms of liver diseases and conditions, such as hepatitis and chronic alcoholism. The following data contains the information collected from the Mayo Clinic trial in primary biliary cirrhosis (PBC) of the liver conducted between 1974 and 1984. A description of the clinical background for the trial and the covariates recorded here is in Chapter 0, especially Section 0.2 of Fleming and Harrington, Counting Processes and Survival Analysis, Wiley, 1991. A more extended discussion can be found in Dickson, et al., Hepatology 10:1-7 (1989) and in Markus, et al., N Eng J of Med 320:1709-13 (1989).

A total of 424 PBC patients, referred to Mayo Clinic during that ten-year interval, met eligibility criteria for the randomized placebo-controlled trial of the drug D-penicillamine. The first 312 cases in the dataset participated in the randomized trial and contain largely complete data. The additional 112 cases did not participate in the clinical trial but consented to have basic measurements recorded and to be followed for survival. Six of those cases were lost to follow-up shortly after diagnosis, so the data here are on an additional 106 cases as well as the 312 randomized participants.

Attribute Information

1) ID: unique identifier
2) N_Days: number of days between registration and the earlier of death, transplantation, or study analysis time in July 1986
3) Status: status of the patient C (censored), CL (censored due to liver tx), or D (death)
4) Drug: type of drug D-penicillamine or placebo
5) Age: age in [days]
6) Sex: M (male) or F (female)
7) Ascites: presence of ascites N (No) or Y (Yes)
8) Hepatomegaly: presence of hepatomegaly N (No) or Y (Yes)
9) Spiders: presence of spiders N (No) or Y (Yes)
10) Edema: presence of edema N (no edema and no diuretic therapy for edema), S (edema present without diuretics, or edema resolved by diuretics), or Y (edema despite diuretic therapy)
11) Bilirubin: serum bilirubin in [mg/dl]
12) Cholesterol: serum cholesterol in [mg/dl]
13) Albumin: albumin in [gm/dl]
14) Copper: urine copper in [ug/day]
15) Alk_Phos: alkaline phosphatase in [U/liter]
16) SGOT: SGOT in [U/ml]
17) Triglycerides: triglicerides in [mg/dl]
18) Platelets: platelets per cubic [ml/1000]
19) Prothrombin: prothrombin time in seconds [s]
20) Stage: histologic stage of disease (1, 2, 3, or 4)

Data Cleaning:

Handle missing values:

Missing values are common in healthcare data due to various reasons such as data entry errors, equipment malfunction, or patients skipping certain tests. It's crucial to handle missing values appropriately as they can significantly impact the analysis. Techniques for handling missing values include:

- Imputation: Replace missing values with a calculated estimate such as mean, median, or mode of the corresponding feature.
- Deletion: Remove records or features with a high percentage of missing values if they don't hold significant importance.
- Predictive modeling: Use machine learning algorithms to predict missing values based on other features.

Handle outliers:

Outliers are data points that significantly deviate from the rest of the data. In healthcare data, outliers can occur due to measurement errors, extreme cases, or anomalies. It's essential to identify and address outliers appropriately:

- Visual inspection: Plotting histograms, box plots, or scatter plots can help identify outliers visually.
- Statistical methods: Techniques such as z-score, interquartile range (IQR), or Tukey's method can be used to detect outliers statistically.
- Transformation: Transforming skewed data using techniques like logarithmic transformation can mitigate the impact of outliers.
- Trimming or winsorization: Assigning extreme values to the nearest non-outlier values or removing extreme values beyond a certain threshold.

Inconsistencies:

In healthcare data, inconsistencies can arise due to variations in data entry methods, coding errors, or inconsistencies in recording formats. To address inconsistencies:

- Standardize data formats: Ensure consistency in date formats, coding systems (e.g., ICD-10 for diagnoses), and units of measurement.
- Data validation: Implement validation checks to identify and correct inconsistencies in the data.
- Data profiling: Analyze the data to identify patterns and inconsistencies, then correct them manually or programmatically.

Feature Engineering:

Patient demographics:
Demographic features such as age, gender, ethnicity, socioeconomic status, and geographic location can provide valuable insights into disease prediction. These features can be directly extracted from patient records or derived from additional sources.

Medical history:
Past medical history including pre-existing conditions, previous treatments, surgeries, medications, and lifestyle factors play a crucial role in disease prediction. Creating features that capture the presence, severity, and duration of medical conditions can enhance predictive models.

Diagnostic test results:
Results from diagnostic tests such as blood tests, imaging studies (e.g., X-rays, MRI), and genetic tests contain valuable information for disease prediction. Feature engineering involves transforming raw test results into meaningful features such as abnormality scores, biomarker levels, or disease risk scores.

Temporal features:
Time-related features such as frequency of hospital visits, time since diagnosis, or temporal patterns in symptoms can provide valuable insights into disease progression and prognosis.

Composite features:
Combining multiple related features or creating interaction terms can capture complex relationships and improve predictive performance. For example, creating a feature that combines age and comorbidity status to assess overall disease risk.

Dimensionality reduction:
Techniques such as principal component analysis (PCA) or feature selection methods can be used to reduce the dimensionality of the feature space while retaining relevant information and improving model efficiency.

By performing effective data cleaning and feature engineering, healthcare data can be transformed into a high-quality dataset suitable for building accurate and reliable predictive models for disease diagnosis, prognosis, and treatment planning.

Module 2. Exploratory Data Analysis (EDA):

Descriptive Statistics:

Calculate descriptive statistics:

Descriptive statistics are used to summarize and describe the main features of a dataset. This includes measures of central tendency (mean, median, mode) and measures of variability (range, variance, standard deviation). The purpose of calculating descriptive statistics is to gain insights into the distribution of variables, identify outliers, and understand the overall characteristics of the data.

Mean:

The average value of the data, calculated by summing all values and dividing by the total number of observations.

Median:

The middle value of the data when arranged in ascending order. It is less sensitive to outliers compared to the mean.

Mode:

The value that appears most frequently in the dataset.

Range:

The difference between the maximum and minimum values in the dataset, providing a measure of the spread of the data.

Variance:

A measure of the dispersion of values around the mean. It indicates how much the values deviate from the mean.

Standard deviation:

The square root of the variance, representing the average deviation of each data point from the mean. It provides a measure of the spread of the data.

Calculating these descriptive statistics helps in understanding the central tendency, variability, and shape of the distribution of variables in the dataset, which are crucial for further analysis and interpretation.

Data Visualization:

Visualize relationships between different features:

Data visualization involves creating graphical representations of data to explore relationships, patterns, and trends. Charts and graphs are effective tools for visualizing relationships between different features in a dataset.

Scatter plots:

Scatter plots are used to visualize the relationship between two continuous variables. Each data point is represented as a point on the graph, with one variable on the x-axis and the other variable on the y-axis. Scatter plots help identify patterns, trends, and correlations between variables.

Bar charts:

Bar charts are useful for comparing the frequency or distribution of categorical variables. They consist of vertical or horizontal bars representing the frequency or proportion of each category.

Line charts:

Line charts are commonly used to visualize trends over time or across different categories. They connect data points with lines, making it easy to see changes and patterns over the specified time period.

Explore patterns and correlations in the data:

Data visualization also helps in exploring patterns and correlations between variables in the dataset. By plotting different combinations of variables and visually inspecting the resulting graphs, analysts can identify correlations, clusters, outliers, and other patterns in the data. Heatmaps, box plots, histograms, and pair plots are some additional visualization techniques that can be used to explore and understand the relationships within the data.

