



DATA SCIENCE WORKSHOP

CERVICAL CANCER

CLASSIFICATION AND PREDICTION
USING MACHINE LEARNING
AND DEEP LEARNING

WITH
PYTHON GUI

BALIGE CITY
NORTH SUMATERA

VIVIAN SIAHAAN
RISMON HASIHOLAN SIAHIPAR

DATA SCIENCE WORKSHOP:
CERVICAL CANCER CLASSIFICATION AND
PREDICTION
USING MACHINE LEARNING AND DEEP LEARNING
WITH PYTHON GUI

DATA SCIENCE WORKSHOP:
CERVICAL CANCER CLASSIFICATION AND
PREDICTION
USING MACHINE LEARNING AND DEEP LEARNING
WITH PYTHON GUI

Second Edition

VIVIAN SIAHAAN
RISMON HASIHOLAN SIANIPAR

Copyright © 2023 BALIGE Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor BALIGE Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book. BALIGE Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BALIGE Publishing cannot guarantee the accuracy of this information.

Copyright © 2023 BALIGE Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor BALIGE Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book. BALIGE Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BALIGE Publishing cannot guarantee the accuracy of this information.

Published: AGUST 2023
Production reference: 4280123
Published by BALIGE Publishing Ltd.
BALIGE, North Sumatera

ABOUT THE AUTHOR

ABOUT THE AUTHOR



Vivian Siahaan is a highly motivated individual with a passion for continuous learning and exploring new areas. Born and raised in Hinalang Bagasan, Balige, situated on the picturesque banks of Lake Toba, she completed her high school education at SMAN 1 Balige. Vivian's journey into the world of programming began with a deep dive into various languages such as Java, Android, JavaScript, CSS, C++, Python, R, Visual Basic, Visual C#, MATLAB, Mathematica, PHP, JSP, MySQL, SQL Server, Oracle, Access, and more. Starting from scratch, Vivian diligently studied programming, focusing on mastering the fundamental syntax and logic. She honed her skills by creating practical GUI applications, gradually building her expertise. One particular area of interest for Vivian is animation and game development, where she aspires to make

significant contributions. Alongside her programming and mathematical pursuits, she also finds joy in indulging in novels, nurturing her love for literature. Vivian Siahaan's passion for programming and her extensive knowledge are reflected in the numerous ebooks she has authored. Her works, published by Sparta Publisher, cover a wide range of topics, including "Data Structure with Java," "Java Programming: Cookbook," "C++ Programming: Cookbook," "C Programming For High Schools/Vocational Schools and Students," "Java Programming for SMA/SMK," "Java Tutorial: GUI, Graphics and Animation," "Visual Basic Programming: From A to Z," "Java Programming for Animation and Games," "C# Programming for SMA/SMK and Students," "MATLAB For Students and Researchers," "Graphics in JavaScript: Quick Learning Series," "JavaScript Image Processing Methods: From A to Z," "Java GUI Case Study: AWT & Swing," "Basic CSS and JavaScript," "PHP/MySQL Programming: Cookbook," "Visual Basic: Cookbook," "C++ Programming for High Schools/Vocational Schools and Students," "Concepts and Practices of C++," "PHP/MySQL For Students," "C# Programming: From A to Z," "Visual Basic for SMA/SMK and Students," and "C# .NET and SQL Server for High School/Vocational School and Students." Furthermore, at the ANDI Yogyakarta publisher, Vivian Siahaan has contributed to several notable books, including "Python Programming Theory and Practice," "Python GUI Programming," "Python GUI and Database," "Build From Zero School Database Management System In Python/MySQL," "Database Management System in Python/MySQL," "Python/MySQL For Management Systems of Criminal Track Record Database," "Java/MySQL For Management Systems of Criminal Track Records Database," "Database and Cryptography Using Java/MySQL," and "Build From Zero School Database Management System With Java/MySQL." Vivian's diverse range of expertise in programming languages, combined with her passion for exploring new horizons, makes her a dynamic and versatile individual in the field of technology. Her dedication to learning, coupled with her strong analytical and problem-solving skills, positions her as a valuable asset in any programming endeavor. Vivian Siahaan's contributions to the

world of programming and literature continue to inspire and empower aspiring programmers and readers alike.



Rismon Hasiholan Sianipar, born in Pematang Siantar in 1994, is a distinguished researcher and expert in the field of electrical engineering. After completing his education at SMAN 3 Pematang Siantar, Rismon ventured to the city of Jogjakarta to pursue his academic journey. He obtained his Bachelor of Engineering (S.T) and Master of Engineering (M.T) degrees in Electrical Engineering from Gadjah Mada University in 1998 and 2001, respectively, under the guidance of esteemed professors, Dr. Adhi Soesanto and Dr. Thomas Sri Widodo. During his studies, Rismon focused on researching non-stationary signals and their energy analysis using time-frequency maps. He explored the dynamic nature of signal energy distribution on time-frequency maps and developed innovative techniques using discrete wavelet transformations to design non-linear filters for data pattern analysis. His research showcased the application of these techniques in various fields. In recognition of his academic prowess, Rismon was awarded the prestigious Monbukagakusho scholarship by the Japanese Government in 2003. He went on to pursue his Master of Engineering (M.Eng) and Doctor of Engineering (Dr.Eng) degrees at Yamaguchi University, supervised by Prof. Dr. Hidetoshi Miike. Rismon's master's and doctoral theses revolved around combining the SR-FHN (Stochastic Resonance Fitzhugh-Nagumo) filter strength with the cryptosystem ECC (elliptic curve cryptography) 4096-bit. This innovative approach effectively suppressed noise in digital images and videos while ensuring their authenticity. Rismon's research findings have been published in renowned international scientific journals, and his patents have been officially registered in Japan. Notably, one of his patents, with registration number 2008-009549, gained recognition. He actively collaborates with several universities and research institutions in Japan, specializing in cryptography, cryptanalysis, and digital forensics, particularly in the areas of audio, image, and video analysis. With a passion for knowledge sharing, Rismon has authored numerous national and international scientific articles and authored several national books. He has also actively participated in workshops related to cryptography, cryptanalysis, digital watermarking, and digital forensics. During these workshops, Rismon has assisted Prof. Hidetoshi Miike in developing applications related to digital image and video processing, steganography, cryptography, watermarking, and more, which serve as valuable training materials. Rismon's field of interest encompasses multimedia security, signal processing, digital image and video analysis, cryptography, digital communication, digital forensics, and data compression. He continues to advance his research by developing applications using programming languages such as Python, MATLAB, C++, C, VB.NET, C#.NET, R, and Java. These applications serve both research and commercial purposes, further contributing to the advancement of signal and image analysis. Rismon Hasiholan Sianipar is a dedicated researcher and expert in the field of electrical engineering, particularly in the areas of signal processing, cryptography, and digital forensics. His academic achievements, patented inventions, and extensive publications demonstrate his commitment to advancing knowledge in these fields. Rismon's contributions to academia and his collaborations with prestigious institutions in Japan have solidified his position as a respected figure in the scientific community. Through his ongoing research and

development of innovative applications, Rismon continues to make significant contributions to the field of electrical engineering.

ABOUT THE BOOK

ABOUT THE BOOK

This book titled " Data Science Workshop: Cervical Cancer Classification and Prediction using Machine Learning and Deep Learning with Python GUI" embarks on an insightful journey starting with an in-depth exploration of the dataset. This dataset encompasses various features that shed light on patients' medical histories and attributes. Utilizing the capabilities of pandas, the dataset is loaded, and essential details like data dimensions, column names, and data types are scrutinized. The presence of missing data is addressed by employing suitable strategies such as mean-based imputation for numerical features and categorical encoding for non-numeric ones.

Subsequently, the project delves into an illuminating visualization of categorized feature distributions. Through the ingenious use of pie charts, bar plots, and heatmaps, the project unveils the distribution patterns of key attributes such as 'Hormonal Contraceptives,' 'Smokes,' 'IUD,' and others. These visualizations illuminate potential relationships between these features and the target variable 'Biopsy,' which signifies the presence or absence of cervical cancer. Such exploratory analyses serve as a vital foundation for identifying influential trends within the dataset.

Transitioning into the core phase of predictive modeling, the workshop orchestrates a meticulous ensemble of machine learning models to forecast cervical cancer outcomes. The repertoire includes Logistic Regression, Decision Trees, Random Forests, Support Vector Machines (SVM), K-Nearest Neighbors (KNN), Gradient Boosting, Naïve Bayes, and the power of ensemble methods like AdaBoost and XGBoost. The models undergo rigorous hyperparameter tuning facilitated by Grid Search and Random Search to optimize predictive accuracy and precision.

As the workshop progresses, the spotlight shifts to the realm of deep learning, introducing advanced neural network architectures. An Artificial Neural Network (ANN) featuring multiple hidden layers is trained using the backpropagation algorithm. Long Short-Term Memory (LSTM) networks are harnessed to capture intricate temporal relationships within the data. The arsenal extends to include Self Organizing Maps (SOMs), Restricted Boltzmann Machines (RBMs), and Autoencoders, showcasing the efficacy of unsupervised feature learning and dimensionality reduction techniques.

The evaluation phase emerges as a pivotal aspect, accentuated by an array of comprehensive metrics. Performance assessment encompasses metrics such as accuracy, precision, recall, F1-score, and ROC-AUC. Cross-validation and learning curves are strategically employed to mitigate overfitting and ensure model generalization. Furthermore,

visual aids such as ROC curves and confusion matrices provide a lucid depiction of the models' interplay between sensitivity and specificity.

Culminating on a high note, the workshop concludes with the creation of a Python GUI utilizing PyQt. This intuitive graphical user interface empowers users to input pertinent medical data and receive instant predictions regarding their cervical cancer risk. Seamlessly integrating the most proficient classification model, this user-friendly interface bridges the gap between sophisticated data science techniques and practical healthcare applications.

In this comprehensive workshop, participants navigate through the intricate landscape of data exploration, preprocessing, feature visualization, predictive modeling encompassing both traditional and deep learning paradigms, robust performance evaluation, and culminating in the development of an accessible and informative GUI. The project aspires to provide healthcare professionals and individuals with a potent tool for early cervical cancer detection and prognosis.

CONTENT
CONTENT

<i>EXPLORING DATASET AND FEATURES DISTRIBUTION</i>	1
Description	1
Exploring Dataset	3
Information of Dataset	8
Checking Null Values	10
Dropping Irrelevant Columns	12
Impact of Hormonal Contraceptives	15
Impact of IUD	16
Impact of STDs	18
Correlation Matrix	20
Distribution of Target Variable	21
Distribution of Hinselmann	24
Distribution of Schiller	25
Distribution of Citology	26

Distribution of Biopsy versus Other Features	27
Distribution of Hinselmann versus Other Features	29
Distribution of Schiller versus Other Features	30
Distribution of Citology versus Other Features	32
Distribution of Hormonal Contraceptives and Smokes versus Biopsy	33
Distribution of IUD and STDs versus Biopsy	35
Histogram and Density	36
	47
<i>PREDICTING CERVICAL CANCER USING MACHINE LEARNING</i>	47
Features Importance Using Random Forest Classifier	51
Features Importance Using Extra Trees Classifier	54
Features Importance Using RFE	57
Resampling and Splitting Data	58
Learning Curve	60
Confusion Matrix and Predicted versus True Values Diagram	62
ROC and Decision Boundaries	63
Training Model and Predicting Cervical Cancer	66
Support Vector Classifier and Grid Search	76
Logistic Regression Classifier and Grid Search	87
K-Nearest Neighbors Classifier and Grid Search	97
Decision Trees Classifier and Grid Search	107
Random Forest Classifier and Grid Search	119
Gradient Boosting Classifier and Grid Search	128
Extreme Gradient Boosting Classifier and Grid Search	134
Multi-Layer Perceptron Classifier and Grid Search	139
Source Code	158
<i>PREDICTING CERVICAL CANCER USING DEEP LEARNING</i>	158
Reading Dataset	159
Preprocessing	160
Resampling and Splitting Data	161
Building, Compiling, and Training ANN Model	175
Accuracy and Loss	178

Predicting Result Using Test Data	179
Classification Report	181
Confusion Matrix	182
True Values versus Predicted Values Diagram	184
Source Code	187
Cervical Cancer Using LSTM Model	192
Cervical Cancer Using Self Organizing Maps (SOMs) Model	197
Cervical Cancer Using Deep Belief Networks (DBNs) Model	202
Cervical Cancer Using Restricted Boltzmann Machines (RBMs) Model	206
Cervical Cancer Using Autoencoders Model	212
 	212
IMPLEMENTING GUI USING PYQT	218
Designing GUI	226
Reading Dataset and Preprocessing	233
Resampling and Splitting Data	240
Distribution of Features	249
Histogram and Density	252
Correlation Matrix and Features Importance	253
Real Values versus Predicted Values and Confusion Matrix	255
ROC	256
Learning Curve	259
Scalability and Performance Curves	263
Training Model and Predicting Result	267
Logistic Regression Classifier	271
Support Vector Classifier	274
K-Nearest Neighbors Classifier	278
Decision Trees Classifier	282
Random Forest Classifier	286
Gradient Boosting Classifier	289
Naïve Bayes Classifier	293
AdaBoost Classifier	296
Extreme Gradient Boosting Classifier	299
Light Gradient Boosting Classifier	303
Multi-Layer Perceptron Classifier	311

ANN Classifier
Source Code

**EXPLORING
DATASET
AND FEATURES DISTRIBUTION**

**EXPLORING
DATASET
AND FEATURES DISTRIBUTION**

Description

About 11,000 new cases of invasive cervical cancer are diagnosed each year in the U.S. However, the number of new cervical cancer cases has been declining steadily over the past decades. Although it is the most preventable type of cancer, each year cervical cancer kills about 4,000 women in the U.S. and about 300,000 women worldwide.

In the United States, cervical cancer mortality rates plunged by 74% from 1955 - 1992 thanks to increased screening and early detection with the Pap test. AGE Fifty percent of cervical cancer diagnoses occur in women ages 35 - 54, and about 20% occur in women over 65 years of age. The median age of diagnosis is 48 years. About 15% of women develop cervical cancer between the ages of 20 - 30. Cervical cancer is extremely rare in women younger than age 20. However, many young women become infected with multiple types of human papilloma virus, which then can increase their risk of getting cervical cancer in the future.

Young women with early abnormal changes who do not have regular examinations are at high risk for localized cancer by the time they are age 40, and for invasive cancer by age 50.

Numerous studies report that high poverty levels are linked with low screening rates. In addition, lack of health insurance, limited transportation, and language difficulties hinder a poor woman's access to screening services. Human papilloma virus (HPV) is the main risk factor for cervical cancer. In adults, the most important risk factor for HPV is sexual activity with an infected person.

Women most at risk for cervical cancer are those with a history of multiple sexual partners, sexual intercourse at age 17 years or younger, or both. A woman who has never been sexually active has a very low risk for developing cervical cancer. Sexual activity with multiple partners increases the likelihood of many other sexually transmitted infections (chlamydia, gonorrhea, syphilis). Studies have

found an association between chlamydia and cervical cancer risk, including the possibility that chlamydia may prolong HPV infection.

Women have a higher risk of cervical cancer if they have a first-degree relative (mother, sister) who has had cervical cancer. Studies have reported a strong association between cervical cancer and long-term use of oral contraception (OC). Women who take birth control pills for more than 5 - 10 years appear to have a much higher risk HPV infection (up to four times higher) than those who do not use OCs. (Women taking OCs for fewer than 5 years do not have a significantly higher risk.) The reasons for this risk from OC use are not entirely clear. Women who use OCs may be less likely to use a diaphragm, condoms, or other methods that offer some protection against sexual transmitted diseases, including HPV.

Some research also suggests that the hormones in OCs might help the virus enter the genetic material of cervical cells. Studies indicate that having many children increases the risk for developing cervical cancer, particularly in women infected with HPV. Smoking is associated with a higher risk for precancerous changes (dysplasia) in the cervix and for progression to invasive cervical cancer, especially for women infected with HPV.

Women with weak immune systems, (such as those with HIV / AIDS), are more susceptible to acquiring HPV. Immunocompromised patients are also at higher risk for having cervical precancer develop rapidly into invasive cancer.

From 1938 - 1971, diethylstilbestrol (DES), an estrogen-related drug, was widely prescribed to pregnant women to help prevent miscarriages. The daughters of these women face a higher risk for cervical cancer. DES is no longer prescribed.

The dataset used in this projects is Obtained from UCI Repository and kindly acknowledged. This file contains a List of Risk Factors for Cervical Cancer leading to a Biopsy Examination.

Exploring Dataset

Step 1 Download dataset from CERVICAL

<https://viviansiahaan.blogspot.com/2023/08/data-science-workshop-cervical-cancer.html> and save it to your working directory. Unzip file, *kag_risk_factors_cervical_cancer.csv* and place it into working directory.

Step 2 Open a new Python script and save it as **cervical.py**.

Step 3 Import all necessary libraries:

```
1 #cervical.py
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import seaborn as sns
6 sns.set_style('darkgrid')
7 from sklearn.preprocessing import LabelEncoder
8 import warnings
9 warnings.filterwarnings('ignore')
10 import os
11 import joblib
12 from sklearn.metrics import
13 roc_auc_score,roc_curve
14 from sklearn.model_selection import
15 train_test_split, RandomizedSearchCV,
16 GridSearchCV, StratifiedKFold
17 from sklearn.preprocessing import StandardScaler,
18 MinMaxScaler
19
```

```
20 from sklearn.linear_model import
21 LogisticRegression
22 from sklearn.naive_bayes import GaussianNB
23 from sklearn.tree import DecisionTreeClassifier
24 from sklearn.svm import SVC
25 from sklearn.ensemble import
26 RandomForestClassifier, ExtraTreesClassifier
27 from sklearn.neighbors import
28 KNeighborsClassifier
29 from sklearn.ensemble import AdaBoostClassifier,
30 GradientBoostingClassifier
31 from xgboost import XGBClassifier
32 from sklearn.neural_network import
33 MLPClassifier
34 from sklearn.linear_model import SGDClassifier
35 from sklearn.preprocessing import StandardScaler,
36 LabelEncoder, OneHotEncoder
from sklearn.metrics import confusion_matrix,
accuracy_score, recall_score, precision_score
from sklearn.metrics import classification_report,
f1_score, plot_confusion_matrix
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import
learning_curve
from mlxtend.plotting import
plot_decision_regions
```

The cervical.py script encompasses essential data preprocessing, model training, and evaluation procedures for predicting cervical cancer biopsy outcomes. The script employs various libraries and functions to facilitate its operations. It starts by importing necessary libraries for data manipulation and visualization, such as NumPy, Pandas, Matplotlib, and Seaborn, and customizes the visualization

style with a dark grid background. The LabelEncoder is employed to encode categorical features, and warning messages are suppressed.

The script then imports machine learning components from scikit-learn, XGBoost, CatBoost, LightGBM, and other libraries, enabling the use of a diverse range of classifiers for modeling. Different classifiers are imported, including Logistic Regression, Naive Bayes, Decision Trees, Support Vector Machines, Random Forests, K-Nearest Neighbors, AdaBoost, Gradient Boosting, XGBoost, Neural Networks, and more. The SMOTE technique from the imblearn library is used to address class imbalance through oversampling.

The script also includes functions for model evaluation, such as confusion matrix, accuracy, recall, precision, F1-score, and ROC curve analysis. These functions facilitate the assessment of model performance and help make informed decisions regarding model selection. The learning_curve function is imported from scikit-learn to analyze the learning curve of the models, providing insights into their training and validation performance.