In summary, descriptive statistics provide quantitative summaries of the data, while data visualization offers visual insights into relationships, patterns, and trends. Together, these techniques facilitate a comprehensive understanding of the dataset and inform subsequent data analysis and decision-making processes.

Module 3. Disease Prediction Model:

Selection of Algorithms:

Choose appropriate machine learning algorithms:

Selecting the right machine learning algorithm is crucial for disease prediction tasks. Different algorithms have different strengths and weaknesses, and the choice depends on factors such as the nature of the data, the complexity of the problem, and the interpretability of the model. Some common algorithms used for disease prediction include:

Logistic Regression:

Suitable for binary classification tasks, logistic regression models the probability of the outcome variable based on one or more predictor variables.

Decision Trees:

Decision trees recursively partition the feature space into regions, making them interpretable and suitable for both classification and regression tasks.

Random Forests:

Random forests are an ensemble learning method that builds multiple decision trees and combines their predictions to improve accuracy and robustness.

Deep Learning Models:

Deep learning models such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) can learn complex patterns from raw data and are particularly useful when dealing with large and high-dimensional datasets.

Target Variable Definition:

Define the target variable:

The target variable in disease prediction tasks typically represents the outcome of interest, such as the presence or absence of a disease. Depending on the specific problem, the target variable can be defined as:

Binary classification:

For diseases with two possible outcomes (e.g., presence or absence), the target variable is binary, with values indicating the positive or negative occurrence of the disease.

Multiclass classification:

For diseases with multiple categories or subtypes, the target variable is multiclass, with values representing different disease categories.

Training Data Split:

Split the dataset:
Before training a machine learning model, it's essential to split the dataset into training and testing sets to evaluate the model's performance accurately. The dataset is typically divided into two subsets:
- **Training set:** Used to train the model on a portion of the data.
- **Testing set:** Used to evaluate the model's performance on unseen data and assess its generalization ability.

Additionally, techniques such as cross-validation can be used to further validate the model and mitigate issues such as overfitting.

Model Training:

Train the selected model:
Once the dataset is split, the selected machine learning model is trained on the training set. During the training process, the model learns the underlying patterns and relationships between the input features and the target variable. The training algorithm adjusts the model parameters to minimize the prediction error or maximize the likelihood of the observed data.

Hyperparameter Tuning:

Optimize model hyperparameters:
Hyperparameters are parameters that control the learning process and affect the performance of the model. Hyperparameter tuning involves selecting the optimal values for these parameters to improve the model's predictive performance. Techniques for hyperparameter tuning include:
- **Grid search:** Exhaustively search through a specified set of hyperparameters to identify the combination that yields the best performance.
- **Random search:** Randomly sample hyperparameter values from predefined distributions and evaluate their performance.
- **Bayesian optimization:** Use probabilistic models to efficiently search for the optimal hyperparameters based on past evaluations.

Hyperparameter tuning is essential for optimizing model performance and achieving the best possible results on unseen data.

In summary, the selection of appropriate algorithms, definition of the target variable, data splitting, model training, and hyperparameter tuning are essential steps in developing a disease prediction model. These steps ensure that the model is trained on relevant data, performs well on unseen data, and generalizes effectively to new cases.

Module 4. Model Evaluation:

Performance Metrics:

Evaluate the model using metrics:

Accuracy:
Accuracy measures the proportion of correctly classified instances out of the total instances. It is calculated as the ratio of the number of correct predictions to the total number of predictions.

Precision:
Precision measures the proportion of true positive predictions out of all positive predictions made by the model. It is calculated as the ratio of true positives to the sum of true positives and false positives.

Recall (Sensitivity):
Recall measures the proportion of true positive predictions out of all actual positive instances in the dataset. It is calculated as the ratio of true positives to the sum of true positives and false negatives.

F1-score:
The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall and is calculated as,

(2 * (precision * recall) / (precision + recall))

Area Under the ROC Curve (AUC-ROC):
The ROC curve is a graphical plot that illustrates the performance of a binary classification model across different threshold settings. AUC-ROC quantifies the model's ability to distinguish between positive and negative instances, with higher values indicating better performance. It ranges from 0 to 1, where 0.5 represents random guessing and 1 represents perfect classification.

Confusion Matrix:

Analyze the confusion matrix:
A confusion matrix is a tabular representation of the actual vs. predicted classifications made by a classification model. It is particularly useful for evaluating the performance of binary classification models and understanding different types of prediction errors.
- The confusion matrix consists of four main elements:
  - True Positives (TP): Instances correctly classified as positive.
  - True Negatives (TN): Instances correctly classified as negative.
  - False Positives (FP): Instances incorrectly classified as positive (Type I error).
  - False Negatives (FN): Instances incorrectly classified as negative (Type II error).
- By analyzing the confusion matrix, you can calculate performance metrics such as accuracy, precision, recall, and F1-score, and gain insights into the model's strengths and weaknesses.

Cross-Validation:

Implement cross-validation:
- Cross-validation is a resampling technique used to assess a model's performance, stability, and generalizability by splitting the dataset into multiple subsets and iteratively training and evaluating the model on different combinations of these subsets.
- The most common type of cross-validation is k-fold cross-validation, where the dataset is divided into k equal-sized folds. The model is trained on k-1 folds and evaluated on the remaining fold, repeating this process k times with each fold serving as the test set exactly once.
- Cross-validation helps mitigate issues such as overfitting and provides more reliable estimates of the model's performance by averaging results across multiple iterations.
- Other variations of cross-validation include stratified cross-validation, leave-one-out cross-validation, and nested cross-validation, each with its own advantages and use cases.

In summary, performance metrics, confusion matrix analysis, and cross-validation are essential techniques for evaluating and validating machine learning models, providing insights into their accuracy, reliability, and generalizability. These techniques help assess model performance comprehensively and guide further refinement and optimization efforts.

Module 5: Documentation:

Code Documentation:

Document the codebase:
This involves providing comments and explanations within the code to clarify its functionality, structure, and purpose. Here are some key aspects to consider when documenting the codebase:
- Function and method documentation:
Include comments before each function or method describing its purpose, input parameters, return values, and any side effects.
- Variable naming:
Use descriptive variable names that convey their meaning and purpose. Avoid cryptic abbreviations or single-letter variable names.

- Code structure:
Document the overall structure of the codebase, including modules, classes, and their relationships.
- Data preprocessing steps:
Document each step of the data preprocessing pipeline, including handling missing values, feature engineering, normalization, and encoding categorical variables.

Algorithms used:

Provide explanations of the algorithms implemented in the codebase, including references to relevant literature or resources for further reading. Describe the underlying principles, assumptions, and limitations of each algorithm.

Model Documentation:

Provide documentation for the model architecture:

Describe the architecture of the machine learning model, including its layers, connections, and parameters. For deep learning models, this may involve providing a high-level overview of the neural network architecture, including the types of layers used (e.g., convolutional, recurrent) and their configurations.

Hyperparameters:

Document the hyperparameters used in the model, including their names, values, and purposes. Hyperparameters control aspects of the model such as its complexity, regularization, and optimization strategy. Common hyperparameters include learning rate, batch size, dropout rate, and the number of layers or units in a neural network.

Training process:

Describe the training process used to train the model, including the loss function, optimization algorithm, and any regularization techniques employed. Explain how the model was trained on the training data, evaluated on the validation data, and fine-tuned using techniques such as early stopping or learning rate scheduling.

Evaluation metrics:

Document the evaluation metrics used to assess the performance of the model, including accuracy, precision, recall, F1-score, and any domain-specific metrics relevant to the problem at hand. Provide interpretations of these metrics and their implications for model performance.

By thoroughly documenting the codebase, algorithms, model architecture, hyperparameters, and training process, you ensure that the model is transparent, reproducible, and understandable to other developers and stakeholders.

Project Code Part:

Prerequisite Packages:

NumPy

- NumPy is a fundamental package 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 is widely used in various fields such as data science, machine learning, engineering, and scientific research.
- It offers functionalities for numerical computations, linear algebra operations, Fourier transforms, random number generation, and more.

Pandas

- Pandas is a powerful and flexible open-source data analysis and manipulation library built on top of NumPy.
- It provides data structures like DataFrame and Series, which allow easy handling and manipulation of structured data.
- Pandas is extensively used for data cleaning, data exploration, data wrangling, and data preparation tasks in data science projects.
- It offers functionalities for reading and writing data from various file formats (e.g., CSV, Excel, SQL databases), handling missing data, reshaping and pivoting data, grouping and aggregating data, and time series analysis.

Seaborn

- Seaborn is a Python data visualization library based on matplotlib.
- It provides a high-level interface for creating attractive and informative statistical graphics.
- Seaborn simplifies the process of creating complex visualizations by providing easy-to-use functions for common visualization tasks.
- It offers support for visualizing univariate and bivariate distributions, categorical data, linear and non-linear relationships, time series data, and more.
- Seaborn is often used in conjunction with Pandas for visualizing data stored in DataFrame objects.

matplotlib.pyplot

- Matplotlib is a comprehensive library for creating static, interactive, and animated visualizations in Python.
- `matplotlib.pyplot` is a collection of functions within Matplotlib that provides a MATLAB-like plotting interface.
- It allows you to create various types of plots such as line plots, scatter plots, bar plots, histograms, heatmaps, and more.
- `pyplot` provides fine-grained control over the appearance of plots, including axis labels, titles, legends, colors, and styles.
- Matplotlib is a foundational library for data visualization in Python and is often used alongside other libraries like NumPy and Pandas.

matplotlib inline

- `%matplotlib inline` is a magic command in Jupyter Notebooks (and some other interactive environments) that enables the inline display of Matplotlib plots directly within the notebook.
- When you include `%matplotlib inline` at the beginning of a notebook cell or at the top of a notebook, it instructs the notebook to render Matplotlib plots as static images within the notebook output.
- This command is particularly useful for quickly visualizing data and iterating on plots within the notebook environment without needing to save the plots to files or use additional display functions.

warnings.filterwarnings('ignore')

- `warnings.filterwarnings('ignore')` is a Python command used to suppress the display of warning messages during program execution.
- Warnings are messages that indicate potential issues or non-critical problems in the code, such as deprecated features, future changes, or data inconsistencies.
- By calling `warnings.filterwarnings('ignore')`, you are instructing Python to ignore all warning messages generated during the execution of your code.
- This can be useful when you are confident that certain warnings can be safely ignored or when you want to avoid cluttering the output with non-essential messages. However, it's important to use this command judiciously, as ignoring warnings indiscriminately may mask important information about potential issues in your code.

train_test_split

- `train_test_split` is a function provided by the `sklearn.model_selection` module in scikit-learn.
- It is used for splitting datasets into two subsets: one for training the model and the other for testing the model.
- The function randomly shuffles the data and splits it into a training set and a testing set based on a specified ratio (usually, the training set is larger).
- The training set is used to train the machine learning model, while the testing set is used to evaluate its performance on unseen data.
- Common parameters of `train_test_split` include `test_size` (the proportion of the dataset to include in the test split) and `random_state` (to ensure reproducibility by fixing the random seed).

GridSearchCV

- `GridSearchCV` is a class provided by the `sklearn.model_selection` module in scikit-learn.
- It is used for systematically searching for the optimal hyperparameters of a machine learning model.
- Grid search involves specifying a grid of hyperparameters and evaluating the performance of the model trained with each combination of hyperparameters using cross-validation.
- The class takes an estimator (model), a dictionary of hyperparameters, and a cross-validation strategy as input.
- After fitting the `GridSearchCV` object to the training data, it identifies the best combination of hyperparameters based on a specified scoring metric (e.g., accuracy, precision, recall, F1-score).
- `GridSearchCV` performs an exhaustive search over all possible combinations of hyperparameters, making it computationally expensive for large parameter grids.

cross_val_score

- `cross_val_score` is a function provided by the `sklearn.model_selection` module in scikit-learn.
- It is used for evaluating a machine learning model's performance using cross-validation.
- Cross-validation is a technique for assessing how well a model generalizes to unseen data by partitioning the dataset into multiple subsets (folds).

- The function takes an estimator (model), input features, target labels, and optionally a cross-validation strategy and returns an array of scores obtained for each fold.
- By default, `cross_val_score` uses stratified k-fold cross-validation for classification and k-fold cross-validation for regression.
- The average score across all folds provides an estimate of the model's performance, and the variability in scores indicates its stability and reliability.

cross_validate
- `cross_validate` is a function provided by the `sklearn.model_selection` module in scikit-learn.
- Similar to `cross_val_score`, it is used for evaluating a machine learning model's performance using cross-validation.
- However, `cross_validate` provides more flexibility by allowing the computation of multiple evaluation metrics and returning additional information such as training scores, fit times, and score times.
- It returns a dictionary containing the evaluation scores and other performance-related metrics for each fold.
- This function is useful when you need more detailed information about the model's performance, such as training and testing times, or when you want to compute multiple evaluation metrics simultaneously.

RepeatedStratifiedKFold
- `RepeatedStratifiedKFold` is a cross-validation iterator provided by the `sklearn.model_selection` module in scikit-learn.
- It is used for performing repeated stratified k-fold cross-validation, which is particularly useful for classification tasks with imbalanced class distributions.
- Stratified k-fold cross-validation ensures that each fold contains approximately the same proportion of samples from each class as the entire dataset.
- RepeatedStratifiedKFold repeats the stratified k-fold cross-validation process multiple times with different randomizations of the data, hence providing more robust estimates of the model's performance.
- It takes parameters such as the number of folds (`n_splits`), the number of repetitions (`n_repeats`), and a random seed (`random_state`) for reproducibility.
- This cross-validation strategy is suitable when you want to obtain more reliable estimates of the model's performance, especially in the presence of class imbalance or when dealing with small datasets.

LogisticRegression
- Logistic Regression is a popular classification algorithm used for binary classification tasks.
- Despite its name, it is a linear model used for estimating probabilities of the binary outcome based on one or more predictor variables.
- It models the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using the logistic function.
- Logistic Regression is widely used due to its simplicity, interpretability, and efficiency.

RidgeClassifier

- RidgeClassifier is a linear classifier that uses Ridge Regression for classification tasks.
- Similar to Logistic Regression, it's primarily used for binary classification.
- Ridge Regression adds a penalty term (L2 regularization) to the linear regression cost function, which helps to prevent overfitting by shrinking the coefficients towards zero.
- RidgeClassifier is useful when dealing with multicollinearity in the input features, as it can handle situations where the independent variables are highly correlated.