Overall, the cervical.py script serves as a comprehensive tool for conducting a wide array of tasks related to cervical cancer prediction, including data preprocessing, model training, hyperparameter tuning, evaluation, and visualization. It demonstrates a thorough and systematic approach to building and evaluating predictive models for medical diagnosis.

Step 4 Read dataset:

```
1 curr_path = os.getcwd()
2 df =
3 pd.read_csv(curr_path+"/kag_risk_factors_cervical_cancer.csv")
4 print(df.iloc[:,0:5].head().to_string())
5 print(df.iloc[:,5:10].head().to_string())
print(df.iloc[:,30:37].head().to_string())
```

Output:

Age Number of sexual partners First sexual intercourse Num of pregnancies Smokes

0	18	4.0	15.0	1.0	0.0
1	15	1.0	14.0	1.0	0.0
2	34	1.0	?	1.0	0.0
3	52	5.0	16.0	4.0	1.0
4	46	3.0	21.0	4.0	0.0

Smokes (years) Smokes (packs/year) Hormonal Contraceptives

0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0
3	37.0	37.0	1.0
4	0.0	0.0	1.0

Dx:HPV Dx Hinselmann Schiller Citology Biopsy

0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	1	0	0	0	0
4	0	0	0	0	0

The code reads a CSV file named "kag_risk_factors_cervical_cancer.csv" located in the current working directory using Pandas. It then prints a subset of the DataFrame to the console to visualize the data.

The first print statement displays the first five rows and the first five columns (columns 0 to 4) of the DataFrame, providing a glimpse of the dataset's attributes and initial data entries.

The second print statement displays the first five rows and the next five columns (columns 5 to 9) of the DataFrame, continuing to show

additional attributes and data entries.

The third print statement displays the first five rows and a range of columns (columns 30 to 36) from the DataFrame, which likely contains more attributes or features related to the dataset.

This code is helpful for quickly examining different sections of the dataset and understanding its structure and contents. It allows you to check the data in segments to get a better sense of its layout and what kind of information it contains.

Step 5 Check the shape of dataset:

```
1 #Checks shape  
2 print(df.shape)
```

Output:

(858, 36)

The print(df.shape) statement outputs the shape of the DataFrame, which represents the number of rows and columns in the dataset. The output will be in the format (number of rows, number of columns). This information is useful for understanding the size and dimensions of the dataset. The printed output will show the total count of rows and columns in the DataFrame.

The output (858, 36) indicates that the dataset contains 858 rows (instances) and 36 columns (features). Each row represents a unique data entry or sample, while each column represents a different attribute or feature associated with the data. This information is helpful for understanding the size and structure of the dataset you are working with.

Step 6 Read every column in dataset:

```
1 #Reads columns  
2 print("Data Columns --> ",df.columns)
```

Output:

Data Columns --> Index(['Age', 'Number of sexual partners', 'First sexual intercourse',
'Num of pregnancies', 'Smokes', 'Smokes (years)', 'Smokes (packs/year)',
'Hormonal Contraceptives', 'Hormonal Contraceptives (years)',
'IUD',
'IUD (years)', 'STDs', 'STDs (number)', 'STDs:condylomatosis',
'STDs:cervical condylomatosis', 'STDs:vaginal condylomatosis',
'STDs:vulvo-perineal condylomatosis', 'STDs:syphilis',
'STDs:pelvic inflammatory disease', 'STDs:genital herpes',
'STDs:molluscum contagiosum', 'STDs:AIDS', 'STDs:HIV',
'STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of diagnosis',
'STDs: Time since first diagnosis', 'STDs: Time since last diagnosis',
'Dx:Cancer', 'Dx:CIN', 'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller',
'Citology', 'Biopsy'],
dtype='object')

The code `print("Data Columns --> ",df.columns)` is used to display the names of the columns in the dataset. It will print a list of column names, providing you with information about the features or attributes present in the dataset. This can be useful for identifying the variables you have available for analysis and modeling.

The output shows the column names present in the dataset. Each column corresponds to a different attribute or feature of the data. Here's a brief description of some of the columns:

1. Age: The age of the individual.
2. Number of sexual partners: The number of sexual partners the individual has had.

3. First sexual intercourse: Age at which the individual had their first sexual intercourse.
4. Num of pregnancies: Number of pregnancies the individual has had.
5. Smokes: Whether the individual smokes or not.
6. Smokes (years): Number of years the individual has been smoking.
7. Smokes (packs/year): Number of packs of cigarettes the individual smokes per year.
8. Hormonal Contraceptives: Whether the individual uses hormonal contraceptives or not.
9. Hormonal Contraceptives (years): Number of years the individual has been using hormonal contraceptives.
10. IUD: Whether the individual uses an intrauterine device (IUD) or not.
11. IUD (years): Number of years the individual has been using an IUD.
12. STDs: Whether the individual has had any sexually transmitted diseases (STDs) or not.
13. STDs (number): Number of STDs the individual has had.
14. STDs:condylomatosis: Specific type of STD (condylomatosis).
15. STDs:cervical condylomatosis: Specific type of STD (cervical condylomatosis).
16. STDs:vaginal condylomatosis: Specific type of STD (vaginal condylomatosis).
17. STDs:vulvo-perineal condylomatosis: Specific type of STD (vulvo-perineal condylomatosis).
18. STDs:syphilis: Whether the individual has had syphilis or not.

19. STDs: pelvic inflammatory disease: Whether the individual has had pelvic inflammatory disease (PID) or not.
20. STDs: genital herpes: Whether the individual has had genital herpes or not.
21. STDs: molluscum contagiosum: Whether the individual has had molluscum contagiosum or not.
22. STDs: AIDS: Whether the individual has had AIDS or not.
23. STDs: HIV: Whether the individual has HIV or not.
24. STDs: Hepatitis B: Whether the individual has hepatitis B or not.
25. STDs: HPV: Whether the individual has human papillomavirus (HPV) or not.
26. STDs: Number of diagnosis: Number of STD diagnoses.
27. STDs: Time since first diagnosis: Time since the first STD diagnosis.
28. STDs: Time since last diagnosis: Time since the last STD diagnosis.
29. Dx: Cancer: Whether the individual has been diagnosed with cancer.
30. Dx: CIN: Whether the individual has been diagnosed with cervical intraepithelial neoplasia (CIN).
31. Dx: HPV: Whether the individual has been diagnosed with HPV.
32. Dx: Diagnosis (combination of cancer, CIN, and HPV diagnoses).
33. Hinselmann: Whether the Hinselmann criterion is met (a type of cervical cancer prediction).

34. Schiller: Whether the Schiller criterion is met (another cervical cancer prediction).
35. Citology: Whether cytology (cellular examination) is performed.
36. Biopsy: Whether a biopsy (tissue sample) was taken for diagnosis.

These columns represent various health-related attributes of individuals and are used for predicting the risk of cervical cancer.

Information of Dataset

Step 1 Check the information of dataset:

1

```
1 #Checks dataset information
2 print(df.info())
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 858 entries, 0 to 857
Data columns (total 36 columns):
 #   Column           Non-Null Count Dtype
 --- 
 0   Age              858 non-null   int64
 1   Number of sexual partners      858 non-null   object
 2   First sexual intercourse     858 non-null   object
 3   Num of pregnancies        858 non-null   object
 4   Smokes             858 non-null   object
 5   Smokes (years)         858 non-null   object
 6   Smokes (packs/year)     858 non-null   object
 7   Hormonal Contraceptives    858 non-null   object
 8   Hormonal Contraceptives (years) 858 non-null   object
 9   IUD                858 non-null   object
 10  IUD (years)          858 non-null   object
```

```

11 STDs           858 non-null object
12 STDs (number) 858 non-null object
13 STDs:condylomatosis 858 non-null object
14 STDs:cervical condylomatosis 858 non-null object
15 STDs:vaginal condylomatosis 858 non-null object
16 STDs:vulvo-perineal condylomatosis 858 non-null
object
17 STDs:syphilis 858 non-null object
18 STDs:pelvic inflammatory disease 858 non-null
object
19 STDs:genital herpes 858 non-null object
20 STDs:molluscum contagiosum 858 non-null
object
21 STDs:AIDS 858 non-null object
22 STDs:HIV 858 non-null object
23 STDs:Hepatitis B 858 non-null object
24 STDs:HPV 858 non-null object
25 STDs: Number of diagnosis 858 non-null int64
26 STDs: Time since first diagnosis 858 non-null object
27 STDs: Time since last diagnosis 858 non-null object
28 Dx:Cancer 858 non-null int64
29 Dx:CIN 858 non-null int64
30 Dx:HPV 858 non-null int64
31 Dx 858 non-null int64
32 Hinselmann 858 non-null int64
33 Schiller 858 non-null int64
34 Cytology 858 non-null int64
35 Biopsy 858 non-null int64
dtypes: int64(10), object(26)
memory usage: 241.4+ KB
None

```

The `df.info()` function provides information about the dataset, including the data types of columns and the number of non-null values in each column. It also gives an overview of the memory usage.

The output represents a summary of the dataset using the info() function in Pandas. Here's an analysis and conclusion based on the information provided:

The dataset contains a total of 858 entries (rows) and 36 columns. Each row corresponds to an individual observation, and each column represents a specific attribute or feature associated with those observations.

The dataset includes both numerical and categorical data. Among the columns, there are 10 columns with numerical data represented by the int64 data type. These numerical columns, such as 'Age', 'STDs: Number of diagnosis', 'Dx:Cancer', 'Dx:CIN', 'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller', 'Citology', and 'Biopsy', likely represent counts, binary indicators, or categorical labels.

The remaining 26 columns have the object data type, indicating that they contain categorical or string data. These columns are likely to represent various attributes related to sexual health and medical history. Some examples of these categorical columns include 'Number of sexual partners', 'Smokes', 'Hormonal Contraceptives', 'STDs', 'STDs:condylomatosis', and so on.

It's important to note that some columns, such as 'Number of sexual partners', 'First sexual intercourse', 'Num of pregnancies', and others, should ideally be numerical but are currently stored as objects. These columns might require data cleaning or conversion to numerical types for accurate analysis and modeling.

In conclusion, this dataset appears to contain information related to various attributes of individuals, particularly focusing on factors related to sexual health and medical

history. The mix of numerical and categorical data types suggests the need for appropriate preprocessing steps, such as handling missing values, converting categorical variables into suitable formats, and performing feature scaling before applying machine learning algorithms for further analysis or prediction tasks.

Checking Null Values

Step Check null values:

1

```
1 #Checks null values
2 df = df.replace('?', np.NaN)
3 print(df.isnull().sum())
4 print('Total number of null values: ', df.isnull().sum().sum())
5 plt.figure(figsize=(10,10))
6 np.round(df.isnull().sum()/df.shape[0]*100).sort_values().plot(\n    kind='barh')
```

Output:

Age	0
Number of sexual partners	26
First sexual intercourse	7
Num of pregnancies	56
Smokes	13
Smokes (years)	13
Smokes (packs/year)	13
Hormonal Contraceptives	108
Hormonal Contraceptives (years)	108
IUD	117
IUD (years)	117
STDs	105
STDs (number)	105
STDs:condylomatosis	105

```

STDs:cervical condylomatosis      105
STDs:vaginal condylomatosis       105
STDs:vulvo-perineal condylomatosis 105
STDs:syphilis                     105
STDs:pelvic inflammatory disease   105
STDs:genital herpes                105
STDs:molluscum contagiosum        105
STDs:AIDS                         105
STDs:HIV                          105
STDs:Hepatitis B                  105
STDs:HPV                          105
STDs: Number of diagnosis         0
STDs: Time since first diagnosis  787
STDs: Time since last diagnosis   787
Dx:Cancer                         0
Dx:CIN                            0
Dx:HPV                            0
Dx                               0
Hinselmann                        0
Schiller                           0
Citology                           0
Biopsy                            0
dtype: int64
Total number of null values: 3622

```

The code checks for missing values in the dataset, replaces any '?' values with NaN, prints the number of null values for each column, and then visualizes the percentage of missing values in each column using a horizontal bar plot. Here's an analysis and conclusion based on the output:

The output of the code shows the count of missing values in each column of the dataset. It's evident that several columns have missing values. The number of missing values varies across columns, with some columns having more missing values than others.

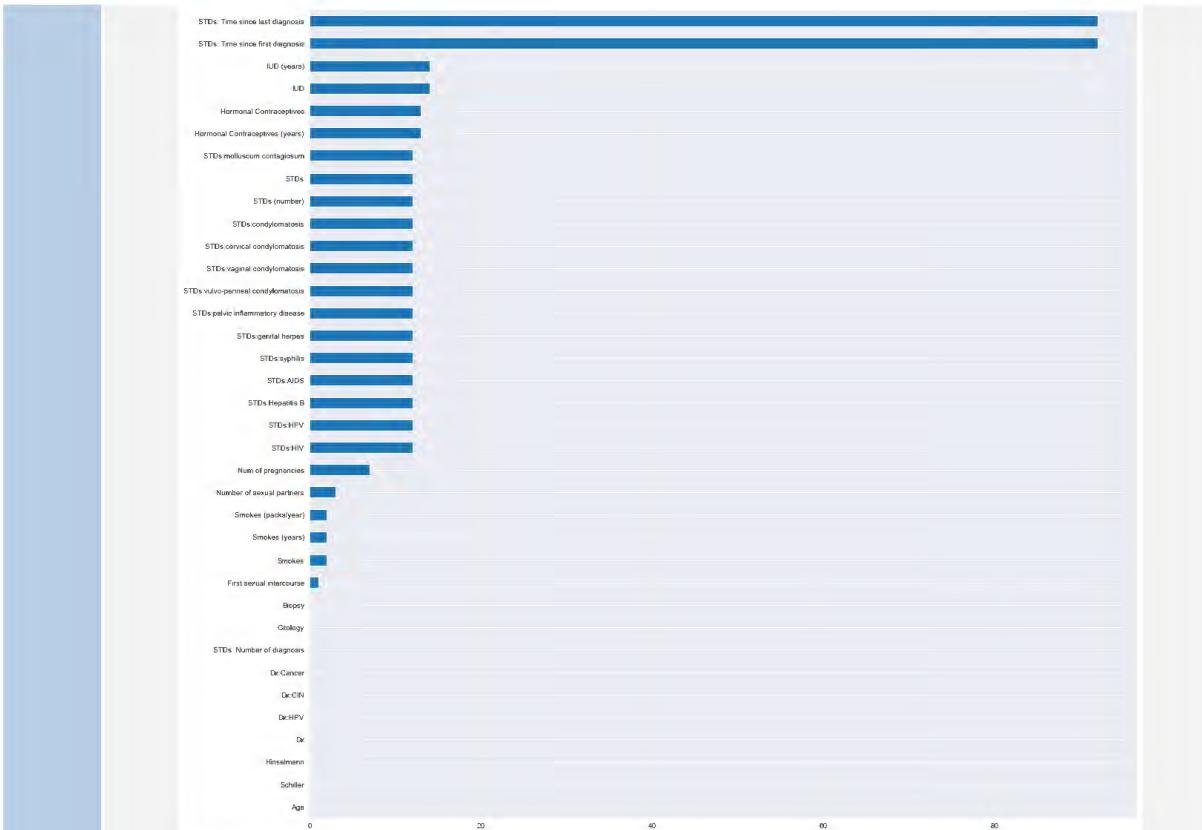


Figure 1 Number of missing values in each feature

The total number of missing values in the entire dataset is also calculated and printed. This gives an overall view of the extent of missing data in the dataset.

The horizontal bar plot provides a visual representation of the percentage of missing values in each column. Columns with a higher percentage of missing values will have longer bars in the plot.

This analysis suggests that the dataset contains missing values that need to be addressed before further analysis or modeling. Missing values can affect the performance of machine learning algorithms and can lead to biased results. Therefore, it is important to handle these missing values appropriately.

In conclusion, the dataset has missing values that need to be imputed or handled before proceeding with any further analysis or modeling.

The visualization of missing values provides insights into which columns have the most missing data and can guide the data preprocessing steps to ensure accurate and reliable results from any subsequent analyses or machine learning tasks.

The total number of missing values in the entire dataset is 3622. This significant number of missing values, especially in columns like 'STDs: Time since first diagnosis' and 'STDs: Time since last diagnosis', suggests that there is a substantial amount of incomplete information in these columns. Handling missing data is crucial for accurate analysis and modeling. Depending on the context and the significance of each column, you may need to decide whether to impute the missing values, remove rows with missing values, or take other appropriate measures to address the missing data before proceeding with further analysis or modeling.

Dropping Irrelevant Columns

Step 1 Drop two columns of STDs as they do not give much information because of missing data:

```
1 #Drops two columns of STDs
2 df.drop(['STDs: Time since first diagnosis',
3          'STDs: Time since last
diagnosis'],inplace=True,axis=1)
```

You have dropped the columns 'STDs: Time since first diagnosis' and 'STDs: Time since last diagnosis' from the DataFrame. These columns had a high number of missing values, and if they were not contributing significantly to your analysis or modeling, it's a reasonable approach to drop them to avoid dealing with the missing data. Remember to consider the implications of dropping columns and ensure

that it aligns with your data analysis goals. If you have any further analysis or tasks, feel free to proceed!

Imputing Null Values

Step 1 Define numerical columns and categorical columns:

```
1 numerical_df = ['Age', 'Number of sexual partners', \
2 'First sexual intercourse','Num of pregnancies', \
3 'Smokes (years)', 'Smokes (packs/year)',\
4 'Hormonal Contraceptives (years)','IUD (years)',\
5 'STDs (number)']

6

7 categorical_df = ['Smokes','Hormonal
8 Contraceptives',\
9 'IUD','STDs','STDs:condylomatosis',\
10 'STDs:cervical condylomatosis',\
11 'STDs:vaginal condylomatosis',\
12 'STDs:vulvo-perineal condylomatosis',
13 'STDs:syphilis',\
14 'STDs:pelvic inflammatory disease', \
15 'STDs:genital herpes','STDs:molluscum
contagiosum', \
16 'STDs:AIDS', 'STDs:HIV','STDs:Hepatitis B',
'STDs:HPV', \
'STDs: Number of diagnosis','Dx:Cancer', 'Dx:CIN',
 \
'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller','Citology',
'Biopsy']
```

The code snippet is a categorization of columns in a dataset into two lists: numerical_df and categorical_df. This

organization helps to distinguish between numerical and categorical features, allowing for easier handling of different types of data preprocessing or analysis tasks.

In the numerical_df list, columns related to numerical data are included, such as 'Age', 'Number of sexual partners', 'First sexual intercourse', 'Num of pregnancies', 'Smokes (years)', 'Smokes (packs/year)', 'Hormonal Contraceptives (years)', 'IUD (years)', and 'STDs (number)'. These columns likely contain continuous or count-based numerical values.

On the other hand, the categorical_df list consists of columns representing categorical data, such as 'Smokes', 'Hormonal Contraceptives', 'IUD', 'STDs', 'STDs:condylomatosis', 'STDs:cervical condylomatosis', 'STDs:vaginal condylomatosis', 'STDs:vulvo-perineal condylomatosis', 'STDs:syphilis', 'STDs:pelvic inflammatory disease', 'STDs:genital herpes', 'STDs:molluscum contagiosum', 'STDs:AIDS', 'STDs:HIV', 'STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of diagnosis', 'Dx:Cancer', 'Dx:CIN', 'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller', 'Citology', and 'Biopsy'. These columns likely contain categorical values or binary indicators.

By categorizing columns in this manner, the code aims to facilitate efficient and organized processing of the dataset based on the data types of its features, enabling tailored operations and analysis for different types of features.

Step 2 Fill the missing values of numeric data columns with mean of the column data:

```
1 #Fills the missing values of numeric data columns
2 with mean of the column data.
3 for feature in numerical_df:
4     print(feature,",df[feature].apply(pd.to_numeric,\
```

```
5     errors='coerce').mean())
6     feature_mean =
7     round(df[feature].apply(pd.to_numeric, \
8             errors='coerce').mean(),1)
9     df[feature] = df[feature].fillna(feature_mean)
10
11 for feature in categorical_df:
    df[feature] = df[feature].apply(pd.to_numeric, \
        errors='coerce').fillna(1.0)
```

Output:

```
Age 26.82051282051282
Number of sexual partners 2.527644230769231
First sexual intercourse 16.995299647473562
Num of pregnancies 2.275561097256858
Smokes (years) 1.2197214125857985
Smokes (packs/year) 0.45314395064923096
Hormonal Contraceptives (years) 2.2564192013893343
IUD (years) 0.514804318488529
STDs (number) 0.17662682602921648
```

In this code, missing values in the dataset are being handled for both numerical and categorical columns.

For the numerical columns listed in numerical_df, each missing value is being filled with the mean value of that column using the .fillna() method. The code iterates through each feature, first calculating the mean value of the column using `df[feature].apply(pd.to_numeric, errors='coerce').mean()`, where pd.to_numeric is used to convert any non-numeric values to NaN. The mean value is then rounded to one decimal place using round() and stored in the feature_mean variable. Finally, the missing values in the column are replaced with this calculated mean using `df[feature] = df[feature].fillna(feature_mean)`.

For the categorical columns listed in categorical_df, a similar approach is taken to convert any non-numeric values to NaN using df[feature].apply(pd.to_numeric, errors='coerce'), followed by filling missing values with the value 1.0 using .fillna(1.0). This effectively converts the categorical columns into binary indicators (1 for present, 0 for missing), which is a common strategy to handle missing values in categorical data.

Overall, this code ensures that the dataset is properly prepared for subsequent analysis or modeling by imputing missing values in a suitable manner for both numerical and categorical columns.

Impact of Hormonal Contraceptives

Step 1 Use pearson correlation to determine which feature is effect 'Hormonal Contraceptives':

```
1 #Effect 'Hormonal Contraceptives'  
2 corrrmat = df.corr()  
3 k = 15 #number of variables for heatmap  
4 cols = corrrmat.nlargest(k, 'Hormonal  
5 Contraceptives')['Hormonal Contraceptives'].index  
6 cm = df[cols].corr()  
7  
8 plt.figure(figsize=(25,20))  
9  
10 sns.set(font_scale=1.25)  
11 hm = sns.heatmap(cm, cbar=True, cmap='Set1'  
12 ,annot=True,\n13 vmin=0,vmax =1, square=True, fmt='%.2f', \
```

```
14     annot_kws={'size': 10}, yticklabels = cols.values,  
15     \  
16         xticklabels = cols.values)  
17 plt.show()
```

The heatmap is shown in Figure 2. The purpose of this code is to visually analyze the correlation between the 'Hormonal Contraceptives' column and a selected number of other variables in the dataset using a heatmap. This visualization provides insights into the relationships and dependencies between these variables.

First, a correlation matrix `corrmat` is calculated for all the variables in the dataset using the `df.corr()` function. This matrix quantifies the linear relationships between pairs of variables, with values ranging from -1 (perfect negative correlation) to 1 (perfect positive correlation). The code then sets the value of `k` to 15, indicating that the top 15 variables with the highest correlation to 'Hormonal Contraceptives' will be considered for the heatmap.

The code identifies the top `k` correlated variables with 'Hormonal Contraceptives' by using the `.nlargest()` method on the `corrmat` matrix. The `cols` variable stores the column names of these selected variables.

A new correlation matrix `cm` is created using only the selected columns. This narrower matrix reflects the correlations between these variables. Subsequently, a larger plot area is defined using `plt.figure(figsize=(25, 20))` to accommodate the heatmap. The heatmap is generated using the Seaborn library's `sns.heatmap()` function, which displays color-coded cells to represent the correlation values. The annotations within the cells provide additional numerical information about the correlations. This visualization helps to identify patterns and trends in the relationships between the

'Hormonal Contraceptives' column and other variables, enabling a better understanding of how these features interact within the dataset.

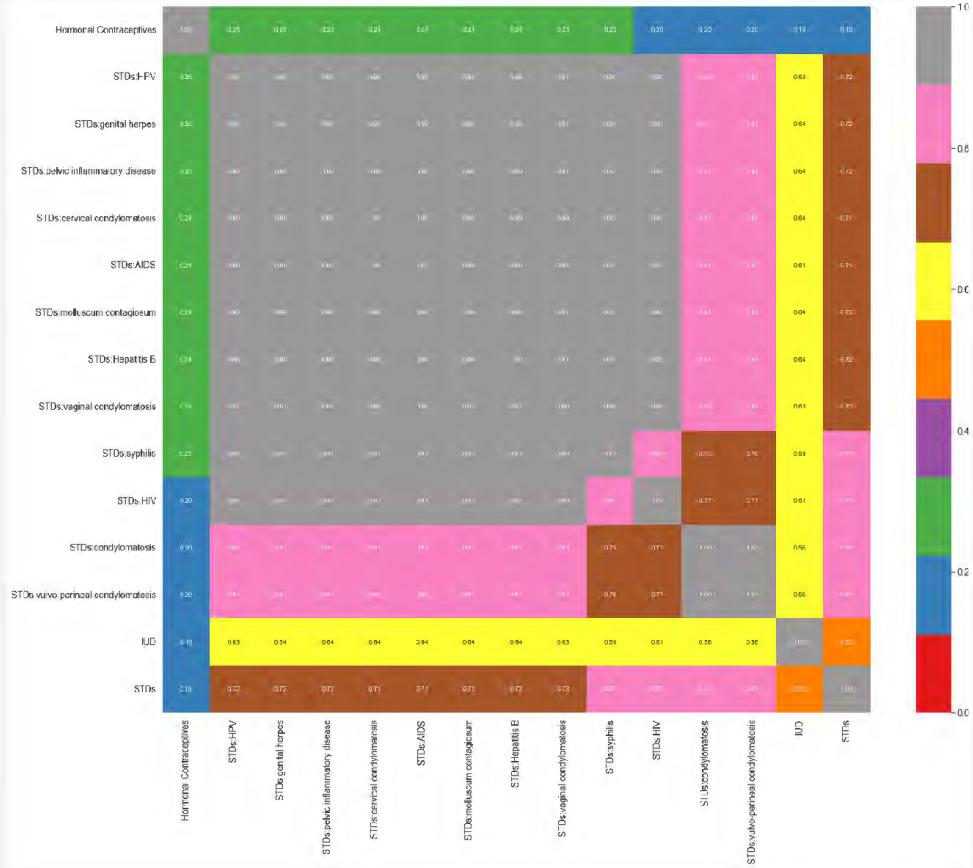


Figure 2 The heatmap showing which feature is the effect of 'Hormonal Contraceptives'

Impact of IUD

Step 1 Use pearson correlation to determine which feature is effect 'IUD':

- 1 #Effect 'IUD'
- 2 corrrmat = df.corr()
- 3 k = 15 #number of variables for heatmap
- 4 cols = corrrmat.nlargest(k, 'IUD')['IUD'].index
- 5 cm = df[cols].corr()

```

6
7     plt.figure(figsize=(25,20))
8
9     sns.set(font_scale=1.25)
10    hm = sns.heatmap(cm,cmap = 'rainbow', cbar=True, \
11                      annot=True,vmin=0,vmax =1, square=True,
12                      fmt='.2f', \
13                      annot_kws={'size': 10},yticklabels = cols.values, \
14                      xticklabels = cols.values)
15
16    plt.show()

```

The heatmap is shown in Figure 3.

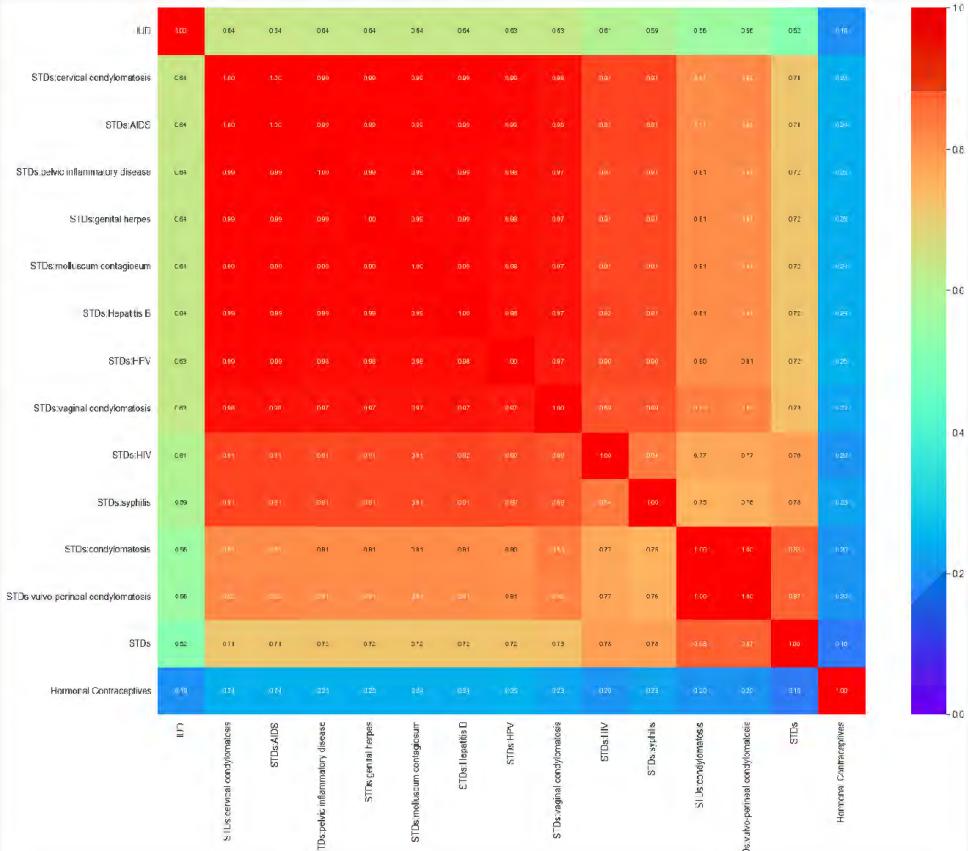


Figure 3 The heatmap showing which feature is the effect of ‘IUD’

The purpose of this code is similar to the previous one, but it focuses on analyzing the correlation between the 'IUD' (Intrauterine Device) column and a selected number of other variables in the dataset using a heatmap.

The code first calculates the correlation matrix `corrmat` for all the variables in the dataset, just like before. The value of `k` is set to 15, indicating that the top 15 variables with the highest correlation to 'IUD' will be considered for the heatmap.

The code identifies the top `k` correlated variables with 'IUD' using the `.nlargest()` method on the `corrmat` matrix. The `cols` variable stores the column names of these selected variables.

A new correlation matrix `cm` is created using only the selected columns. The code then defines a larger plot area using `plt.figure(figsize=(25, 20))` to accommodate the heatmap. The heatmap is generated using the Seaborn library's `sns.heatmap()` function. This time, the colormap is set to 'rainbow' to visually distinguish between different correlation values. The heatmap cells are color-coded to represent the strength and direction of correlations, and annotations provide numerical details. This visualization aids in understanding the relationships between the 'IUD' column and other variables, helping to identify potential patterns and dependencies within the dataset.

Impact of STDs

Step 1 Use pearson correlation to determine which feature is effect 'STDs':

```
1 #Effect 'STDs'  
2 corrmat = df.corr()
```

```
3 k = 15 #number of variables for heatmap
4 cols = corrrmat.nlargest(k, 'STDs')['STDs'].index
5 cm = df[cols].corr()
6
7 plt.figure(figsize=(25,20))
8
9 sns.set(font_scale=1.25)
10 hm = sns.heatmap(cm,cmap = 'hot', cbar=True, \
11     annot=True,vmin=0,vmax =1, square=True,
12     fmt='%.2f', \
13     annot_kws={'size': 10}, yticklabels = cols.values,
14     \
15     xticklabels = cols.values)
16 plt.show()
```

The heatmap is shown in Figure 4. The purpose of this code is once again to analyze the correlation between a specific column, in this case, 'STDs' (Sexually Transmitted Diseases), and a selected number of other variables using a heatmap.

Similar to the previous examples, the code first calculates the correlation matrix `corrrmat` for all the variables in the dataset.

The value of `k` is set to 15, indicating that the top 15 variables with the highest correlation to 'STDs' will be considered for the heatmap.

The code identifies the top `k` correlated variables with 'STDs' using the `.nlargest()` method on the `corrrmat` matrix. The `cols` variable stores the column names of these selected variables.

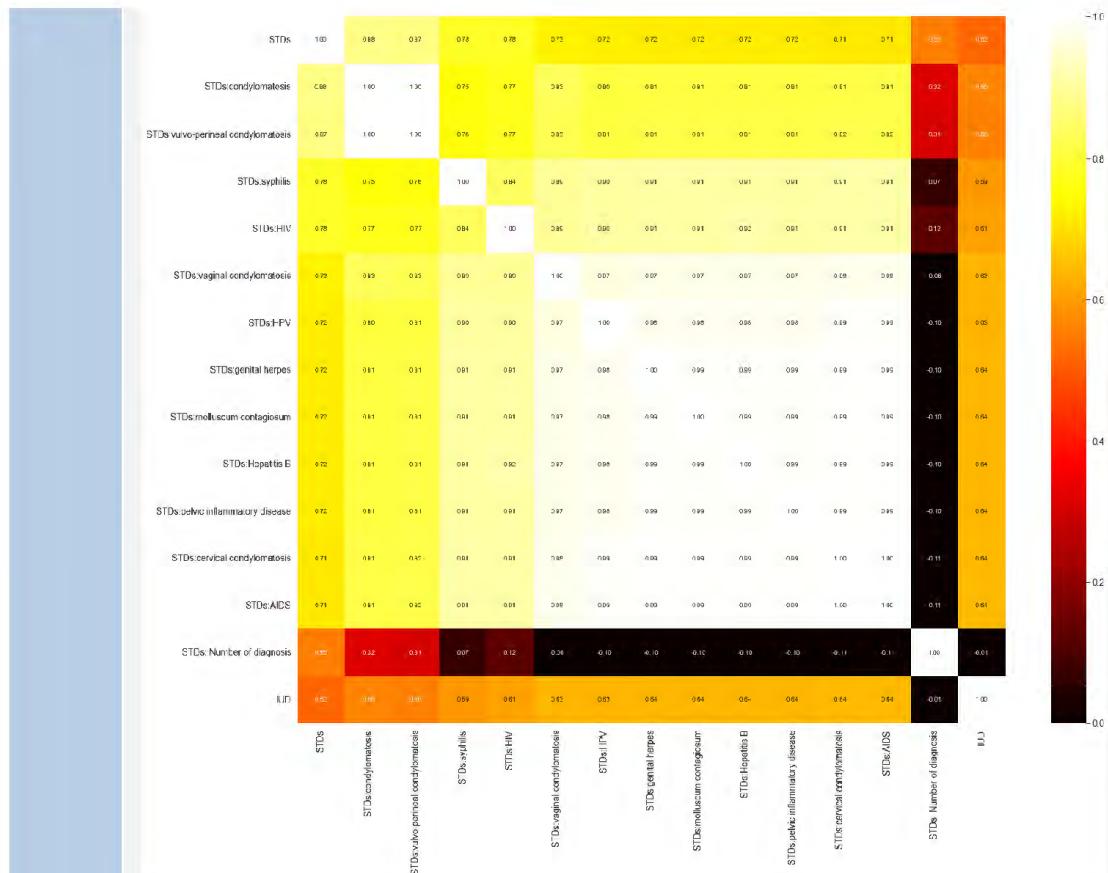


Figure 4 The heatmap showing which feature is the effect of 'STDs'

A new correlation matrix cm is created using only the selected columns.

The code then defines a larger plot area using plt.figure(figsize=(25, 20)) to accommodate the heatmap.

The heatmap is generated using the Seaborn library's sns.heatmap() function. This time, the colormap is set to 'hot' to visually represent correlations. The heatmap cells are color-coded to reflect the strength and direction of correlations, and annotations provide numerical details.

By visualizing the correlation between the 'STDs' column and other variables, this code helps uncover potential relationships and patterns within the dataset related to

sexually transmitted diseases. This type of analysis can provide insights into factors that may be associated with the presence of STDs and guide further investigation or modeling efforts.

Correlation Matrix

Step Plot correlation matrix for each feature:

1

```
1 #Correlation Matrix for each feature
2 cormat = round(df.corr(), 2)
3 top_corr_features = cormat.index
4 plt.figure(figsize=(25,20))
5 #plot heat map
6 g=sns.heatmap(df[top_corr_features].corr(),annot=True,\n7     fmt='.{2f}',cmap="YlGnBu")
```

The correlation matrix of each feature is shown in Figure 5. The purpose of this code is to generate a comprehensive correlation matrix heatmap for all features in the dataset. This visualization allows for a broader understanding of the relationships between different variables.

The code calculates the correlation matrix `cormat` for all features in the dataset using the `.corr()` function. The `round()` function is used to round the correlation values to two decimal places for clarity.

The variable `top_corr_features` stores the index of the columns in the `cormat` dataframe, representing all the features.

The `plt.figure(figsize=(25,20))` line sets the size of the plot to ensure a clear and visually appealing heatmap.

The Seaborn library's `sns.heatmap()` function is then used to create the heatmap. The heatmap includes annotations (`annot=True`) to display the correlation values within each cell, and the `.2f` format specifier is used to format the displayed values to two decimal places. The colormap is set to "YlGnBu," which transitions from yellow to green to blue, indicating lower to higher correlation values.

This heatmap provides a holistic view of the correlations between all pairs of features in the dataset. It helps to identify patterns, dependencies, and potential multicollinearity between variables. Such insights can guide feature selection, data preprocessing, and modeling decisions in data analysis and machine learning tasks.