PassiveAggressiveClassifier
- Passive-Aggressive algorithms are a family of online learning algorithms, primarily used for binary classification.
- They are particularly useful when dealing with large-scale and streaming data because they update the model parameters incrementally, one instance at a time, instead of using entire batches of data.
- The "passive" part of the name refers to the fact that the algorithm does not update its parameters when predictions are correct but rather updates them aggressively (hence the "aggressive" part) when predictions are incorrect.
- PassiveAggressiveClassifier is suitable for scenarios where the data distribution may change over time or where computational resources are limited.

SGDClassifier
- Stochastic Gradient Descent (SGD) Classifier is a linear classifier that uses stochastic gradient descent optimization to learn the model parameters.
- It's a versatile algorithm capable of performing linear classification, regression, and other machine learning tasks with sparse data.
- SGDClassifier updates the model parameters iteratively based on the gradient of the loss function computed on a single sample or a small batch of samples, making it efficient for large-scale datasets.
- It's often used in scenarios where traditional batch learning algorithms are impractical due to computational constraints or when dealing with streaming data.

The K-Nearest Neighbors (KNN) Classifier is a simple and effective supervised machine learning algorithm used for classification tasks. It's a non-parametric method, meaning it doesn't make any assumptions about the underlying data distribution.

Algorithm:
- Given a dataset with labeled instances, KNN classifies a new instance by assigning it the majority class label among its K nearest neighbors.
- The "nearest" neighbors are determined based on a distance metric, often Euclidean distance, although other distance metrics like Manhattan distance, cosine similarity, or Minkowski distance can also be used.
- KNN doesn't involve a training phase like many other classifiers. Instead, it memorizes the training instances and their labels to use them during the classification phase.

Key Parameters:
- n_neighbors: Specifies the number of neighbors to consider when making predictions. It's a hyperparameter that needs to be tuned, typically through techniques like cross-validation.

- metric: Defines the distance metric used to measure the distance between instances. Common options include Euclidean distance, Manhattan distance, and cosine similarity.
- weights: Determines how the contributions of the neighbors are weighted when making predictions. Options include uniform weights (where all neighbors have equal influence) and distance weights (where closer neighbors have more influence).

Advantages:
- Simple and intuitive algorithm, easy to understand and implement.
- Non-parametric nature allows it to handle complex decision boundaries and nonlinear relationships between features and labels.
- KNN can be effective for both binary and multiclass classification tasks.
- It doesn't require a training phase, making it suitable for online learning and incremental updates.

Limitations:
- KNN can be computationally expensive, especially with large datasets or high-dimensional feature spaces, as it requires calculating distances between the new instance and all training instances.
- The choice of K affects the algorithm's performance. A smaller K value can lead to noisy predictions, while a larger K value can lead to oversmoothed decision boundaries.
- KNN is sensitive to the scale of the features, so feature scaling is often necessary to ensure all features contribute equally to the distance calculation.
- The algorithm may struggle with imbalanced datasets or datasets with irrelevant or noisy features.

Use Cases:
- KNN is commonly used in applications such as recommendation systems, where it's used to find similar items or users based on their features or preferences.
- It's also used in image recognition, text categorization, and other domains where the data can be represented as points in a high-dimensional space.
- KNN can serve as a baseline model for comparison with more complex classifiers, or in scenarios where interpretability and simplicity are prioritized over predictive performance.

RandomForestClassifier

- Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (classification) or the mean prediction (regression) of the individual trees.
- It builds each tree independently using a random subset of the training data and a random subset of features, which helps to reduce overfitting and increase robustness.
- Random Forest is known for its high accuracy, scalability, and ability to handle high-dimensional datasets with mixed feature types.
- It's suitable for both classification and regression tasks and is often used as a baseline model due to its simplicity and effectiveness.

AdaBoostClassifier

- AdaBoost (Adaptive Boosting) is an ensemble learning method that combines multiple weak learners (usually decision trees) to create a strong learner.
- It sequentially trains a series of weak learners, with each subsequent learner focusing on the instances that were misclassified by the previous learners.
- AdaBoost assigns weights to the training instances based on their classification accuracy, with more weight given to misclassified instances, thereby emphasizing the difficult-to-classify instances.
- It's particularly effective in binary classification tasks and can be sensitive to noisy data and outliers.

GradientBoostingClassifier
- Gradient Boosting is another ensemble learning method that builds an ensemble of weak learners (typically decision trees) in a sequential manner.
- Unlike AdaBoost, which adjusts the instance weights, Gradient Boosting fits each subsequent learner to the residual errors made by the previous learners, gradually reducing the error during training.
- Gradient Boosting optimizes a differentiable loss function using gradient descent, resulting in a more flexible and powerful ensemble model.
- It's highly customizable, allowing users to specify various hyperparameters such as the number of trees, tree depth, and learning rate.
- Gradient Boosting is widely used in both classification and regression tasks and is known for its high predictive accuracy.

HistGradientBoostingClassifier
- HistGradientBoosting is a histogram-based implementation of Gradient Boosting specifically optimized for large datasets.
- It uses histograms to approximate the gradients and Hessians, making it more memory-efficient and faster than traditional Gradient Boosting methods.
- HistGradientBoostingClassifier is particularly suitable for datasets with a large number of samples and features, as well as categorical features.
- It offers similar flexibility and customization options as GradientBoostingClassifier but with improved scalability and performance.

BaggingClassifier
- Bagging (Bootstrap Aggregating) is an ensemble learning method that creates multiple subsets of the training data by sampling with replacement (bootstrap samples) and trains a base learner on each subset.
- The predictions of the base learners are then aggregated (e.g., by averaging for regression or by voting for classification) to make the final prediction.
- BaggingClassifier can use any base estimator, such as decision trees, and is effective at reducing variance and improving generalization by creating diverse models.
- It's particularly useful when the base estimator is unstable or prone to overfitting, as bagging helps to smooth out the predictions.

ExtraTreesClassifier

- Extra Trees (Extremely Randomized Trees) is an ensemble learning method that builds multiple decision trees using random subsets of the features and random thresholds for splitting.
- Unlike Random Forest, which selects the optimal feature and threshold for each split, Extra Trees randomly selects features and thresholds, leading to a higher level of randomness and diversity among the trees.
- ExtraTreesClassifier is computationally efficient and less prone to overfitting than traditional decision trees or Random Forest, making it suitable for large datasets with high dimensionality.

VotingClassifier

- Voting is an ensemble learning method that combines the predictions of multiple base estimators (classifiers or regressors) and outputs the majority vote (classification) or the average prediction (regression) as the final prediction.
- VotingClassifier supports different voting strategies, including hard voting (simple majority) and soft voting (weighted average of probabilities).
- It can combine different types of base estimators, such as decision trees, support vector machines, logistic regression, etc., to create a diverse ensemble model.
- VotingClassifier can often achieve better generalization and robustness than individual base estimators, especially when the base estimators have different strengths and weaknesses.

DecisionTreeClassifier

- Decision Tree is a versatile supervised learning algorithm used for classification and regression tasks.
- It partitions the feature space into regions and predicts the target variable based on the majority class (classification) or the mean value (regression) of the instances within each region.
- DecisionTreeClassifier recursively splits the data based on feature values to minimize impurity (e.g., Gini impurity or entropy), resulting in a tree-like structure of decisions.
- Decision trees are intuitive, easy to interpret, and capable of handling both numerical and categorical data.