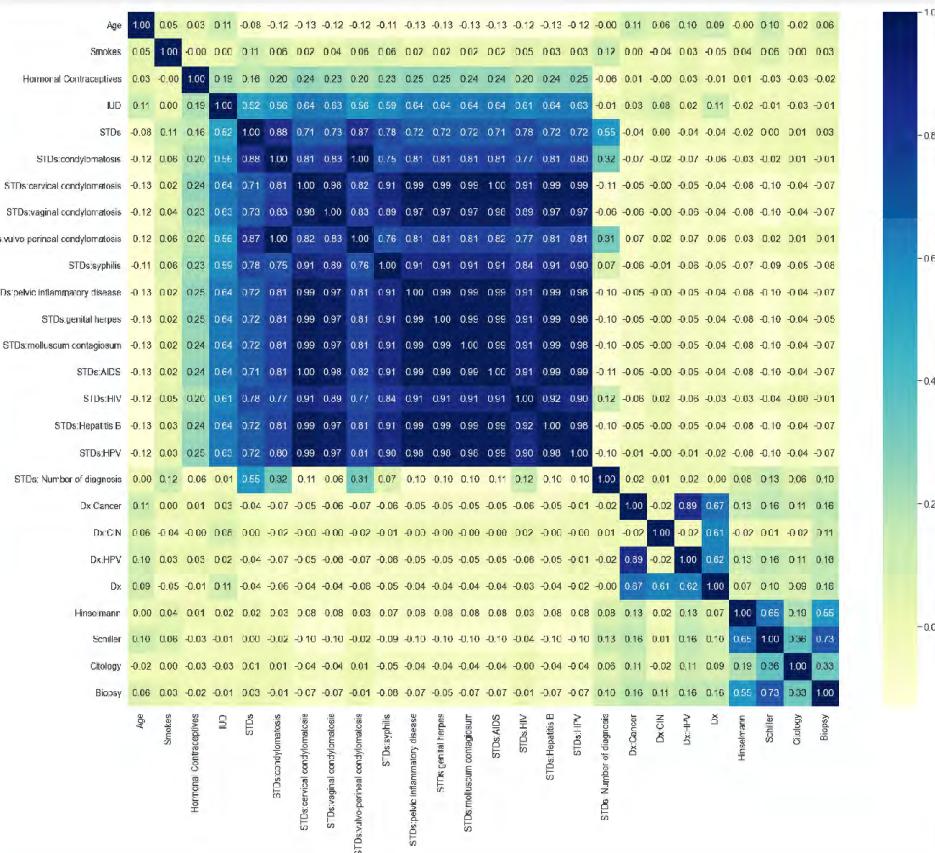


Figure 5 The correlation matrix of each feature in dataset

Distribution of Target Variable

Step 1 Plot distribution of target variable (**Biopsy**) in pie chart:

```
1 # Defines function to create pie chart and bar plot as
2 subplots
3 def plot_pie_barchart(df, var, title=""):
4     plt.figure(figsize=(25, 10))
5
6     # Pie Chart (Left Subplot)
7     plt.subplot(121)
8     label_list = list(df[var].value_counts().index)
9     colors = sns.color_palette("Set2", len(label_list))
10    # Use the 'pastel' color palette
11    _, _, autopcts = plt.pie(df[var].value_counts(),
12    autopct="%1.1f%%", colors=colors,
13                           startangle=60,
14                           labels=label_list,
15                           wedgeprops={"linewidth": 2,
16                           "edgecolor": "white"}, # Add white edge
17                           shadow=True, textprops=
18                           {'fontsize': 20})
19    plt.title("Distribution of " + var + " variable " +
20    title, fontsize=25)
21
22    # Extract percentage values from autopcts
23    percentage_values = [float(p.get_text().strip('%'))
24    for p in autopcts]
25
26    # Print percentage values
27    print("Percentage values:")
28    for label, percentage in zip(label_list,
29    percentage_values):
30        print(f"{label}: {percentage:.1f}%")
31
```

```

32 # Bar Plot (Right Subplot)
33 plt.subplot(122)
34 ax = df[var].value_counts().plot(kind="barh",
35 color=colors, alpha=0.8) # Increase opacity (alpha)
36 for i, j in enumerate(df[var].value_counts().values):
37     ax.text(.7, i, j, weight="bold", fontsize=20)
38
39 plt.title("Count of " + var + " cases " + title,
40 fontsize=25)

# Print count values
value_counts = df[var].value_counts()
print("Count values:")
print(value_counts)
plt.show()

plot_pie_barchart(df,'Biopsy')

```

The result is shown in Figure 6.

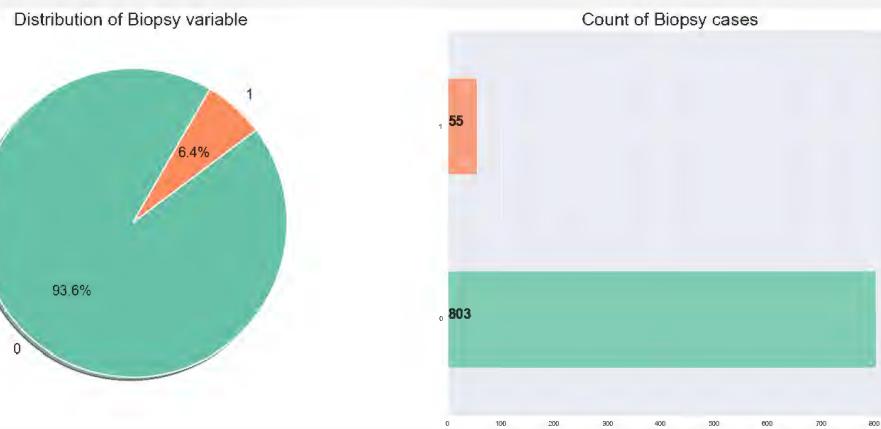


Figure 6 The distribution of target variable (**Biopsy**)

The code defines a function called `plot_pie_barchart()` that creates a side-by-side visualization of a pie chart and a horizontal bar plot to analyze the distribution and count of a categorical variable in a given DataFrame (df). The function takes three main inputs: the DataFrame (df), the variable to be analyzed (var), and an optional title for the plots (title).

In the pie chart (left subplot), the code first extracts the unique labels of the categorical variable using `df[var].value_counts().index`. It then generates a color palette using the Seaborn library's "Set2" color palette. The pie chart is created using `plt.pie()` with various settings for appearance, such as the start angle, label placement, and shadow. The percentage values for each category are extracted from the `autopct` labels and printed to the console.

In the bar plot (right subplot), the code uses `df[var].value_counts().plot(kind="barh")` to create a horizontal bar plot showing the count of each category. Text annotations are added to each bar to display the corresponding count values. The color palette and opacity settings match those used in the pie chart.

Both subplots are displayed side by side using `plt.subplot()`, and titles are added to each subplot. After creating the visualizations, the function prints both the percentage values and the count values for each category to provide a comprehensive summary.

Finally, the function is called with the example of analyzing the distribution and count of the "Biopsy" variable in the DataFrame `df`. This function is useful for quickly visualizing the distribution and count of categorical variables, making it easier to understand and communicate insights from the data.

Output:

Percentage values:

0: 93.6%

1: 6.4%

Count values:

0 803

1 55

Name: Biopsy, dtype: int64

The analysis of the "Biopsy" variable using the plot_pie_barchart() function reveals important insights about the dataset:

- Class Imbalance: The distribution of the "Biopsy" variable is highly imbalanced, with approximately 93.6% of cases having a negative biopsy result (0) and only about 6.4% having a positive biopsy result (1). This indicates a significant disparity in the representation of the two classes, with the negative class being dominant. Such class imbalance can impact the performance of machine learning algorithms, especially those that are sensitive to class distribution.
- Minority Class: The positive biopsy results (1) represent a relatively small minority of the dataset. This suggests that the dataset might be skewed toward negative results, making it more challenging to accurately predict and classify positive biopsy cases. Models trained on imbalanced datasets may struggle to learn patterns from the minority class due to its limited representation.
- Model Considerations: When building predictive models using this dataset, it's important to address the class imbalance issue. Techniques such as resampling (oversampling the minority class, undersampling the majority class), using appropriate evaluation metrics (precision, recall, F1-score), and applying algorithms

that are robust to class imbalance (e.g., ensemble methods, tree-based algorithms) can help mitigate the challenges posed by the imbalanced classes.

In conclusion, the analysis highlights the need to handle the class imbalance in the "Biopsy" variable when developing predictive models for cervical cancer diagnosis. Addressing this imbalance is crucial to ensure that the models provide accurate and reliable predictions for both negative and positive biopsy outcomes.

Distribution of Hinselmann

Step 1 Plot distribution of target variable (**Hinselmann**) in pie chart:

```
1 plot_pie_barchart(df,'Hinselmann')
```

The result is shown in Figure 7.

Output:

Percentage values:

0: 95.9%

1: 4.1%

Count values:

0 823

1 35

Name: Hinselmann, dtype: int64

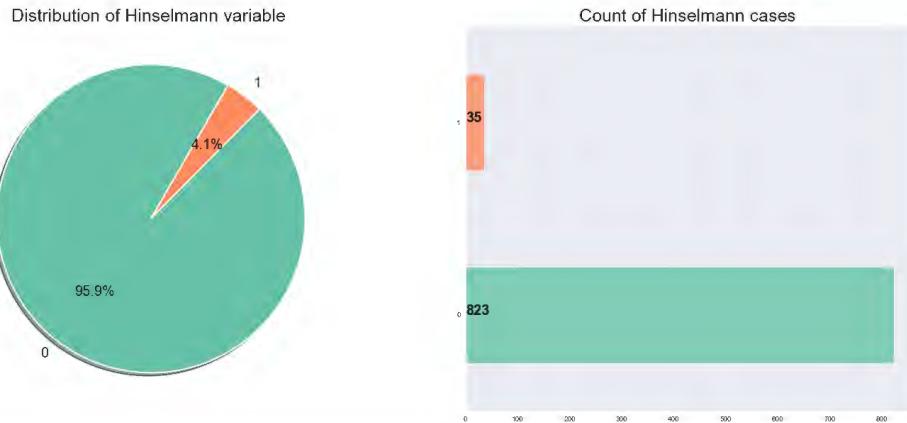


Figure 7 The distribution of target variable (**Hinselmann**)

The analysis of the "Hinselmann" variable using the `plot_pie_barchart()` function provides the following insights:

- **Class Imbalance:** Similar to the "Biopsy" variable, the "Hinselmann" variable also exhibits a significant class imbalance. About 95.9% of cases are classified as negative (0), while only 4.1% are classified as positive (1). This substantial imbalance indicates that the majority class greatly outweighs the minority class.
- **Minority Class:** The positive cases (1) are notably underrepresented in the dataset. This skewness can potentially lead to challenges when training predictive models, as algorithms may struggle to accurately capture patterns from the minority class due to its limited presence.
- **Model Implications:** As with the "Biopsy" variable, addressing the class imbalance in the "Hinselmann" variable is crucial for building reliable and effective predictive models. Techniques like resampling,

appropriate choice of evaluation metrics, and algorithm selection can help mitigate the impact of class imbalance.

In summary, the analysis underscores the importance of handling class imbalance when working with the "Hinselmann" variable. Ensuring a balanced representation of both classes is essential for developing robust models that can accurately predict cases of positive classification while avoiding the pitfalls of biased predictions toward the majority class.

Distribution of Schiller

Step 1 Plot distribution of target variable (**Schiller**) in pie chart:

```
1 plot_pie_barchart(df,'Schiller')
```

The result is shown in Figure 8.

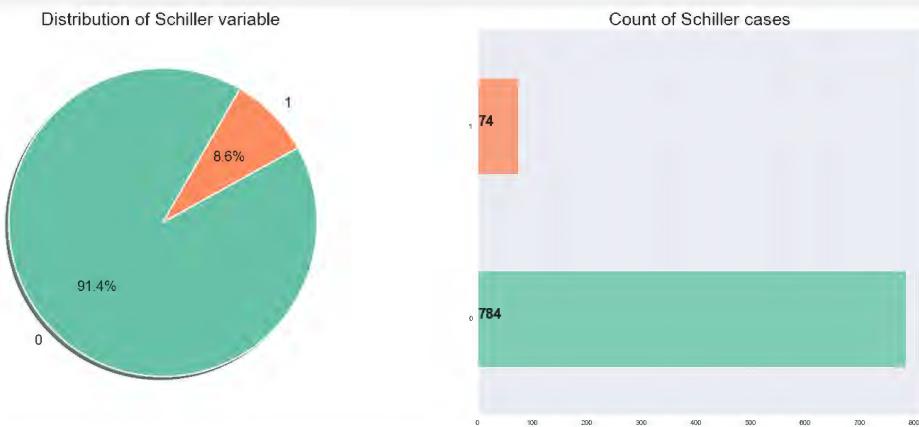


Figure 8 The distribution of target variable (**Schiller**)

Output:

Percentage values:

0: 91.4%

1: 8.6%

Count values:

0 784

1 74

Name: Schiller, dtype: int64

The analysis of the "Schiller" variable reveals that approximately 91.4% of cases are categorized as negative (0), while around 8.6% are classified as positive (1). This distribution indicates a class imbalance, where the majority class dominates the dataset, potentially posing challenges for building accurate predictive models and emphasizing the need for addressing this imbalance to ensure reliable model performance.

Distribution of Citology

Step 1 Plot distribution of target variable (**Citology**) in pie chart:

```
1 plot_pie_barchart(df,'Citology')
```

The result is shown in Figure 9.

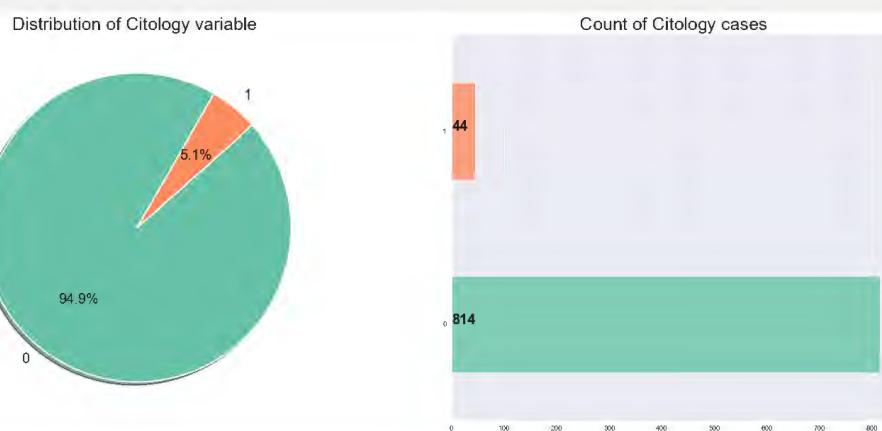


Figure 9 The distribution of target variable (**Citology**)

Output:

Percentage values:

0: 94.9%

1: 5.1%

Count values:

0 814

1 44

Name: Citology, dtype: int64

The examination of the "Citology" variable indicates that approximately 94.9% of instances belong to the negative class (0), while around 5.1% fall into the positive class (1). This distribution suggests another case of class imbalance in the dataset, where the majority class prevails. Similar to previous variables, addressing this imbalance becomes essential to develop robust and accurate predictive models that can provide meaningful insights and make reliable predictions in real-world scenarios.

Distribution of Biopsy versus Other Features

Step 1 Plot distribution of Biopsy versus other features:

```
1 #Plots distribution of number of cases of all other
2 features versus one feature
3 def others_versus_one_feat(columns, one_feat):
4     ROWS = 7
5     COLS = 5
6     fig, ax = plt.subplots(ROWS, COLS, figsize=(80,
7     60), constrained_layout=True, facecolor='#fbe7dd')
8
9     for i in range(ROWS):
10        for j in range(COLS):
11            ax_index = COLS * i + j
```

```

12 if ax_index >= len(columns):
13     break
14
15     g = sns.countplot(data=df,
16                         x=columns[ax_index], hue=one_feat, palette='Set2',
17                         ax=ax[i, j])
18         g.set_xlabel(columns[ax_index], fontsize=50)
19         g.set_ylabel('Number of Cases', fontsize=40)
20         g.tick_params(axis='both', labelsize=30)
21         g.legend(title=one_feat, title_fontsize=50,
22                         fontsize=30)
23
24 for p in g.patches:
25     g.annotate(format(p.get_height(), '.0f'),
26                 (p.get_x() + p.get_width() / 2.,
27                  p.get_height()),
28                 ha='center', va='center', xytext=(0, 10),
29                 weight='bold', fontsize=30,
30                 textcoords='offset points')
31
32 plt.show()

#Plots distribution of count of other features versus
Biopsy
columns=list(df.columns)
columns.remove('Biopsy')
others_vs_one_feat(columns, "Biopsy")

```

The result is shown in Figure 10.

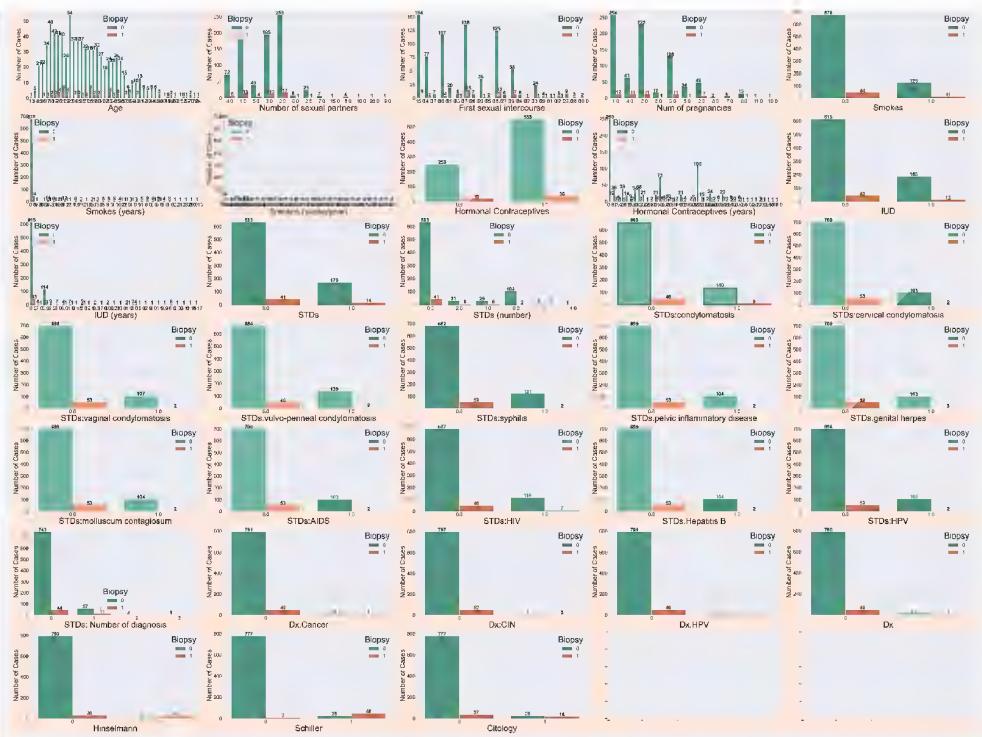


Figure 10 The distribution of six categorical features versus **Biopsy** variable

The code defines a function named `others_vs_one_feat()` that generates a set of count plots, each showcasing the distribution of the number of cases for different categorical features against a specified target feature. In this context, the target feature is "Biopsy," which appears to be a binary classification indicating the presence or absence of a medical condition. The function iterates over the categorical features and creates a grid of count plots, displaying the distribution of cases for each categorical feature based on the "Biopsy" class. The count plots utilize the Seaborn library to visualize the data.

The resulting grid of count plots provides insights into the relationship between various categorical features and the likelihood of a positive or negative biopsy outcome. Each plot illustrates the distribution of cases for a specific categorical feature, with different bars representing the count of cases for the two classes of the "Biopsy" variable (0 and 1).

1). This visualization helps to assess how different categorical features may be associated with the biopsy outcome and whether any significant patterns or trends emerge. The function enhances the understanding of potential correlations between categorical attributes and the target variable, enabling researchers or analysts to identify noteworthy relationships and patterns in the dataset.

By plotting the distribution of each feature against the "Biopsy" outcome, the code offers a comprehensive visual exploration of how these features may influence the likelihood of a positive or negative biopsy result. This information could be valuable for medical professionals and researchers aiming to uncover important factors that contribute to the prediction of biopsy outcomes, which could subsequently guide clinical decisions and patient care.

Distribution of Hinselmann versus Other Features

Step 1 Plot distribution of other features versus **Hinselmann**:

```
1 #Plots distribution of count of other features versus
2 Hinselmann
3 columns=list(df.columns)
4 columns.remove('Hinselmann')
others_vsone_feat(columns, "Hinselmann")
```

The result is shown in Figure 11.