SVC (Support Vector Classifier)

- Support Vector Machine (SVM) is a powerful supervised learning algorithm used for classification and regression tasks.
- SVC finds the optimal hyperplane that separates the classes in the feature space by maximizing the margin between the classes.
- It can handle linear and nonlinear classification tasks using different kernel functions (e.g., linear, polynomial, radial basis function).
- SVC is effective in high-dimensional spaces and is particularly useful when the number of features is greater than the number of samples.

StandardScaler

- StandardScaler is a preprocessing technique used for feature scaling in machine learning.
- It standardizes the features by removing the mean and scaling to unit variance, resulting in a mean of 0 and a standard deviation of 1 for each feature.
- StandardScaler ensures that all features are on the same scale, which is important for many machine learning algorithms that are sensitive to the scale of the features (e.g., SVM, KNN).

- It's applied independently to each feature and does not assume any specific distribution of the data.

LGBMClassifier (LightGBM Classifier)
- LightGBM is a gradient boosting framework that's known for its efficiency, speed, and high performance.
- LGBMClassifier implements the LightGBM algorithm for classification tasks.
- It uses a tree-based learning algorithm that grows trees vertically (leaf-wise) rather than horizontally (level-wise), resulting in faster training times and lower memory usage.
- LightGBM supports categorical features without requiring one-hot encoding, making it efficient for datasets with a mixture of numerical and categorical features.

CatBoostClassifier
- CatBoost is another gradient boosting framework designed for categorical data and high-dimensional feature spaces.
- CatBoostClassifier implements the CatBoost algorithm for classification tasks.
- It uses an innovative algorithm for handling categorical features, including the use of ordered boosting, which reduces overfitting and improves generalization.
- CatBoost automatically handles missing values and provides robustness to noisy data.

XGBClassifier (XGBoost Classifier)
- XGBoost (Extreme Gradient Boosting) is an optimized gradient boosting library known for its scalability, speed, and performance.
- XGBClassifier implements the XGBoost algorithm for classification tasks.
- It uses a regularized gradient boosting framework that combines the advantages of both gradient boosting and regularization techniques to prevent overfitting and improve model generalization.
- XGBoost supports various objective functions, customizable loss functions, and advanced features like early stopping to prevent overfitting.

GaussianNB (Gaussian Naive Bayes)
- Gaussian Naive Bayes is a probabilistic classifier based on Bayes' theorem and the assumption of feature independence.
- It's particularly suitable for classification tasks with continuous or normally distributed features.
- GaussianNB models the likelihood of each class using Gaussian distributions and calculates the posterior probability of each class given the features.
- Despite its simplicity and the "naive" assumption of feature independence, GaussianNB can perform well on many real-world datasets, especially when the independence assumption approximately holds.

BernoulliNB (Bernoulli Naive Bayes)
- Bernoulli Naive Bayes is another variant of Naive Bayes specifically designed for binary feature data.
- It assumes that features are binary (e.g., presence or absence of a feature) and models the likelihood of each class using Bernoulli distributions.

- BernoulliNB is commonly used for text classification tasks, where features represent the presence or absence of words in documents (bag-of-words model).
- Like GaussianNB, BernoulliNB is simple, efficient, and performs well on text classification tasks with sparse binary feature vectors.

Each of these classifiers and preprocessing techniques has its strengths and weaknesses, and the choice depends on factors such as the nature of the data, the complexity of the problem, and computational considerations. Experimentation and validation on your specific dataset are essential to determine the most suitable approach.

joblib
- Joblib is a library in Python that provides utilities for saving and loading Python objects (such as NumPy arrays, Scikit-learn estimators, etc.) to and from disk.
- It's particularly useful for efficiently storing large objects, especially those generated during machine learning model training.
- Joblib's `dump` and `load` functions are commonly used for serialization and deserialization of objects, offering better performance compared to Python's built-in `pickle` module.

scipy
- SciPy is an open-source scientific computing library in Python that builds on top of NumPy.
- It provides a wide range of mathematical functions, numerical routines, and optimization algorithms for scientific and engineering applications.
- SciPy includes submodules for linear algebra, optimization, interpolation, integration, signal processing, statistics, and more.
- It's widely used in various fields such as physics, engineering, biology, and finance for tasks such as data analysis, simulation, and modeling.

sklearn
- Scikit-learn, also known as sklearn, is a popular machine learning library in Python.
- It provides simple and efficient tools for data mining and data analysis, built on top of NumPy, SciPy, and matplotlib.
- Scikit-learn offers a wide range of supervised and unsupervised learning algorithms, including classification, regression, clustering, dimensionality reduction, and model selection.
- It also provides utilities for data preprocessing, model evaluation, and model tuning, making it a comprehensive toolkit for machine learning tasks.

SMOTE
- Synthetic Minority Over-sampling Technique (SMOTE) is an oversampling technique used to address class imbalance in binary classification tasks.
- It works by generating synthetic samples of the minority class by interpolating between existing minority class instances.
- SMOTE helps to balance the class distribution in the training data, which can improve the performance of machine learning models, particularly those sensitive to class imbalance.
- Scikit-learn provides an implementation of SMOTE through the `imbalanced-learn` package, which can be used in conjunction with other preprocessing techniques.

Counter

- Counter is a container in Python's collections module that is used to count the occurrences of elements in an iterable or a collection (such as a list, tuple, or string).
- It's particularly useful for tasks such as frequency counting, finding the most common elements, and identifying unique elements in a dataset.
- Counter objects behave like dictionaries, with elements as keys and their counts as values, but they also provide additional methods for operations like addition, subtraction, intersection, and union of counts.

classification_report

- classification_report is a function in scikit-learn that generates a text report showing the main classification metrics for a classification model.
- It calculates metrics such as precision, recall, F1-score, and support for each class in the classification problem.
- The report also includes macro-average and weighted-average metrics, which provide overall performance summaries across all classes.
- classification_report is useful for evaluating the performance of a classifier and understanding its strengths and weaknesses for different classes.

confusion_matrix

- confusion_matrix is a function in scikit-learn that computes a confusion matrix for evaluating the performance of a classification model.
- A confusion matrix is a table that summarizes the predictions made by a classifier compared to the actual class labels in the test data.
- It provides insights into the classifier's true positive, false positive, true negative, and false negative predictions for each class, which can help identify areas of improvement in the model.
- The confusion matrix is often used in combination with other evaluation metrics to assess the overall performance of a classifier.

plot_confusion_matrix

- plot_confusion_matrix is a function in scikit-learn that plots a confusion matrix as a heatmap for visualizing the performance of a classification model.
- It provides a more intuitive and visually appealing representation of the confusion matrix compared to the tabular format.
- The heatmap colors cells based on the number of instances in each combination of predicted and actual classes, making it easier to identify patterns and discrepancies in the classifier's predictions.
- plot_confusion_matrix is useful for communicating the performance of a classifier to stakeholders and decision-makers in a more understandable format.

Step by Step Code Execution:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
from sklearn.model_selection import train_test_split, GridSearchCV,
cross_val_score, cross_validate, RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression, RidgeClassifier,
PassiveAggressiveClassifier, SGDClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier,
GradientBoostingClassifier, HistGradientBoostingClassifier, BaggingClassifier,
ExtraTreesClassifier, VotingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier
from xgboost import XGBClassifier
from sklearn.naive_bayes import GaussianNB, BernoulliNB
import joblib, scipy, sklearn
from imblearn.over_sampling import SMOTE
from collections import Counter
from sklearn.metrics import classification_report, confusion_matrix,
plot_confusion_matrix
```

```
df = pd.read_csv("D:/cirrhosis.csv")

df.head()
```

```
df.drop(['ID'],axis=1,inplace=True)
df.shape
```

```
df.info()
```

```
df.describe()
```

```
#Checking for percentage of null data in each column

df.isnull().sum() / len(df) * 100
```

```
#Checking for duplicate data


df[df.duplicated()]
```

```
#Exploratory Data Analysis


for col in df.columns:
    if df[col].dtypes != 'object':
        if col not in ['Stage']:
            print(col.title())
            print("Skewness:",df[col].skew())
            print("Kurtosis:",df[col].kurtosis())
            plt.figure(figsize=(8,8))
            sns.distplot(df[col])
            plt.show()
            sns.boxplot(df[col])
            plt.show()
            scipy.stats.probplot(df[col],plot=plt,rvalue=True)
            plt.show()
```

Analysis:
The distributions of the features Bilirubin, Cholesterol, Copper, Alkaline Phosphatase, SGOT, Tryglicerides and Prothrombin are highly skewed and their kurtosis values are more significant as well. As a result, they are more prone to having outliers which is clearly indicated by their respective box plots.

```
def pie_chart(df,col):
   labels = df[col].value_counts().keys()
   values = df[col].value_counts().values
   explode = [0]*np.size(labels)
   explode[0] = 0.2
   explode[1] = 0.1
   plt.figure(figsize=(8,8))

plt.pie(values,labels=labels,explode=explode,shadow=True,autopct='%1.2f%%')
   plt.show()

pie_chart(df,'Stage')
```

Analysis:
Majority of the patients had third histologic stage of Cirrhosis disease accounting for almost 38% of the total share, closely followed by those having 4th stage which comprised a share of about 35%. The proportion of the number of patients having 2nd stage of Cirrhosis disease stood at a little more than one-fifth of the total share. However, a tiny fraction of the total share was occupied by patients having the first histologic stage of the disease.

```
pie_chart(df,'Status')
```

Analysis:
A significant proportion of the entire population of patients had "Censored" status, contributing to a little more than half of the total share. They are followed by the patients who died during the course of the clinical trials, possessing a share of just less than 40%. Patients, having the status of "Censored due to Liver tx", had a minimal share of just over 5%.

```
pie_chart(df,'Sex')
```

Analysis:
A vast majority of the entire population of patients are females, contributing to a substantial share of almost 90%. Males comprised a little more than one-tenth of the total share.

```
pie_chart(df,'Drug')
```

Analysis:
Both the drugs, D-penicillamine and Placebo, recorded an equal amount of usage during the clinical trials.

```
pie_chart(df,'Ascites')
```

Analysis:

Only a tiny fraction of the population of patients had Ascites disease.

```
pie_chart(df,'Hepatomegaly')
```

Analysis:

The percentage share of patients suffering from the Hepatomegaly disease and those who did not have this disease is almost similar.

```
pie_chart(df,'Spiders')
```

Analysis:
Most of the patients did not have spiders contributing to a little more than 70% of the total share.

```
pie_chart(df,'Edema')
```

Analysis:
A lion's share of the population of patients neither had edema nor diuretic therapy for edema, contributing to almost 85% of the total share. The patients in case of which edema was present without diuretics or edema was resolved by diuretics had a share of just over one-tenth whereas those who had edema despite having diuretic therapy had a minimal share of a little less than 5%.

```
plt.figure(figsize=(13,8),dpi=150)
ax = sns.countplot(df.Stage)
for p in ax.patches:
   x=p.get_bbox().get_points()[:,0]
   y=p.get_bbox().get_points()[1,1]
   ax.annotate('{:.1f}%'.format(100.*y/len(df)), (x.mean(), y),
         ha='center', va='bottom')
```

Analysis:

It is clearly evident that the classes of the target feature "Stage" are highly imbalanced with 3rd stage being the majority class and 1st stage being the minority class.

```
#Bivariate Analysis

plt.figure(figsize=(12,8))
ax =
sns.scatterplot(x='Bilirubin',y='Copper',data=df,hue='Stage',s=200,alpha=0.9,pal
ette='spring')
plt.legend(bbox_to_anchor=(1.2,0.5),title="Stage")
```

Analysis: There is a mild positive correlation between Bilirubin and Copper.

```
males = df[df.Sex == 'M']
females = df[df.Sex == 'F']
```

```
males.describe()

females.describe()
```

Analysis:
Females tend to have more quantities of Bilirubin, Cholesterol, SGOT and Platelets as compared to males whereas males are more likely to have higher quantities of Albumin, Copper, Alkaline Phosphatase and Tryglicerides in comparison to their female counterparts.

```
round(males.Status.value_counts() / len(males) * 100,2)
```

```
round(females.Status.value_counts() / len(females) * 100,2)
```

Analysis:
Male patients are more prone to death in comparison to females while female patients are more likely to have "Censored" status.

```
round(males.Stage.value_counts() / len(males) * 100,2)

round(females.Stage.value_counts() / len(females) * 100,2)
```

Analysis:
Male patients are highly vulnerable to the critical stages (i.e. 3rd and 4th) of the Cirrhosis disease as compared to female patients.

```
round(males.Hepatomegaly.value_counts() / len(males) * 100,2)

round(females.Hepatomegaly.value_counts() / len(females) * 100,2)
```

Analysis:
Male patients are more likely to suffer from Hepatomegaly disease in comparison to female patients.

```
#Multivariate Analysis

plt.figure(figsize=(12,8))
sns.heatmap(df.corr(),annot=True,cmap='plasma',vmin=-1,vmax=1)


#There are no major correlations between any pair of features in the dataset.

sns.pairplot(df,hue='Stage')
```

Analysis:
Most of the numerical features such as Bilirubin, Prothrombin, Triglycerides, etc. follow the Gaussian distribution so their missing values can be imputed with their corresponding median values.

```
#Feature Engineering


df.Age = (df.Age.values/365.0).round()
df.head()


#Imputation of missing values


for col in df.columns:
    if df[col].dtypes != 'object':
        df[col].fillna(df[col].median(),inplace=True)
    else:
        df[col].fillna(df[col].mode()[0],inplace=True)



#Treatment of outliers


for col in df.columns:
    if df[col].dtypes != 'object':
        lower_limit, upper_limit = df[col].quantile([0.25,0.75])
        IQR = upper_limit- lower_limit
        lower_whisker = lower_limit- 1.5 * IQR
        upper_whisker = upper_limit + 1.5 * IQR
        df[col] =
np.where(df[col]>upper_whisker,upper_whisker,np.where(df[col]<lower_whiske
r,lower_whisker,df[col]))



#Encoding of categorical features

df.Sex.replace(['M','F'],[1,0],inplace=True)
df.Sex = df.Sex.astype(np.float64)
df.Edema.replace(['N','S','Y'],[0,-1,1],inplace=True)
df.Edema = df.Edema.astype(np.float64)
df.Ascites.replace(['Y','N'],[1,0],inplace=True)
df.Hepatomegaly.replace(['Y','N'],[1,0],inplace=True)
df.Spiders.replace(['Y','N'],[1,0],inplace=True)
df.Ascites = df.Ascites.astype(np.float64)
df.Hepatomegaly = df.Hepatomegaly.astype(np.float64)
df.Spiders = df.Spiders.astype(np.float64)
df.Drug.replace(['D-penicillamine','Placebo'],[0,1],inplace=True)
df.Drug = df.Drug.astype(np.float64)
df.Stage = df.Stage.astype(np.int64)

#Separating the independent predictor features and the target label

#We will not be using 'Status' and 'N_days' as our features since they will cause
#data leakage.