Figure 11 The distribution of six categorical features versus **Hinselmann** variable

The code continues its analysis using the others_vs_one_feat() function to create count plots showcasing the distribution of case counts for different features in relation to the "Hinselmann" target. These plots offer insights into potential correlations between features and the binary "Hinselmann" outcome. By visually comparing case counts across features and classes of the target, the plots provide a comprehensive understanding of how various attributes may influence "Hinselmann" predictions, supporting medical decision-making and facilitating insights into factors impacting the outcome.

Distribution of Schiller versus Other Features

Step 1 Plot distribution of six categorical features versus **Schiller**:

```

1 #Plots distribution of count of other features versus
2 Schiller
3 columns=list(df.columns)
4 columns.remove('Schiller')
others_vs_one_feat(columns, "Schiller")

```

The result is shown in Figure 12.



Figure 12 The distribution of six categorical features versus Schiller variable

The code utilizes the `others_vs_one_feat()` function to generate a series of count plots illustrating the distribution of case counts for different features concerning the "Schiller" target. By excluding the "Schiller" feature from the analysis, the code aims to examine potential relationships between various attributes and the binary "Schiller" outcome. The resulting plots offer a visual representation of how different features contribute to predictions related to "Schiller," providing valuable insights into the interplay between these attributes and the target variable.

Each count plot in the grid displays the distribution of case counts across different levels of a specific feature, segmented by the "Schiller" classes. This graphical approach allows for a quick and intuitive comparison of how each feature's different categories relate to the presence or absence of "Schiller." The count values are labeled on the bars, providing precise numerical information for each combination of feature and outcome. These visualizations enable medical professionals and analysts to identify patterns and potential correlations between specific attributes and the likelihood of "Schiller," facilitating a deeper understanding of the factors influencing the medical condition and supporting evidence-based decision-making.

Distribution of Citology versus Other Features

Step 1 Plot distribution of six categorical features versus **Citology**:

```
1 #Plots distribution of count of other features versus
2 Citology
3 columns=list(df.columns)
4 columns.remove('Citology')
others_vs_one_feat(columns, "Citology")
```

The result is shown in Figure 13.



Figure 13 The distribution of six categorical features versus **Citology** variable

The code snippet utilizes the `others_vs_one_feat()` function to create a series of count plots that illustrate the distribution of case counts for different features concerning the "Citology" target variable. By excluding the "Citology" feature from the analysis, the code aims to investigate the relationship between various attributes and the binary "Citology" outcome. The resulting visualizations provide insights into how different features contribute to predictions related to "Citology," helping to uncover potential connections between these attributes and the target variable.

Each count plot in the grid showcases the distribution of case counts across different categories of a specific feature, categorized by the "Citology" classes. This visual representation allows for a quick and intuitive comparison of how different levels of each feature relate to the presence or absence of "Citology." The height of the bars on the plots corresponds to the count values, offering precise numerical

information for each combination of feature and outcome. These visualizations empower medical professionals and analysts to identify potential patterns and associations between specific attributes and the likelihood of "Citology." By providing a visual understanding of the factors influencing this medical condition, the count plots aid in making informed decisions and furthering the comprehension of the underlying data patterns.

Distribution of Hormonal Contraceptives and Smokes versus Biopsy

Step 1 Plots distribution of **Hormonal Contraceptives** and **Smokes** versus **Biopsy** in pie chart:

```
1 #Plots distribution of two features versus Biopsy in
2 pie chart
3 def three_feats_vs_biopsy(feat1, feat2):
4     features = [feat1, feat2]
5     _, ax = plt.subplots(2, 2, figsize=(25, 25),
6     facecolor='#fbe7dd')
7
8     for i in range(2):
9         gs = df[df['Biopsy'] == 0]
10        [features[i]].value_counts()
11        ss = df[df['Biopsy'] == 1]
12        [features[i]].value_counts()
13        labels_gs = list(gs.index)
14        labels_ss = list(ss.index)
15
16        ax[i][0].pie(gs, labels=labels_gs, shadow=True,
17        autopct='%.1f%%', textprops={'fontsize': 32})
18        ax[i][0].set_xlabel(features[i] + " feature",
19        fontsize=30)
20
```

```

21     ax[i][1].pie(ss, labels=labels_ss, shadow=True,
22     autopct='%.1f%%', textprops={'fontsize': 32})
23     ax[i][1].set_xlabel(features[i] + " feature",
24     fontsize=30)
25
26     ax[i][0].set_title('Biopsy = 0', fontsize=30)
     ax[i][1].set_title('Biopsy = 1', fontsize=30)

plt.show()

```

#Plots distribution of Hormonal Contraceptives and Smokes versus Biopsy in pie chart
 three_feats_vs_biopsy("Hormonal Contraceptives",
 "Smokes")

The result is shown in Figure 14. The code introduces the function `three_feats_vs_biopsy()`, which serves the purpose of generating a set of visualizations to compare the distribution of two specific features, "Hormonal Contraceptives" and "Smokes," in relation to the binary "Biopsy" outcome. By utilizing pie charts, this function offers an intuitive and informative way to explore the potential impact of these features on the likelihood of a positive "Biopsy" result. The function's execution generates a grid of four pie charts, each representing the distribution of one of the two features for both "Biopsy = 0" (negative result) and "Biopsy = 1" (positive result) cases.

In each set of pie charts, the left chart illustrates the distribution of the chosen feature (either "Hormonal Contraceptives" or "Smokes") for cases where the "Biopsy" result is negative (0). The right chart, on the other hand, displays the distribution of the same feature for cases with a positive "Biopsy" result (1). The proportions of different feature values are visually represented by slices in the pie charts, and the percentage of each category is labeled within

the chart. This enables an immediate comparison between the two "Biopsy" classes regarding the distribution of the selected features.

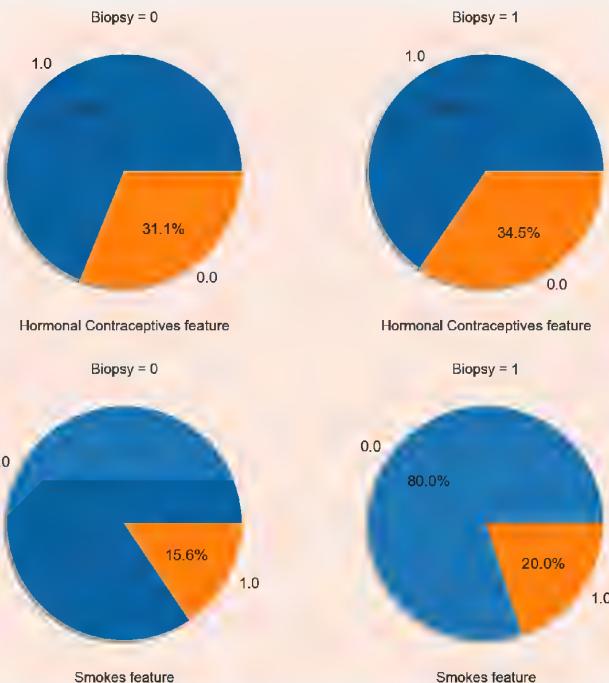


Figure 14 The distribution of **Smokes** and **Hormonal Contraceptives** versus **Biopsy**

The purpose of this visualization is to gain insights into how the presence or absence of certain features, such as "Hormonal Contraceptives" and "Smokes," may be associated with the likelihood of a positive "Biopsy" outcome. By examining these visualizations, medical practitioners and analysts can quickly assess whether these features exhibit significant differences in their distribution across the two "Biopsy" classes. This analysis aids in identifying potential correlations between features and the target variable, supporting informed decision-making and further exploration of relationships within the dataset.

Distribution of IUD and STDs versus Biopsy

Step 1 Plot distribution of **IUD** and **STDs** versus **Biopsy** in pie chart:

```
1 #Plots distribution of IUD and STDs versus Biopsy  
2 in pie chart  
three_feats_vs_biopsy("IUD", "STDs")
```

The result is shown in Figure 15.

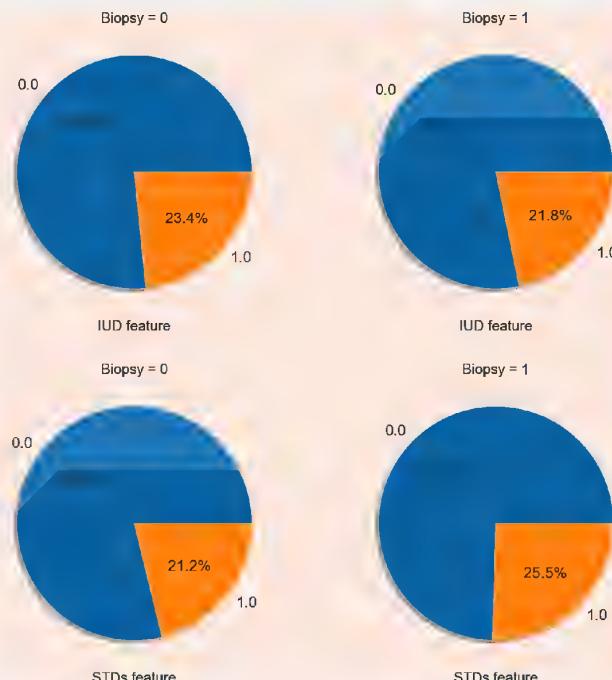


Figure 15 The distribution of **IUD** and **STDs** versus **Biopsy**

The resulting pie chart visualizations depict the distribution of two features, "IUD" (Intrauterine Device) and "STDs" (Sexually Transmitted Diseases), in relation to the binary outcome of "Biopsy." The charts are divided into two sections: on the left side, they show the distribution of the respective feature for cases with a negative "Biopsy" result (0), while on the right side, they display the distribution for cases with a positive "Biopsy" result (1). These charts provide insights into the prevalence of "IUD" and "STDs"

within each "Biopsy" outcome category. By comparing the segments' sizes, it becomes apparent how the presence or absence of these features might be associated with the likelihood of a positive "Biopsy" result. This visualization aids in understanding potential relationships between these features and the medical condition indicated by a positive biopsy, offering valuable insights for further analysis.

Histogram and Density

Step 1 Plot the distribution of six categorical features versus **Age** feature:

```
1 plt.rcParams['figure.dpi'] = 600
2 fig = plt.figure(figsize=(3.7,2), facecolor='#fbe7dd)
3 gs = fig.add_gridspec(3, 2)
4 gs.update(wspace=0.5, hspace=0.75)
5
6 background_color = "#72b7a1"
7 sns.set_palette(['#ff355d','#ffd514'])
8
9 def feat_versus_other(feat,another,legend,ax0,title):
10     for s in ["right", "top"]:
11         ax0.spines[s].set_visible(False)
12
13     ax0.set_facecolor(background_color)
14     ax0_sns = sns.histplot(data=df, x=feat,\n15         ax=ax0,zorder=2,kde=False,hue=another,\n16         multiple="stack",
17         shrink=.8,linewidth=0.3,alpha=1)
18
19     ax0_sns.set_xlabel("",fontsize=4, weight='bold')
20     ax0_sns.set_ylabel("",fontsize=4, weight='bold')
21
```

```
22     ax0_sns.grid(which='major', axis='x', zorder=0, \
23         color='#EEEEEE', linewidth=0.4)
24     ax0_sns.grid(which='major', axis='y', zorder=0, \
25         color='#EEEEEE', linewidth=0.4)
27
28     ax0_sns.tick_params(labelsize=3, width=0.5,
29         length=1.5)
30     ax0_sns.legend(legend, ncol=2,
31         facecolor='#D8D8D8', \
32             edgecolor=background_color, fontsize=3, \
33             bbox_to_anchor=(1, 0.989), loc='upper right')
34     ax0.set_facecolor(background_color)
35     ax0_sns.set_xlabel(title)
36
37 def hist_feat_versus_six_cat(feat,title):
38     ax0 = fig.add_subplot(gs[0, 0])
39     print(df.Smokes.value_counts())
40     feat_versus_other(feat,df.Smokes,\n41         ['Smokes','No Smoke'],ax0,title)
42
43     ax1 = fig.add_subplot(gs[0, 1])
44     print(df.IUD.value_counts())
45     feat_versus_other(feat,df.IUD,['IUD','No\n46 IUD'],ax1,title)
47
48     ax2 = fig.add_subplot(gs[1, 0])
49     print(df.STDs.value_counts())
50     feat_versus_other(feat,df.STDs,['STDs','No\n51 STDs'],ax2,title)
52
53     ax3 = fig.add_subplot(gs[1, 1])
54     print(dff['Hormonal Contraceptives'].value_counts())
55     feat_versus_other(feat,\n56         df['Hormonal Contraceptives'],\n57         ['Hormonal','No Hormonal'],ax3,title)
58
```

```

59     ax4 = fig.add_subplot(gs[2, 0])
60     print(df.Dx.value_counts())
61     feat_versus_other(feat,df.Dx,['Dx','No
62     Dx'],ax4,title)
63
64     ax5 = fig.add_subplot(gs[2, 1])
65     print(df.Biopsy.value_counts())
66     feat_versus_other(feat,df.Biopsy,\n
67     ['Biopsy=1','Biopsy=0'],ax5,title)
68
69     hist_feat_versus_six_cat(df.Age,"Age")

```

The result is shown in Figure 16.

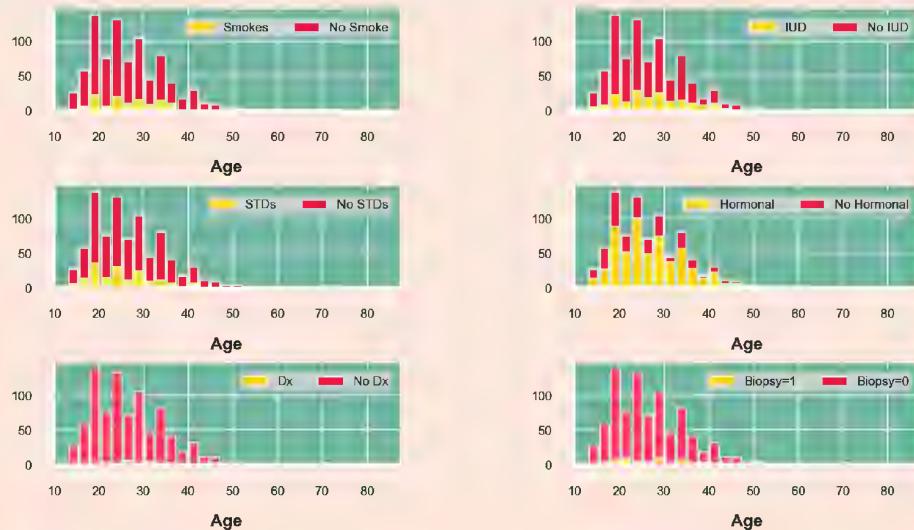


Figure 16 The distribution of six categorical features versus **Age** feature

The code generates a set of six subplots in a single figure, each displaying a histogram that compares the distribution of a specific feature (given by the "feat" variable) across different categories of another feature. The code begins by configuring the figure size, gridspec layout, and color palette for the plots.

The `feat_versus_other()` function is defined to create a histogram plot using Seaborn's `histplot`. This function takes inputs such as the primary and secondary features, legend labels, axis, and title. It sets the appearance of the plot, adds gridlines, customizes tick parameters, and adds a legend to each subplot.

The `hist_feat_versus_six_cat()` function iterates through six categories of features: "Smokes," "IUD," "STDs," "Hormonal Contraceptives," "Dx," and "Biopsy." For each category, it creates a subplot using the `feat_versus_other` function, specifying the main feature ("Age" in this case) and the current category feature. This results in a matrix of histograms, where each subplot provides insights into how the distribution of the selected feature varies across different categories of the categorical feature.

For example, the "Age" feature's distribution is compared across categories like "Smokes" (Smoking Status), "IUD" (Intrauterine Device Usage), "STDs" (Sexually Transmitted Diseases), "Hormonal Contraceptives," "Dx" (Diagnostics), and "Biopsy" results. The histograms reveal how the age distribution differs for each subgroup, providing a comprehensive view of the relationships between different categorical variables and the age distribution. This analysis helps identify potential trends and patterns that can contribute to a better understanding of the dataset.

Step 2 Plot the density of six categorical features versus **Age** feature:

```
1 def
2 prob_feat_versus_other(feat,another,legend,ax0,title):
3     for s in ["right", "top"]:
4         ax0.spines[s].set_visible(False)
5
```

```

6     ax0.set_facecolor(background_color)
7     ax0_sns = sns.kdeplot(x=feat,ax=ax0,hue=another,\n
8
9     linewidth=0.3,fill=True,cbar='g',zorder=2,alpha=1,\n
10    multiple='stack')
11
12    ax0_sns.set_xlabel("",fontsize=4, weight='bold')
13    ax0_sns.set_ylabel("",fontsize=4, weight='bold')
14
15    ax0_sns.grid(which='major', axis='x', zorder=0, \
16      color='#EEEEEE', linewidth=0.4)
17    ax0_sns.grid(which='major', axis='y', zorder=0, \
18      color='#EEEEEE', linewidth=0.4)
19
20    ax0_sns.tick_params(labelsize=3, width=0.5,
21      length=1.5)
22    ax0_sns.legend(legend, ncol=2,
23      facecolor='#D8D8D8', \
24        edgecolor=background_color, fontsize=3, \
25        bbox_to_anchor=(1, 0.989), loc='upper right')
27    ax0.set_facecolor(background_color)
28    ax0_sns.set_xlabel(title)
29
30 def prob_feat_versus_six_cat(feat,title):
31     fig.clf()
32     ax0 = fig.add_subplot(gs[0, 0])
33     print(df.Smokes.value_counts())
34     prob_feat_versus_other(feat,df.Smokes,\n
35       ['Smokes','No Smoke'],ax0,title)
36
37     ax1 = fig.add_subplot(gs[0, 1])
38     print(df.IUD.value_counts())
39
40
41
42

```

43
44
45
46
47
48
49
50
51
52
53
54
55
56

```

prob_feat_versus_other(feat,df.IUD,['IUD','No
IUD'],ax1,title)

    ax2 = fig.add_subplot(gs[1, 0])
    print(df.STDs.value_counts())
    prob_feat_versus_other(feat,df.STDs,\

        ['STDs','No STDs'],ax2,title)

    ax3 = fig.add_subplot(gs[1, 1])
    print(dff['Hormonal Contraceptives'].value_counts())
    prob_feat_versus_other(feat,dff['Hormonal
    Contraceptives'],\
        ['Hormonal','No Hormonal'],ax3,title)

    ax4 = fig.add_subplot(gs[2, 0])
    print(df.Dx.value_counts())
    prob_feat_versus_other(feat,df.Dx,['Dx','No
Dx'],ax4,title)

    ax5 = fig.add_subplot(gs[2, 1])
    print(df.Biopsy.value_counts())
    prob_feat_versus_other(feat,df.Biopsy,['Biopsy=1',\
'Biopsy=0'],ax5,title)

prob_feat_versus_six_cat(df.Age,"Age")

```

The result is shown in Figure 17. The code snippet enhances the previous visualization by generating probability density estimation (KDE) plots. The `prob_feat_versus_other` function creates a KDE plot using Seaborn's `kdeplot`. Similar to the previous function, it takes inputs such as the primary and secondary features, legend labels, axis, and title. This function sets the appearance of the plot, including gridlines, tick parameters, and legends.

The `prob_feat_versus_six_cat()` function iterates through the same six categorical features: "Smokes," "IUD," "STDs," "Hormonal Contraceptives," "Dx," and "Biopsy." For each category, it creates a subplot using the `prob_feat_versus_other()` function, specifying the main feature ("Age" in this case) and the current category feature. The resulting subplots display KDE plots that show the probability density estimation of the selected feature's distribution across different categories of the categorical feature. This enables a more detailed comparison of the distribution shapes and overlaps between the feature and each category.