X = df.drop(['Stage','N_Days','Status'],axis=1)
y = df.Stage



#Balancing the imbalanced target column "Stage" using Synthetic Minority Over-
#Sampling Technique(SMOTE)

‘’’
```

  - Synthetic Minority Over-sampling Technique (SMOTE) is an oversampling technique used to address class imbalance in binary classification tasks.
  - It works by generating synthetic samples of the minority class by interpolating between existing minority class instances.
  - SMOTE helps to balance the class distribution in the training data, which can improve the performance of machine learning models, particularly those sensitive to class imbalance.
  - Scikit-learn provides an implementation of SMOTE through the `imbalanced-learn` package, which can be used in conjunction with other preprocessing techniques.

‘’’

```
smote = SMOTE()
X,y = smote.fit_resample(X,y)

sns.countplot(y)
```

#Feature Scaling

```
scaler = StandardScaler()
features = X.columns
X = scaler.fit_transform(X)
X = pd.DataFrame(X,columns=features)
X.head()
```

#Model Training & Evaluation

#Dividing the dataset into training and test sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)
```

#Logistic Regression

```
logmodel = LogisticRegression()
logmodel.fit(X_train,y_train)
```

```
y_pred = logmodel.predict(X_test)
print(classification_report(y_test,y_pred))


plot_confusion_matrix(logmodel,X_test,y_test)

svm = SVC()
svm.fit(X_train,y_train)

svm_pred = svm.predict(X_test)
print(classification_report(y_test,svm_pred))


plot_confusion_matrix(svm,X_test,y_test)


scores = []
for i in np.arange(1,21):
   knn = KNeighborsClassifier(n_neighbors=i)
   knn.fit(X_train,y_train)
   scores.append(knn.score(X_test,y_test))

plt.figure(figsize=(12,8))
sns.lineplot(np.arange(1,21),scores)
plt.xlabel('No. of neighbors')
plt.ylabel('Accuracy Score')


knn = KNeighborsClassifier(n_neighbors=17)
knn.fit(X_train,y_train)

knn_pred = knn.predict(X_test)
print(confusion_matrix(y_test,knn_pred))
print(classification_report(y_test,knn_pred))

sgd = SGDClassifier()
sgd.fit(X_train,y_train)

sgd_pred = sgd.predict(X_test)
```

```
print(confusion_matrix(y_test,sgd_pred))
print(classification_report(y_test,sgd_pred))

pac = PassiveAggressiveClassifier()
pac.fit(X_train,y_train)


pac_pred = pac.predict(X_test)
print(confusion_matrix(y_test,pac_pred))
print(classification_report(y_test,pac_pred))

ridge = RidgeClassifier()
ridge.fit(X_train,y_train)

ridge_pred = ridge.predict(X_test)
print(confusion_matrix(y_test,ridge_pred))
print(classification_report(y_test,ridge_pred))


dtree = DecisionTreeClassifier()
dtree.fit(X_train,y_train)

dtree_pred = dtree.predict(X_test)
print(confusion_matrix(y_test,dtree_pred))
print(classification_report(y_test,dtree_pred))

scores = []
for i in np.arange(100,1001,100):
   rf = RandomForestClassifier(n_estimators=i)
   rf.fit(X_train,y_train)
   scores.append(rf.score(X_test,y_test))

plt.figure(figsize=(12,8))
sns.lineplot(np.arange(100,1001,100),scores)
plt.xlabel('No. of Estimators',labelpad=15)
plt.ylabel('Accuracy Score',labelpad=15)


rf = RandomForestClassifier(n_estimators=1000)
rf.fit(X_train,y_train)
```

```
rf_pred = rf.predict(X_test)
print(confusion_matrix(y_test,rf_pred))
print(classification_report(y_test,rf_pred))


feature_importances =
pd.concat([pd.Series(features),pd.Series(rf.feature_importances_)],axis=1)
feature_importances.columns = ['Feature','Importance']
feature_importances =
feature_importances.sort_values('Importance',ascending=False)
sns.barplot(x='Importance',y='Feature',data=feature_importances,orient='h')


et = ExtraTreesClassifier()
et.fit(X_train,y_train)

et_pred = et.predict(X_test)
print(confusion_matrix(y_test,et_pred))
print(classification_report(y_test,et_pred))


abc = AdaBoostClassifier()
abc.fit(X_train,y_train)

abc_pred = abc.predict(X_test)
print(confusion_matrix(y_test,abc_pred))
print(classification_report(y_test,abc_pred))

cb = CatBoostClassifier()
cb.fit(X_train,y_train)

cb_pred = cb.predict(X_test)
print(confusion_matrix(y_test,cb_pred))
print(classification_report(y_test,cb_pred))

hgb = HistGradientBoostingClassifier()
hgb.fit(X_train,y_train)

hgb_pred = hgb.predict(X_test)
```

```
print(confusion_matrix(y_test,hgb_pred))
print(classification_report(y_test,hgb_pred))

bag = BaggingClassifier()
bag.fit(X_train,y_train)

bag_pred = bag.predict(X_test)
print(confusion_matrix(y_test,bag_pred))
print(classification_report(y_test,bag_pred))

gnb = GaussianNB()
gnb.fit(X_train,y_train)

gnb_pred = gnb.predict(X_test)
print(confusion_matrix(y_test,gnb_pred))
print(classification_report(y_test,gnb_pred))


gbc = GradientBoostingClassifier()
gbc.fit(X_train,y_train)

gbc_pred = gbc.predict(X_test)
print(confusion_matrix(y_test,gbc_pred))
print(classification_report(y_test,gbc_pred))

lgbm = LGBMClassifier()
lgbm.fit(X_train,y_train)


lgbm_pred = lgbm.predict(X_test)
print(confusion_matrix(y_test,lgbm_pred))
print(classification_report(y_test,lgbm_pred))

bnb = BernoulliNB()
bnb.fit(X_train,y_train)

bnb_pred = bnb.predict(X_test)
print(confusion_matrix(y_test,bnb_pred))
print(classification_report(y_test,bnb_pred))
```

```
xgb = XGBClassifier()
xgb.fit(X_train,y_train)


xgb_pred = xgb.predict(X_test)
print(confusion_matrix(y_test,xgb_pred))
print(classification_report(y_test,xgb_pred))

#Model Performance Analysis


print("Accuracy Score of Logistic
Regression:",str(np.round(logmodel.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Passive Aggressive
Classifier:",str(np.round(pac.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of SGD
Classifer:",str(np.round(sgd.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Ridge
Classifier:",str(np.round(ridge.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Gaussian Naive
Bayes:",str(np.round(gnb.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Bernoulli Naive
Bayes:",str(np.round(bnb.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of K Neighbors
Classifier:",str(np.round(knn.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Support Vector
Classifier:",str(np.round(svm.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Decision Tree
Classifier:",str(np.round(dtree.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Random Forest
Classifier:",str(np.round(rf.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of XG Boost
Classifier:",str(np.round(xgb.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Cat Boost
Classifier:",str(np.round(cb.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Gradient Boosting
Classifier:",str(np.round(gbc.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Histogram Gradient Boosting
Classifier:",str(np.round(hgb.score(X_test,y_test)*100,2)) + '%')
```

```
print("Accuracy Score of Bagging
Classifier:",str(np.round(bag.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Ada Boost
Classifier:",str(np.round(abc.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Extra Trees
Classifier:",str(np.round(et.score(X_test,y_test)*100,2)) + '%')
print("Accuracy Score of Light GBM
Classifier:",str(np.round(lgbm.score(X_test,y_test)*100,2)) + '%')
```