For example, the "Age" feature's KDE distribution is compared across categories like "Smokes," "IUD," "STDs," "Hormonal Contraceptives," "Dx," and "Biopsy" results. The KDE plots provide insights into the probability distribution

of age for each subgroup, highlighting potential differences and patterns. This analysis offers a deeper understanding of how the distribution of the selected feature varies within different categorical contexts, helping to uncover potential relationships and trends within the dataset.

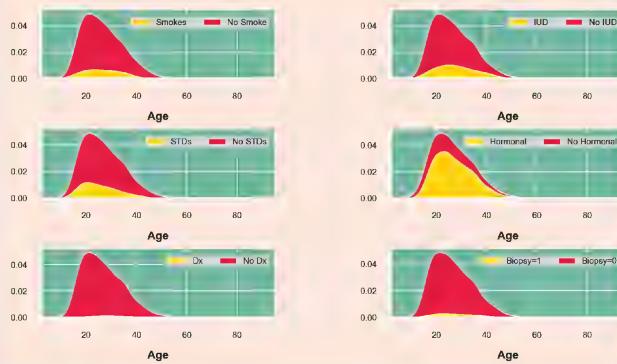


Figure 17 The density of six categorical features versus **Age** feature

Step 3

Plot the distribution of six categorical features density versus **Number of sexual partners** feature:

```

1 hist_feat_vs_six_cat(df["Num of
2 pregnancies"].apply(pd.to_numeric,
3 errors='coerce').convert_dtypes(),'Num of
4 pregnancies')
5 prob_feat_vs_six_cat(df["Num of
pregnancies"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Num of
pregnancies')

```

The results are shown in Figure 18 and 19. The code uses the `hist_feat_vs_six_cat()` function to generate histograms and KDE plots for the "Num of pregnancies" feature across six categorical features: "Smokes," "STDs," "Hormonal Contraceptives," "Dx," and "Biopsy". The feature values are first converted to numeric using `pd.to_numeric` with error handling, and then converted to the appropriate data type using `convert_dtypes()` to handle missing values and non-numeric entries. This ensures that the "Num of pregnancies" feature is correctly analyzed.

Subsequently, the same analysis is performed using the `prob_feat_vs_six_cat()` function, which generates probability density estimation (KDE) plots for the "Num of pregnancies" feature across the same six categorical features. These KDE plots provide a visual representation of the distribution of the "Num of pregnancies" feature within each category, allowing for a deeper understanding of how the feature's distribution varies across different contexts.

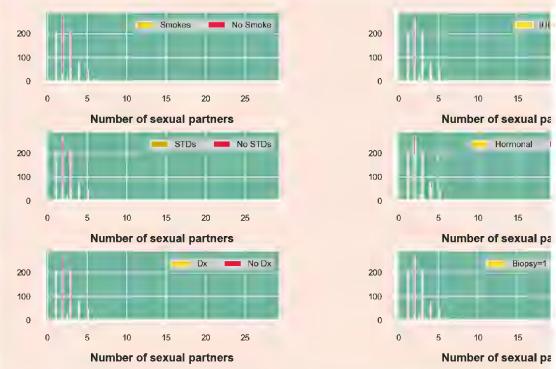


Figure 18 The distribution of six categorical features
Number of sexual partners feature

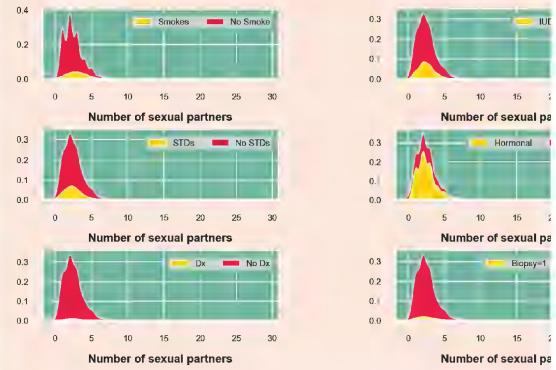


Figure 19 The density of six categorical features
Number of sexual partners feature

By employing both histogram and KDE plots, this snippet enables a comprehensive exploration of the pregnancies' feature's distribution and its relations with various categorical features. This analysis can reveal into potential correlations, trends, and patterns between number of pregnancies and other categorical variables in the dataset.

Step 4

Plot the distribution of six categorical features density versus **Num of pregnancies** feature:

```

1 hist_feat_vs_six_cat(df["Num of
2 pregnancies"].apply(
3     pd.to_numeric, errors='coerce').convert_dt_
4     'Num of pregnancies')
5 prob_feat_vs_six_cat(df["Num of
6 pregnancies"].apply(
    pd.to_numeric, errors='coerce').convert_dt_
    'Num of pregnancies')
```

The results are shown in Figure 20 and 21.

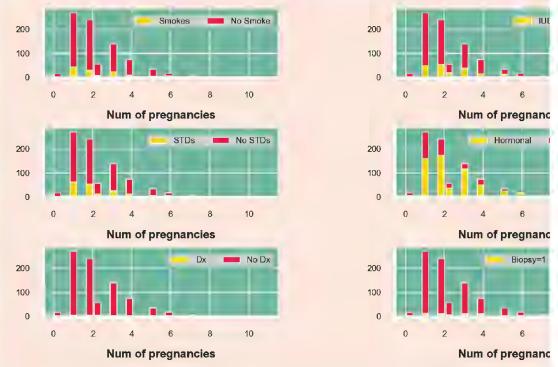


Figure 20 The distribution of six categorical features
Num of pregnancies feature

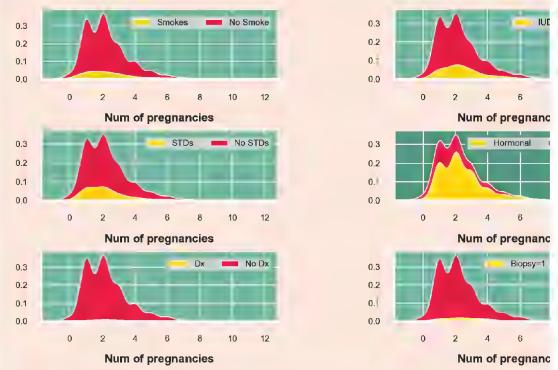


Figure 21 The density of six categorical features
Num of pregnancies feature

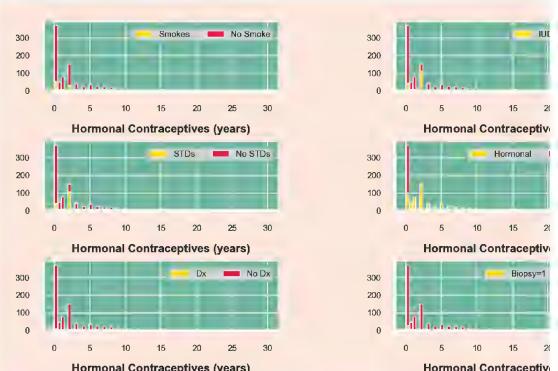


Figure 22 The distribution of six categorical features
Hormonal Contraceptives (years) feature

Step 5

Plot the distribution of six categorical features density versus **Hormonal Contraceptives (years)** f

```
1 hist_feat_vs_six_cat(\n2   df["Hormonal Contraceptives (years)"].app
```

```

3 pd.to_numeric, errors='coerce').convert_dt
4 'Hormonal Contraceptives (years)')
5 prob_feat_versus_six_cat(
6 df["Hormonal Contraceptives (years)"].apply(
7 pd.to_numeric, errors='coerce').convert_dt
8 'Hormonal Contraceptives (years)')

```

The results are shown in Figure 22 and 23.

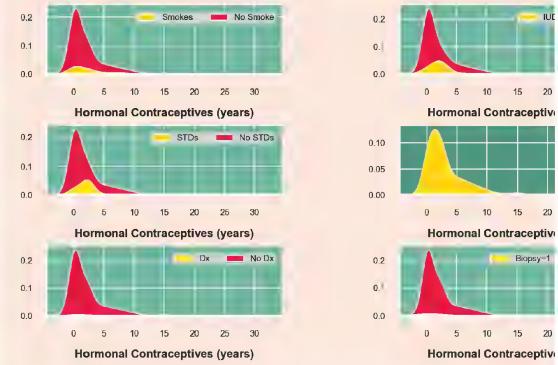


Figure 23 The density of six categorical features
Hormonal Contraceptives (years) feature

Step 6

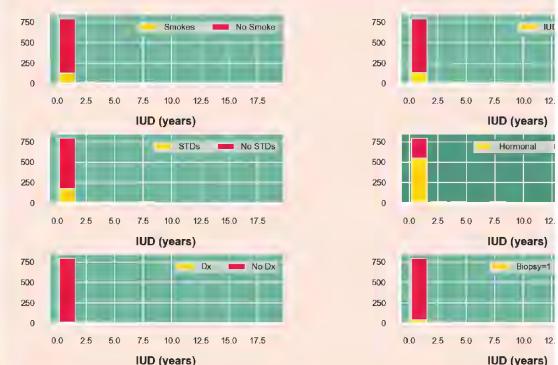
Plot the distribution of six categorical features density versus **IUD (years)** feature:

```

1 hist_feat_versus_six_cat(
2     df["IUD (years)"].apply(
3         pd.to_numeric, errors='coerce').convert_dt
4     'IUD (years)')
5 prob_feat_versus_six_cat(
6     df["IUD (years)"].apply(
7         pd.to_numeric, errors='coerce').convert_dt
8     'IUD (years)')

```

The results are shown in Figure 24 and 25.



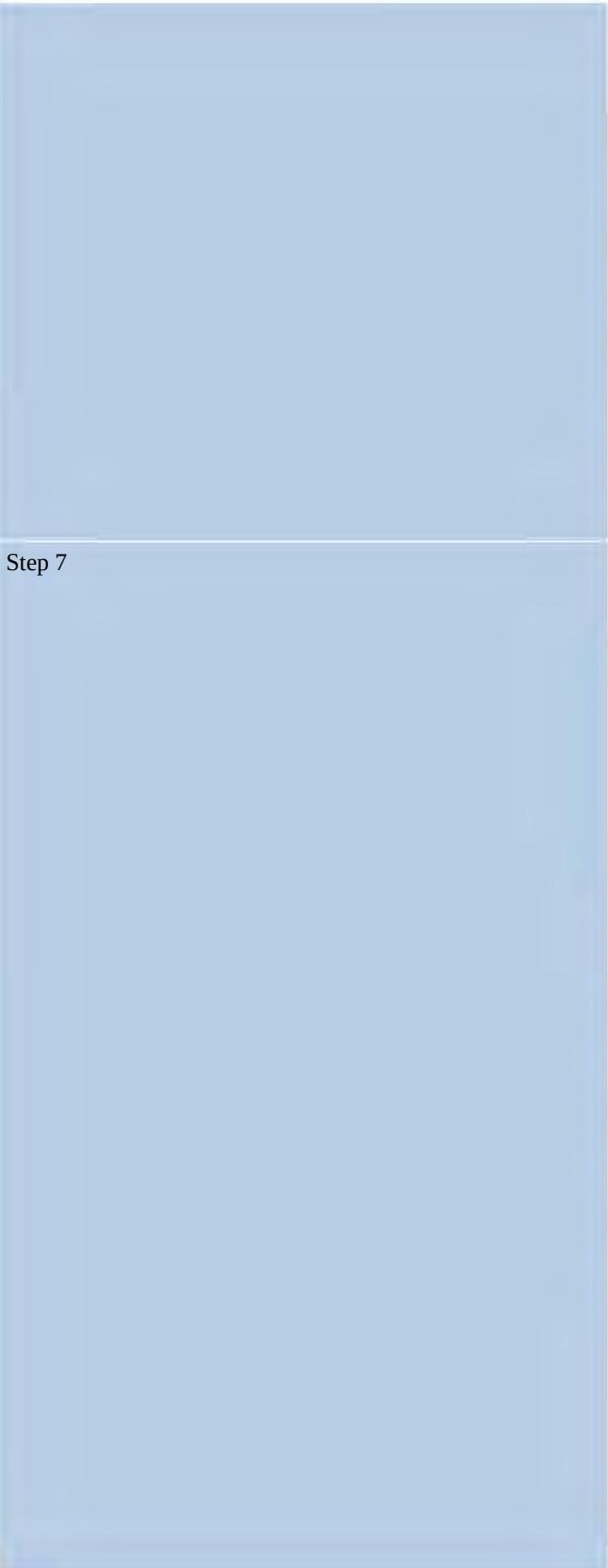


Figure 24 The distribution of six categorical features versus IUD (years) feature

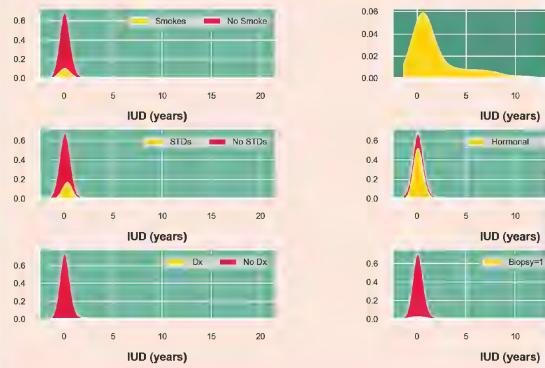


Figure 25 The density of six categorical features versus IUD (years) feature

Plot the distribution of six categorical features versus density versus **Smokes (packs/year)** feature:

```

1 hist_feat_vs_six_cat(df["Smokes
2 (packs/year]").apply(\n3     pd.to_numeric, errors='coerce').convert_dt
4     'Smokes (packs/year)')
5 prob_feat_vs_six_cat(df["Smokes
6 (packs/year]").apply(\n     pd.to_numeric, errors='coerce').convert_dt
     'Smokes (packs/year)')
```

The results are shown in Figure 26 and 27.



Figure 26 The distribution of six categorical features versus Smokes (packs/year) feature

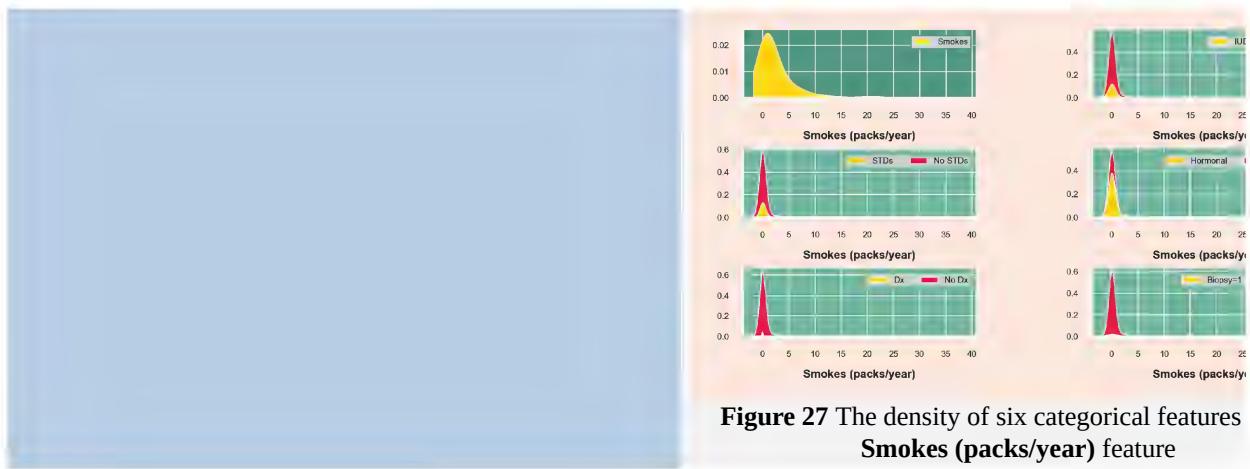


Figure 27 The density of six categorical features
Smokes (packs/year) feature

PREDICTING CERVICAL CANCER USING MACHINE LEARNING PREDICTING CERVICAL CANCER USING MACHINE LEARNING

Features Importance Using Random Forest Classifier

Step 1 Plot feature importance using RandomForest Classifier:

```

1 #Extracts input and output variables
2 X = df.drop('Biopsy', axis =1)
3 y = df["Biopsy"]
4
5 #Feature Importance using RandomForest Classifier
6 names = X.columns
7 rf = RandomForestClassifier()
8 rf.fit(X, y)
9

```

```

10 result_rf = pd.DataFrame()
11 result_rf['Features'] = X.columns
12 result_rf ['Values'] = rf.feature_importances_
13 result_rf.sort_values('Values', inplace = True,
14 ascending = False)
15
16 plt.figure(figsize=(25,25))
17 sns.set_color_codes("pastel")
18 sns.barplot(x = 'Values',y = 'Features', data=result_rf,
19 color="Blue")
20 plt.xlabel('Feature Importance', fontsize=30)
21 plt.ylabel('Feature Labels', fontsize=30)
22 plt.tick_params(axis='x', labelsize=20)
23 plt.tick_params(axis='y', labelsize=20)
24 plt.show()
25
26 # Print the feature importance table
print("Feature Importance:")
print(result_rf)

```

The result is shown in Figure 28.

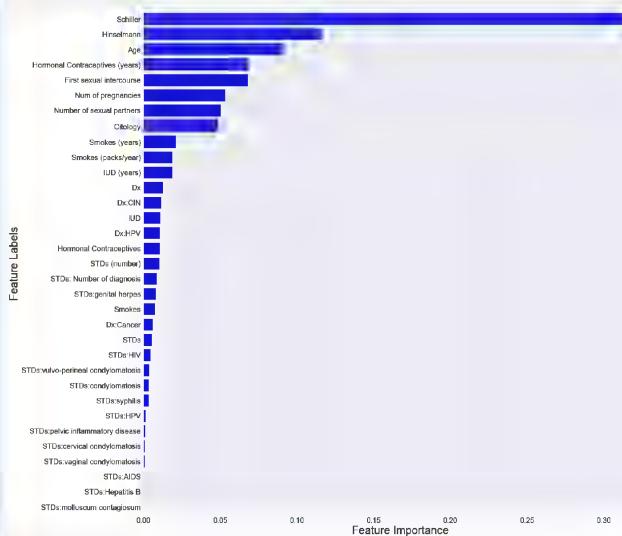


Figure 28 The feature importance using Random Forest classifier

The purpose of this code is to perform feature importance analysis using a RandomForest Classifier on a given dataset and then visualize the importance scores of each feature using a bar plot. Let's break down the code step by step:

1. Data Extraction: The code begins by separating the input features and the target variable from the dataset. X contains all the input features except for the "Biopsy" column, and y contains the values of the "Biopsy" column.

2. Feature Importance Calculation: A RandomForest Classifier (rf) is created and trained using the input features (X) and the target variable (y). RandomForest is an ensemble learning method that constructs multiple decision trees during training and combines their outputs to make predictions.
3. Creating the DataFrame: A DataFrame called result_rf is created to store the feature names and their corresponding importance values obtained from the RandomForest Classifier. The result_rf DataFrame is structured with two columns: "Features" and "Values".
4. Sorting Features: The result_rf DataFrame is sorted in descending order based on the "Values" column, which represents the importance scores of each feature. This sorting provides a ranked view of feature importance, with the most important features at the top.
5. Bar Plot Visualization: A bar plot is created using Seaborn to visualize the feature importance scores. Each bar in the plot represents a feature, and its height corresponds to the importance score. The x-axis shows the importance values, and the y-axis displays the feature names. The bars are color-coded using pastel blue.
6. Plot Customization: The plot is customized by adding labels to the x and y axes, adjusting the font sizes of the labels, and specifying tick sizes.
7. Display the Plot: The bar plot is displayed using plt.show().
8. Print Feature Importance Table: Finally, the code prints the feature importance table (result_rf) to the console. This table provides a tabular view of feature names and their importance scores, complementing the visual representation in the bar plot.