‘’’

Final Analysis:
Cat Boost Classifier is the best performing model that boasts a superb prediction accuracy close to 70%.
‘’’

‘’’

Extension Part:

Hyperparameter Tuning using GridSearchCV and other cross validation techniques
‘’’

```
param_grid = {'n_estimators': [200,400,600,800,1000],
            'criterion': ['gini','entropy'],
            'max_features': ['auto','sqrt','log2'],
            'class_weight': ['balanced','balanced_subsample']}
grid_rf =
GridSearchCV(RandomForestClassifier(),param_grid,verbose=2,cv=RepeatedStra
tifiedKFold(n_splits=5,n_repeats=2))
grid_rf.fit(X_train,y_train)

grid_rf.best_params_


grid_rf_pred = grid_rf.predict(X_test)
print(classification_report(y_test,grid_rf_pred))

cb_cv = cross_validate(cb,X_test,y_test,cv=8,verbose=2)
cb_cv = pd.DataFrame(cb_cv)
print("Mean Accuracy Score of Cat Boost
Classifier:",str(np.round(cb_cv.test_score.mean()*100,2)) + '%')
```

```
cv_cb =
cross_validate(cb,X_test,y_test,cv=RepeatedStratifiedKFold(n_splits=5,n_repeat
s=2),verbose=2,scoring=['accuracy','f1_weighted','recall_weighted','precision_w
eighted','r2','roc_auc_ovr_weighted'])
cv_cb = pd.DataFrame(cv_cb)
print("Mean Accuracy Score of Cat Boost
Classifier:",str(np.round(cv_cb.test_accuracy.mean()*100,2)) + '%')

cb_cv = cross_val_score(cb,X_test,y_test,cv=10,verbose=1,scoring='accuracy')
print("Mean Accuracy Score of Cat Boost
Classifier:",str(np.round(np.mean(cb_cv)*100,2)) + '%')


rf_cv = cross_val_score(rf,X_test,y_test,cv=10,verbose=1,scoring='accuracy')
print("Mean Accuracy Score of Random Forest
Classifier:",str(np.round(np.mean(rf_cv)*100,2)) + '%')

cv_rf =
cross_validate(rf,X_test,y_test,cv=RepeatedStratifiedKFold(n_splits=5,n_repeats
=2),verbose=2,scoring=['accuracy','f1_weighted','recall_weighted','precision_we
ighted','r2','roc_auc_ovr_weighted'])
cv_rf = pd.DataFrame(cv_rf)
print("Mean Accuracy Score of Random Forest
Classifier:",str(np.round(cv_rf.test_accuracy.mean()*100,2)) + '%')



et_cv =
cross_validate(et,X_test,y_test,cv=RepeatedStratifiedKFold(n_splits=3,n_repeats
=10),verbose=1,scoring=['accuracy','roc_auc_ovr_weighted','roc_auc_ovo_weig
hted','f1_weighted'])
et_cv = pd.DataFrame(et_cv)
et_cv.head()

print("Mean Accuracy Score of Extra Trees
Classifier:",str(np.round(et_cv.test_accuracy.mean()*100,2)) + '%')
```

```
param_grid = {'n_neighbors': np.arange(1,51),
            'weights': ['uniform','distance'],
            'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']}
grid_knn =
GridSearchCV(knn,param_grid,cv=RepeatedStratifiedKFold(n_splits=2,n_repeats
=6),verbose=2)
grid_knn.fit(X_train,y_train)


grid_knn.best_params_

grid_knn_pred = grid_knn.predict(X_test)
print(classification_report(y_test,grid_knn_pred))

param_grid = {'learning_rate': [0.2,0.4,0.5,0.8,1.0],
            'loss': ['auto', 'binary_crossentropy', 'categorical_crossentropy']}

grid_hgb =
GridSearchCV(HistGradientBoostingClassifier(),param_grid,cv=RepeatedStratifie
dKFold(n_splits=4,n_repeats=3),verbose=2)
grid_hgb.fit(X_train,y_train)

grid_hgb.best_params_

grid_hgb_pred = grid_hgb.predict(X_test)
print(classification_report(y_test,grid_hgb_pred))

sorted(sklearn.metrics.SCORERS.keys())

cv_lgbm =
cross_validate(lgbm,X_test,y_test,cv=RepeatedStratifiedKFold(n_splits=4,n_repe
ats=2),verbose=2,scoring=['accuracy','f1_weighted','recall_weighted','precision_
weighted','r2','roc_auc_ovr_weighted'])
cv_lgbm = pd.DataFrame(cv_lgbm)
print("Mean Accuracy Score of Light Gradient Boosting
Model:",str(np.round(cv_lgbm.test_accuracy.mean()*100,2)) + '%')
```

```
cv_abc =
cross_validate(abc,X_test,y_test,cv=RepeatedStratifiedKFold(n_splits=5,n_repea
ts=2),verbose=2,scoring=['accuracy','f1_weighted','recall_weighted','precision_
weighted','r2','roc_auc_ovr_weighted'])
cv_abc = pd.DataFrame(cv_abc)
print("Mean Accuracy Score of Ada Boost
Classifier:",str(np.round(cv_abc.test_accuracy.mean()*100,2)) + '%')


cv_hgb =
cross_validate(hgb,X_test,y_test,cv=RepeatedStratifiedKFold(n_splits=5,n_repea
ts=2),verbose=2,scoring=['accuracy','f1_weighted','recall_weighted','precision_
weighted','r2','roc_auc_ovr_weighted'])
cv_hgb = pd.DataFrame(cv_hgb)
print("Mean Accuracy Score of Histogram-based Gradient Boosting
Classifier:",str(np.round(cv_hgb.test_accuracy.mean()*100,2)) + '%')


cv_xgb =
cross_validate(xgb,X_test,y_test,cv=RepeatedStratifiedKFold(n_splits=5,n_repea
ts=2),verbose=2,scoring=['accuracy','f1_weighted','recall_weighted','precision_
weighted','r2','roc_auc_ovr_weighted'])
cv_xgb = pd.DataFrame(cv_xgb)
cv_xgb.head()


print("Mean Accuracy Score of XG Boost
Classifier:",str(np.round(cv_xgb.test_accuracy.mean()*100,2)) + '%')


#Saving the model for future use


joblib.dump(cb,'model.pkl')


model = joblib.load('model.pkl')
model
```

```
joblib.dump(scaler,'scaler.bin')

scaler = joblib.load('scaler.bin')
scaler
```

THANK YOU