The overall purpose of this code is to provide insights into which features have the most influence in predicting the "Biopsy" outcome using a RandomForest Classifier. By visualizing and analyzing feature importance, you can identify the most relevant features in your dataset and

potentially use this information for feature selection, model interpretation, or data-driven decision making.

Output:

Feature Importance:

	Features	Values
31	Schiller	0.318892
30	Hinselmann	0.116280
0	Age	0.091285
8	Hormonal Contraceptives (years)	0.069005
2	First sexual intercourse	0.068288
3	Num of pregnancies	0.053371
1	Number of sexual partners	0.050463
32	Citology	0.048653
5	Smokes (years)	0.021212
6	Smokes (packs/year)	0.018990
10	IUD (years)	0.018709
29	Dx	0.012772
27	Dx:CIN	0.011521
9	IUD	0.011159
28	Dx:HPV	0.010888
7	Hormonal Contraceptives	0.010817
12	STDs (number)	0.010476
25	STDs: Number of diagnosis	0.008634
19	STDs:genital herpes	0.008242
4	Smokes	0.007539
26	Dx:Cancer	0.006095
11	STDs	0.005457
22	STDs:HIV	0.004623
16	STDs:vulvo-perineal condylomatosis	0.003786
13	STDs:condylomatosis	0.003725
17	STDs:syphilis	0.003385
24	STDs:HPV	0.001399
18	STDs:pelvic inflammatory disease	0.001196
14	STDs:cervical condylomatosis	0.000987
15	STDs:vaginal condylomatosis	0.000798
21	STDs:AIDS	0.000596
23	STDs:Hepatitis B	0.000503
20	STDs:molluscum contagiosum	0.000253

The output displays the feature importance scores calculated using a RandomForest Classifier on the given dataset. Each row represents a feature, and the "Values" column indicates the importance score assigned to that feature. Let's analyze the output and draw conclusions:

1. Highly Important Features:

- "Schiller" has the highest importance score (0.318892). This suggests that the "Schiller" feature is a strong predictor of the target variable ("Biopsy").
- "Hinselmann" follows with a relatively high importance score (0.116280). It

also appears to have a significant impact on the target variable.

2. Moderately Important Features:

- "Age" has a moderate importance score (0.091285), indicating that it contributes to predicting the target variable to a reasonable extent.
- "Hormonal Contraceptives (years)" and "First sexual intercourse" have importance scores around 0.07, indicating their relevance.

3. Less Important Features:

Several features have importance scores between 0.05 and 0.03, including "Num of pregnancies," "Number of sexual partners," "IUD (years)," "Smokes (years)," and others. These features contribute, but to a lesser extent compared to the highly and moderately important features.

4. Lowest Important Features:

The features with the lowest importance scores, such as "STDs:HIV," "STDs:genital herpes," "Smokes," and others, have values close to or less than 0.005. These features have minimal impact on predicting the target variable.

5. Conclusion:

- Based on the feature importance scores, "Schiller" and "Hinselmann" are the most influential features for predicting the "Biopsy" outcome. It suggests that the presence or absence of certain conditions indicated by these features has a strong impact on the likelihood of a positive biopsy result.
- Other important features, like "Age," "Hormonal Contraceptives (years)," and "First sexual intercourse," also contribute significantly to the prediction.
- Features with low importance scores may not contribute significantly to the prediction and might be candidates for potential feature selection or further investigation.

Overall, this analysis helps prioritize features for modeling, understand their relative contributions, and potentially improve the model's performance by focusing on the most relevant features. It provides insights into the relationships between features and the target variable, which can guide data-driven decisions and model enhancements.

Features Importance Using Extra Trees Classifier

Step 1 Plot feature importance using ExtraTreesClassifier:

```
1 #Feature Importance using ExtraTreesClassifier
2 model = ExtraTreesClassifier()
3 model.fit(X, y)
4
5 result_et = pd.DataFrame()
6 result_et['Features'] = X.columns
7 result_et ['Values'] = model.feature_importances_
8 result_et.sort_values('Values', inplace=True,
9 ascending =False)
10
11 plt.figure(figsize=(25,25))
12 sns.set_color_codes("pastel")
13 sns.barplot(x = 'Values',y = 'Features',
14 data=result_et, color="red")
15 plt.xlabel('Feature Importance', fontsize=30)
16 plt.ylabel('Feature Labels', fontsize=30)
17 plt.tick_params(axis='x', labelsize=20)
18 plt.tick_params(axis='y', labelsize=20)
19 plt.show()
20
21 # Print the feature importance table
print("Feature Importance:")
print(result_et)
```

The result is shown in Figure 29.

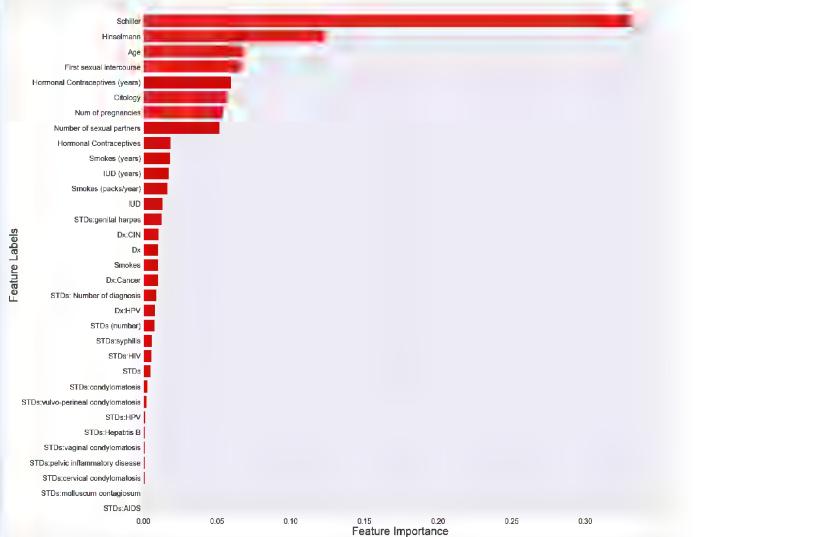


Figure 29 The feature importance using ExtraTreesClassifier

The code calculates and visualizes feature importance using the ExtraTreesClassifier algorithm. In this process, an

ExtraTreesClassifier model is created and trained on the input data X and target variable y. The feature importance scores are then extracted from the model and stored in a DataFrame called result_et, which includes two columns: "Features" to hold the feature names and "Values" to store their respective importance scores. The DataFrame is sorted in descending order based on the feature importance values. Subsequently, a bar plot is generated using Seaborn, where the x-axis represents the feature importance scores, and the y-axis corresponds to the feature names. The bars are colored in red, and the plot is displayed with appropriate labels, tick sizes, and a large figure size. Finally, the script prints the calculated feature importance table.

In essence, this code enables us to gain insights into the significance of different features in predicting the target variable by employing the ExtraTreesClassifier algorithm. By visualizing the feature importance scores, we can readily identify which features contribute the most to the model's predictions. This information aids in making informed decisions about feature selection, enhancing model interpretability, and potentially guiding further data preprocessing or model improvement efforts.

Output:

Feature Importance:

	Features	Values
31	Schiller	0.332101
30	Hinselmann	0.123504
0	Age	0.068701
2	First sexual intercourse	0.066087
8	Hormonal Contraceptives (years)	0.059459
32	Citology	0.057020
3	Num of pregnancies	0.054084
1	Number of sexual partners	0.051548
7	Hormonal Contraceptives	0.018389
5	Smokes (years)	0.018154
10	IUD (years)	0.017353
6	Smokes (packs/year)	0.016244
9	IUD	0.012929
19	STDs:genital herpes	0.012399
27	Dx:CIN	0.010298
29	Dx	0.010035
4	Smokes	0.010021
26	Dx:Cancer	0.009923
25	STDs: Number of diagnosis	0.008751
28	Dx:HPV	0.007917
12	STDs (number)	0.007559
17	STDs:syphilis	0.005901
22	STDs:HIV	0.005566
11	STDs	0.004820
13	STDs:condylomatosis	0.002891
16	STDs:vulvo-perineal condylomatosis	0.002415
24	STDs:HPV	0.001178

```

23      STDs:Hepatitis B  0.000915
15      STDs:vaginal condylomatosis 0.000883
18      STDs:pelvic inflammatory disease 0.000881
14      STDs:cervical condylomatosis 0.000877
20      STDs:molluscum contagiosum 0.000610
21          STDs:AIDS 0.000588

```

The output presents the feature importance values calculated using the ExtraTreesClassifier algorithm. Each row in the table corresponds to a feature, and the "Features" column lists the names of these features. The "Values" column contains the corresponding importance scores, representing the relative contribution of each feature to the model's predictive performance.

By analyzing the feature importance values, we can observe that the "Schiller" feature has the highest importance score of 0.332, indicating its strong influence on the model's predictions. Similarly, "Hinselmann" and "Age" are also significant with importance scores of 0.124 and 0.069, respectively. Other features such as "First sexual intercourse," "Hormonal Contraceptives (years)," and "Citology" also exhibit notable importance values.

This feature importance analysis is crucial for understanding which variables contribute most to the model's decisions. It can guide feature selection, provide insights into the underlying data patterns, and aid in focusing on the most influential features when interpreting the model's outcomes or making decisions based on its predictions.

Features Importance Using RFE

Step Plot feature importance using RFE:

1

```

1 #Feature Importance using RFE
2 from sklearn.feature_selection import RFE
3 model = LogisticRegression()
4 # create the RFE model
5 rfe = RFE(model)
6 rfe = rfe.fit(X, y)
7
8 result_lg = pd.DataFrame()
9 result_lg['Features'] = X.columns
10 result_lg ['Ranking'] = rfe.ranking_
11 result_lg.sort_values('Ranking', inplace=True ,
12 ascending = False)
13
14 plt.figure(figsize=(25,25))
15 sns.set_color_codes("pastel")
16 sns.barplot(x = 'Ranking',y = 'Features',
17 data=result_lg, color="orange")

```

```

18 plt.ylabel('Feature Labels', fontsize=30)
19 plt.tick_params(axis='x', labelsize=20)
20 plt.tick_params(axis='y', labelsize=20)
21 plt.show()
22
print("Feature Ranking:")
print(result_lg)

```

The result is shown in Figure 30. The code demonstrates the use of Recursive Feature Elimination (RFE) technique to determine feature importance. RFE is a feature selection method that recursively removes less important features from a model and ranks them based on their contribution to the model's performance.

In this specific case, the code uses a logistic regression model (imported from `sklearn.linear_model`) and applies RFE to it. The model iteratively removes the least important feature(s) until a specified number of features or a specific performance criterion is met.

The output shows the ranking of features based on their importance. The "Features" column lists the names of the features, and the "Ranking" column assigns a ranking to each feature. A higher ranking indicates lower importance, i.e., features with higher rankings are considered less important.

The resulting plot visualizes the feature rankings using a bar plot. Features are represented on the y-axis, and their corresponding rankings are shown on the x-axis. Features with higher rankings are depicted towards the right side of the plot, while those with lower rankings (higher importance) are on the left.

This analysis helps in identifying the relative importance of features according to the RFE ranking. By selecting the top-ranked features, one can potentially improve model efficiency and reduce overfitting by focusing on the most relevant predictors.





Figure 30 The feature importance using RF

Output:

Feature Ranking:

	Features	Ranking
16	STDs:vulvo-perineal condylomatosis	18
1	Number of sexual partners	17
12	STDs (number)	16
6	Smokes (packs/year)	15
10	IUD (years)	14
13	STDs:condylomatosis	13
0	Age	12
2	First sexual intercourse	11
5	Smokes (years)	10
3	Num of pregnancies	9
21	STDs:AIDS	8
8	Hormonal Contraceptives (years)	7
7	Hormonal Contraceptives	6
14	STDs:cervical condylomatosis	5
4	Smokes	4
18	STDs:pelvic inflammatory disease	3
20	STDs:molluscum contagiosum	2
17	STDs:syphilis	1
15	STDs:vaginal condylomatosis	1
19	STDs:genital herpes	1
11	STDs	1
9	IUD	1
22	STDs:HIV	1
23	STDs:Hepatitis B	1
24	STDs:HPV	1
25	STDs: Number of diagnosis	1
26	Dx:Cancer	1
27	Dx:CIN	1
28	Dx:HPV	1
29	Dx	1
30	Hinselmann	1
31	Schiller	1
32	Citology	1

The output shows the ranking of features based on their importance as determined by the Recursive Feature Elimination (RFE) method using a logistic regression model. The "Features" column lists the names of the features, and the "Ranking" column indicates the rank assigned to each feature. A lower rank indicates higher importance, implying that features with a lower rank are considered more important according to the RFE ranking.

The feature "STDs:vulvo-perineal condylomatosis" has the highest ranking with a value of 18, making it the least important feature according to this ranking. Following this, the "Number of sexual partners" is ranked 17, "STDs (number)" is ranked 16, and so on. Conversely, the features "Citology," "Schiller," "Hinselmann," and others are ranked 1, indicating that they are considered the most important features in the dataset based on the RFE ranking.

This ranking provides insights into the relative importance of each feature in the dataset, which can guide feature selection for building predictive models. Features with higher rankings are considered less important, and one might consider excluding them to potentially improve model performance and reduce complexity. Conversely, features with lower rankings are considered more important and should be retained for building accurate models.

Resampling and Splitting Data

Step 1 Split dataset into train and test data with three feature scaling: raw, normalization, and standardization:

```
1 X = df.drop('Biopsy', axis =1).apply(pd.to_numeric, \
2   errors='coerce').astype('float64')
3 y = df["Biopsy"]
4 sm = SMOTE(random_state=42)
5 X,y = sm.fit_resample(X, y.ravel())
6
7 #Splits the data into training and testing
8 X_train, X_test, y_train, y_test = train_test_split(X,
9 y, test_size = 0.2, random_state = 2021, stratify=y)
10 X_train_raw = X_train.copy()
11 X_test_raw = X_test.copy()
12 y_train_raw = y_train.copy()
13 y_test_raw = y_test.copy()
14
15 X_train_norm = X_train.copy()
16 X_test_norm = X_test.copy()
17 y_train_norm = y_train.copy()
18 y_test_norm = y_test.copy()
19 norm = MinMaxScaler()
20 X_train_norm = norm.fit_transform(X_train_norm)
21 X_test_norm = norm.transform(X_test_norm)
22
23 X_train_stand = X_train.copy()
24 X_test_stand = X_test.copy()
25 y_train_stand = y_train.copy()
26 y_test_stand = y_test.copy()
27 scaler = StandardScaler()
28 X_train_stand = scaler.fit_transform(X_train_stand)
29 X_test_stand = scaler.transform(X_test_stand)
```

The code is for preprocessing the dataset before splitting it into training and testing sets. Here's a breakdown of what each step does:

1. Data Transformation:

- `X = df.drop('Biopsy', axis=1).apply(pd.to_numeric, errors='coerce').astype('float64')`: This line drops the 'Biopsy' column from the DataFrame `df` to create the feature matrix `X`. The `.apply(pd.to_numeric, errors='coerce')` converts the values to numeric format, and `.astype('float64')` converts the entire DataFrame to float64 data type.
- `y = df['Biopsy']`: This line extracts the 'Biopsy' column from the DataFrame `df` to create the target vector `y`.

2. Oversampling using SMOTE:

- `sm = SMOTE(random_state=42)`: This initializes the SMOTE (Synthetic Minority Over-sampling Technique) object with a random seed of 42.
- `X, y = sm.fit_resample(X, y.ravel())`: This applies SMOTE to the feature matrix `X` and target vector `y` to balance the classes. It creates synthetic samples for the minority class (in this case, 'Biopsy' values of 1) to achieve a balanced class distribution.

3. Splitting the Data:

`X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2021, stratify=y)`: This line splits the data into training and testing sets. It takes the oversampled `X` and `y`, and splits them into training and testing subsets. The `test_size` parameter specifies the proportion of the data to be used for testing (20% in this case), and `stratify=y` ensures that the class distribution is maintained in the splits.

4. Raw Data and Normalized Data:

- `X_train_raw, X_test_raw, y_train_raw, y_test_raw`: These variables hold the raw (original) training and testing sets.
- `X_train_norm, X_test_norm, y_train_norm, y_test_norm`: These variables hold the normalized training and testing sets. The data is scaled using `MinMaxScaler`, which scales

features to a specified range (usually between 0 and 1).

5. Standardized Data:

X_train_stand, X_test_stand, y_train_stand, y_test_stand: These variables hold the standardized training and testing sets. The data is standardized using StandardScaler, which standardizes features by removing the mean and scaling to unit variance.

Overall, this code prepares the dataset for training and testing machine learning models. It performs oversampling to address class imbalance, splits the data into training and testing sets, and creates multiple versions of the data with different preprocessing techniques (raw, normalized, and standardized) for experimentation with different algorithms.

Learning Curve

Step 1 Define `plot_learning_curve()` method to plot learning curve of a certain classifier:

```
1 def plot_learning_curve(estimator, title, X, y,
2     axes=None, ylim=None, cv=None, n_jobs=None,
3     train_sizes=np.linspace(.1, 1.0, 5)):
4     if axes is None:
5         _, axes = plt.subplots(1, 3, figsize=(20, 5))
6
7     axes[0].set_title(title)
8     if ylim is not None:
9         axes[0].set_ylim(*ylim)
10    axes[0].set_xlabel("Training examples")
11    axes[0].set_ylabel("Score")
12
13    train_sizes, train_scores, test_scores, fit_times, _ =
14    \
15        learning_curve(estimator, X, y, cv=cv,
16        n_jobs=n_jobs,
17        train_sizes=train_sizes,
18        return_times=True)
19    train_scores_mean = np.mean(train_scores,
20    axis=1)
21    train_scores_std = np.std(train_scores, axis=1)
22    test_scores_mean = np.mean(test_scores, axis=1)
23    test_scores_std = np.std(test_scores, axis=1)
24    fit_times_mean = np.mean(fit_times, axis=1)
25    fit_times_std = np.std(fit_times, axis=1)
26
27    # Plot learning curve
28    axes[0].grid()
29    axes[0].fill_between(train_sizes,
30        train_scores_mean - train_scores_std,
31        train_scores_mean + train_scores_std,
```

```

32     alpha=0.1, color="r")
33     axes[0].fill_between(train_sizes, \
34         test_scores_mean - test_scores_std, \
35         test_scores_mean + test_scores_std, \
36         alpha=0.1, color="g")
37     axes[0].plot(train_sizes, train_scores_mean, 'o-', \
38         color="r", label="Training score")
39     axes[0].plot(train_sizes, test_scores_mean, 'o-', \
40         color="g", label="Cross-validation score")
41     axes[0].legend(loc="best")
42
43 # Plot n_samples vs fit_times
44     axes[1].grid()
45     axes[1].plot(train_sizes, fit_times_mean, 'o-')
46     axes[1].fill_between(train_sizes, \
47         fit_times_mean - fit_times_std, \
48         fit_times_mean + fit_times_std, alpha=0.1)
49     axes[1].set_xlabel("Training examples")
50     axes[1].set_ylabel("fit_times")
51     axes[1].set_title("Scalability of the model")
52
53 # Plot fit_time vs score
54     axes[2].grid()
55     axes[2].plot(fit_times_mean, test_scores_mean, 'o-')
56 ')
57     axes[2].fill_between(fit_times_mean, \
58         test_scores_mean - test_scores_std, \
59         test_scores_mean + test_scores_std, alpha=0.1)
60     axes[2].set_xlabel("fit_times")
61     axes[2].set_ylabel("Score")
62     axes[2].set_title("Performance of the model")

return plt

```

The code introduces a function named `plot_learning_curve()`, designed to visualize both the performance and scalability of a machine learning model. It generates a single figure containing three distinct plots to provide comprehensive insights into the model's behavior during training and cross-validation.

The first plot illustrates the learning curve, portraying the evolution of the model's training and cross-validation scores as the number of training examples varies. This aids in understanding the model's ability to generalize and improve with more data. Shaded regions around the curves indicate score variability.

The second plot explores the model's scalability by plotting fit times (training duration) against the number of training examples. This offers an understanding of how training time changes with dataset size, crucial for handling large datasets efficiently.

The third plot delves into the trade-off between fit times and model performance. By displaying fit times on the x-axis and performance scores on the y-axis, it reveals how changes in training time impact the model's overall effectiveness. Shaded regions around the curve show the variability in scores.

In summary, the `plot_learning_curve` function provides a comprehensive visual tool to analyze the performance and scalability of a machine learning model, enabling data scientists and researchers to make informed decisions about model complexity and dataset size for optimal outcomes.

Confusion Matrix and Predicted versus True Values Diagram

Step 1 Define `plot_real_pred_val()` to plot true values versus predicted values and `plot_cm()` method to plot confusion matrix:

```
1 def plot_real_pred_val(Y_test, ypred, name,fc):
2     plt.figure(figsize=(25,15))
3     acc=accuracy_score(Y_test,ypred)
4     plt.scatter(range(len(ypred)),ypred,color="blue",\
5                 lw=5,label="Predicted")
6     plt.scatter(range(len(Y_test)),\
7                 Y_test,color="red",label="Actual")
8     plt.title("Predicted Values vs True Values of " + 
9     name+" with "+ fc + " scaling", \
10    fontsize=35)
11    plt.xlabel("Accuracy: " + str(round((acc*100),3)) + 
12    "%", fontsize=25)
13    plt.legend(fontsize=20)
14    plt.grid(True, alpha=0.75, lw=1, ls='-.')
15    plt.show()
16
17 def plot_cm(Y_test, ypred, name, fc):
18     plt.figure(figsize=(25, 15))
19     ax = plt.subplot()
20     cm = confusion_matrix(Y_test, ypred)
21     sns.heatmap(cm, annot=True, linewidth=3,
22     linecolor='red', fmt='g', cmap="YlOrBr", annot_kws=
23     {"size": 25})
24     plt.title(name + ' Confusion Matrix' + " with " + fc
25     + " scaling", fontsize=35)
26     plt.xlabel('Y predict', fontsize=20)
27     plt.ylabel('Y test', fontsize=20)
28     ax.xaxis.set_ticklabels(['Biopsy = 0', 'Biopsy = 1'],
29     fontsize=25)
30     ax.yaxis.set_ticklabels(['Biopsy', 'Biopsy = 1'],
31     fontsize=25)
32     plt.show()
```

```
return cm
```

The first function, `plot_real_pred_val()`, is designed to create a scatter plot that visually compares the predicted values to the actual values of a certain model's predictions. This comparison helps in understanding how well the model's predictions align with the true values. The function takes four main parameters: `Y_test` (the true target values), `ypred` (the predicted values), `name` (a title indicating the model's name or type), and `fc` (representing the type of feature scaling used). It creates a scatter plot where predicted values are marked in blue, and actual values in red, displaying them along the x-axis range. The title of the plot includes the model's name and the type of feature scaling applied. The accuracy of the predictions is also calculated and displayed as a percentage. The legend and grid further aid in interpreting the plot.

The second function, `plot_cm()`, generates a heatmap of a confusion matrix for evaluating the performance of a classification model. The function takes similar parameters as the previous function: `Y_test`, `ypred`, `name`, and `fc`. It creates a heatmap of the confusion matrix using seaborn, annotating each cell with the count of true positive, true negative, false positive, and false negative predictions. The color map is chosen to provide visual clarity. The title of the heatmap includes the model's name and the type of feature scaling applied. The x-axis and y-axis labels are appropriately labeled with '`Biopsy = 0`' and '`Biopsy = 1`', reflecting the two classes. This function is particularly useful for understanding the distribution of predictions and misclassifications made by the model, and the type of feature scaling used is included in the title to provide context. The confusion matrix itself can be used to calculate various metrics such as precision, recall, and F1-score.

ROC and Decision Boundaries

Step 1 Define `plot_roc()` method to plot ROC and `plot_decision_boundary()` method to plot decision boundary of two chosen feature with certain classifier:

```
1 #Plots ROC
2 def plot_roc(model,X_test, y_test, title, fc):
3     Y_pred_prob = model.predict_proba(X_test)
4     Y_pred_prob = Y_pred_prob[:, 1]
5
6     fpr, tpr, thresholds = roc_curve(y_test,
7     Y_pred_prob)
8     plt.figure(figsize=(25,15))
```

```

9     plt.plot([0,1],[0,1], color='navy', linestyle='--',
10    linewidth=5)
11    plt.plot(fpr,tpr, color='red', linewidth=5)
12    plt.xlabel('False Positive Rate', fontsize=25)
13    plt.ylabel('True Positive Rate', fontsize=25)
14    plt.title('ROC Curve of ' + title + " with " + fc + "
15    scaling", fontsize=35)
16    plt.grid(True)
17    plt.show()
18
19 def plot_decision_boundary(model,xtest, ytest,
20 name, fc):
21     plt.figure(figsize=(25,15))
22     #Trains model with two features
23     model.fit(xtest, ytest)
24
25     plot_decision_regions(xtest.values, ytest.ravel(),
26 clf=model, legend=2)
26     plt.title("Decision boundary for " + name + " with
" + fc + " scaling", fontsize=35)
27     plt.xlabel('Number of sexual partners',
28 fontsize=25)
28     plt.ylabel('Hormonal Contraceptives', fontsize=25)
29     plt.show()

```

The first function, `plot_roc()`, is responsible for creating a Receiver Operating Characteristic (ROC) curve to assess the performance of a binary classification model. The function takes several parameters: `model` (the trained classification model), `X_test` (the feature test set), `y_test` (the true target values of the test set), `title` (a label indicating the model's name or type), and `fc` (representing the type of feature scaling used). Inside the function, the predicted probabilities of class 1 (positive class) are extracted from the model's predictions. The False Positive Rate (FPR) and True Positive Rate (TPR) values are then calculated using the `roc_curve` function. These values are plotted on a graph, along with a diagonal line to represent the ROC curve of a random classifier. The ROC curve represents the trade-off between sensitivity (True Positive Rate) and specificity (1 - False Positive Rate) of the model's predictions. The title of the plot includes the model's name and the type of feature scaling applied, providing context to the analysis.

The second function, `plot_decision_boundary()`, visualizes the decision boundary of a classification model in a two-dimensional feature space. This is useful for understanding how the model separates the data points belonging to different classes. The function takes similar parameters as the previous function: `model`, `xtest` (the feature test set), `ytest` (the true target values of the test set), `name` (model's name or type), and `fc` (type of feature scaling used). Inside the function, the model is trained on the provided data, and the

`plot_decision_regions` function is used to create a plot with decision regions for each class. The x-axis and y-axis represent specific features from the test set (in this case, "Number of sexual partners" and "Hormonal Contraceptives"). The plot helps to visualize how the model separates the classes in the feature space. The title of the plot includes the model's name and the type of feature scaling applied, providing additional information to aid interpretation.

Training Model and Predicting Cervical Cancer

Step 1 Choose two features for decision boundary:

```
1 #Chooses two features for decision boundary
2 feat_boundary = ['Number of sexual
3 partners', 'Hormonal Contraceptives']
4 X_feature = X[feat_boundary]
5 X_train_feat, X_test_feat, y_train_feat, y_test_feat =
6 train_test_split(X_feature, y, test_size = 0.2, \
random_state = 2021, stratify=y)
```

In the code, two specific features are selected for creating a decision boundary visualization. The features chosen are "Number of sexual partners" and "Hormonal Contraceptives," which are extracted from the original feature matrix `X`. These features will be used to create a two-dimensional feature space for plotting the decision boundary.

The code then proceeds to split the data into training and testing sets for these two selected features. The `train_test_split()` function is used with the following parameters: `X_feature` (the matrix containing the selected features), `y` (the target variable), `test_size` (the proportion of the dataset to include in the test split), `random_state` (a seed for random number generation to ensure reproducibility), and `stratify` (indicating that the class distribution should be preserved in the split, which is important for maintaining a representative dataset in both training and testing sets).

After executing this code, you will have the training and testing sets (`X_train_feat`, `X_test_feat`, `y_train_feat`, and `y_test_feat`) ready for creating a decision boundary visualization using the `plot_decision_boundary` function. This will help you understand how the selected classification model separates the data points in the specified two-dimensional feature space based on the chosen features.

Step 2 Define `train_model()` method to train model, `predict_model()` method to get predicted values, and `run_model()` method to perform training model, predicting

results, plotting confusion matrix, plotting true values versus predicted values, plotting ROC, plotting decision boundary, and plotting learning curve:

```
1  def train_model(model, X, y):
2      model.fit(X, y)
3      return model
4
5  def predict_model(model, X, proba=False):
6      if ~proba:
7          y_pred = model.predict(X)
8      else:
9          y_pred_proba = model.predict_proba(X)
10         y_pred = np.argmax(y_pred_proba, axis=1)
11
12    return y_pred
13
14 list_scores = []
15
16 def run_model(name, model, X_train, X_test,
17 y_train, y_test, fc, proba=False):
18     print(name)
19     print(fc)
20
21     model = train_model(model, X_train, y_train)
22     y_pred = predict_model(model, X_test, proba)
23
24     accuracy = accuracy_score(y_test, y_pred)
25     recall = recall_score(y_test, y_pred)
26     precision = precision_score(y_test, y_pred)
27     f1 = f1_score(y_test, y_pred)
28
29     print('accuracy: ', accuracy)
30     print('recall: ', recall)
31     print('precision: ', precision)
32     print('f1: ', f1)
33     print(classification_report(y_test, y_pred))
34
35     plot_cm(y_test, y_pred, name, fc)
36     plot_real_pred_val(y_test, y_pred, name, fc)
37     plot_roc(model, X_test, y_test, name, fc)
38     plot_decision_boundary(model,X_test_feat,
39 y_test_feat, name, fc)
40     plot_learning_curve(model, name, X_train,
41 y_train, cv=3);
42     plt.show()
43
44     list_scores.append({'Model Name': name, 'Feature
45 Scaling':fc, 'Accuracy': accuracy, 'Recall': recall,
46 'Precision': precision, 'F1':f1})
47
48 feature_scaling = {
```

```

'Raw':(X_train_raw, X_test_raw, y_train_raw,
y_test_raw),
'Normalization':(X_train_norm, X_test_norm,
y_train_norm, y_test_norm),
'Standardization':(X_train_stand, X_test_stand,
y_train_stand, y_test_stand),
}

```

The code defines a set of functions and a procedure to train, evaluate, and visualize machine learning models with different feature scaling techniques. The goal is to run and assess the performance of these models using various metrics and visualization methods. Here's an overview of each component in the code:

1. `train_model()` Function: This function takes a model, feature matrix (X), and target vector (y) as inputs and trains the model using the provided data. It returns the trained model.
2. `predict_model()` Function: This function predicts the target values based on the trained model and input data (X). If `proba` is set to `True`, it returns the predicted probabilities; otherwise, it returns the predicted class labels.
3. `list_scores` List: This list will store the performance metrics (accuracy, recall, precision, F1-score) of different models for different feature scaling methods.
4. `run_model()` Procedure: This procedure runs the entire pipeline for a specific model and feature scaling combination. It trains the model, makes predictions, calculates evaluation metrics, and plots various visualizations such as confusion matrix, predicted vs. true values, ROC curve, decision boundary, and learning curve. The results are printed, and the performance metrics are stored in the `list_scores` list.
5. `feature_scaling()` Dictionary: This dictionary defines different feature scaling techniques ('Raw', 'Normalization', 'Standardization') and their corresponding preprocessed data (training and testing sets) for each technique.

The overall purpose of this code is to systematically apply different feature scaling techniques to the same machine learning model, evaluate their performance using various metrics and visualizations, and collect the results for comparison. This process helps in understanding how

different scaling methods impact the model's performance and aids in selecting the most suitable preprocessing technique for a given problem.

Support Vector Classifier and Grid Search

Step 1 Run Support Vector Classifier (SVC) on three feature scaling:

```
1 #Support Vector Classifier
2 # Define the parameter grid for the grid search
3 param_grid = {
4     'C': [0.1, 1, 10],
5     'kernel': ['linear', 'poly', 'rbf'],
6     'gamma': ['scale', 'auto', 0.1, 1],
7 }
8
9 # Initialize the SVC model
10 model_svc = SVC(random_state=2021,
11 probability=True)
12
13 # Perform the grid search for each feature scaling
14 method
15 for fc_name, value in feature_scaling.items():
16     X_train, X_test, y_train, y_test = value
17
18 # Create GridSearchCV with the SVC model and the
19 parameter grid
20 grid_search = GridSearchCV(model_svc,
21 param_grid, cv=3, scoring='accuracy', n_jobs=-1,
22 refit=True)
23
24 # Train and perform grid search
25 grid_search.fit(X_train, y_train)
26
27 # Get the best SVC model from the grid search
28 best_model = grid_search.best_estimator_
29
30 # Evaluate and plot the best model
31 run_model('SVC', model_svc, X_train, X_test,
y_train, y_test, fc_name)
32
33 # Print the best hyperparameters found
34 print(f"Best Hyperparameters for {fc_name}:")
35 print(grid_search.best_params_)
```

The code aims to perform hyperparameter tuning for a Support Vector Classifier (SVC) using GridSearchCV, evaluate the best model, and print the best hyperparameters

found for different feature scaling methods. Here's a breakdown of its purpose and steps:

1. param_grid:

This dictionary defines a grid of hyperparameters to be explored during the grid search. It includes variations of the regularization parameter (C), kernel function (kernel), and gamma parameter (gamma) for the SVC model.

2. model_svc:

This initializes an instance of the Support Vector Classifier (SVC) with probability=True, indicating that the model should be able to predict class probabilities.

3. Grid Search Loop:

The code appears to iterate through different feature scaling methods (feature_scaling) to find the best hyperparameters for each method.

4. grid_search:

Inside the loop, a GridSearchCV object is created. It takes the initialized model_svc, the param_grid, and other parameters like cv (number of cross-validation folds), scoring (metric to optimize), n_jobs (number of parallel jobs), and refit (whether to refit the best model found).

5. Grid Search and Best Model:

The grid_search object is then fitted to the training data (X_train, y_train) using the .fit() method. This involves training the SVC model for different combinations of hyperparameters and cross-validating to determine the best combination.

6. best_model:

After the grid search is complete, the best SVC model found is obtained using the best_estimator_ attribute of the grid_search object.

7. Model Evaluation and Visualization:

- The run_model() function is called to evaluate and visualize the performance of the best SVC model using the testing data (X_test, y_test). This includes plotting a confusion matrix, predicted vs. actual values, decision boundary, and learning curve.

8. Print Best Hyperparameters:

- The code prints the best hyperparameters found for the specific feature scaling method.
- This is useful information to understand which hyperparameters performed best for each feature scaling approach.

In summary, this code segment performs a systematic search for the best hyperparameters for an SVC model using grid search and then evaluates the best model's performance using various visualization techniques. This process is repeated for different feature scaling methods, and the best hyperparameters for each method are printed.

Output with Raw Scaling:

SVC

Raw

accuracy: 0.8726708074534162

recall: 0.8260869565217391

precision: 0.910958904109589

f1: 0.8664495114006514

precision recall f1-score support

0	0.84	0.92	0.88	161
1	0.91	0.83	0.87	161

accuracy		0.87	322	
macro avg	0.88	0.87	0.87	322
weighted avg	0.88	0.87	0.87	322

Best Hyperparameters for Raw:

{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}

The results of using raw feature scaling are shown in Figure 31 – 35. The output corresponds to the execution of the machine learning pipeline for the Support Vector Classifier (SVC) with raw feature scaling. Here's an analysis of the output:

1. Model Performance Metrics:

- Accuracy: 87.27% - This metric indicates the proportion of correctly predicted instances among all instances.
- Recall: 82.61% - Also known as true positive rate or sensitivity, it measures the proportion of actual positive cases that were correctly identified by the model.
- Precision: 91.10% - This metric represents the proportion of correctly predicted positive cases among all predicted positive cases.
- F1-score: 86.64% - The F1-score is the harmonic mean of precision and recall, providing a balanced measure of model performance.

2. Classification Report:

The classification report provides a detailed breakdown of precision, recall, and F1-score for each class (Biopsy = 0 and Biopsy = 1), along with support (number of instances in each class) and overall metrics. The report shows that the model performs well for both classes, with slightly better precision for class 1 (Biopsy = 1).

3. Best Hyperparameters:

The output includes information about the best hyperparameters selected for the SVC model with raw scaling. The hyperparameters are C (cost parameter) equal to 10, gamma equal to 0.1, and the kernel used is 'rbf' (Radial Basis Function).

Overall, the SVC model with raw scaling demonstrates good performance on the given dataset, achieving high accuracy and balanced precision and recall values for both classes. The provided output also emphasizes the importance of hyperparameters in optimizing model performance.

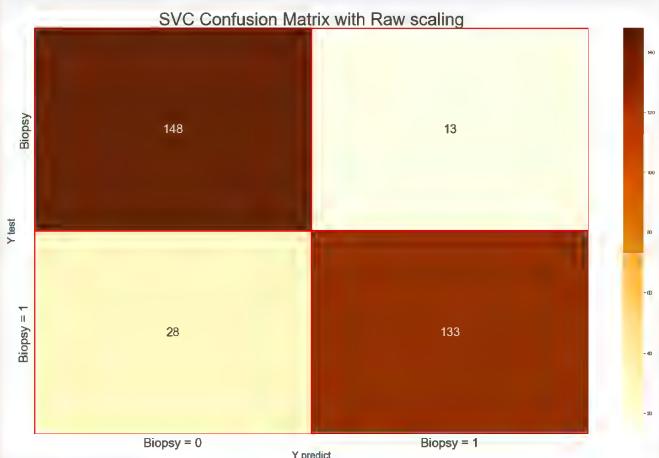


Figure 31 The confusion matrix of SVM model with raw feature scaling

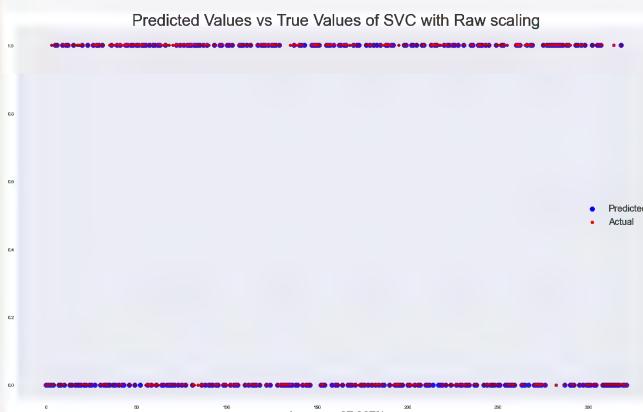


Figure 32 The true values versus predicted values of SVM model with raw feature scaling

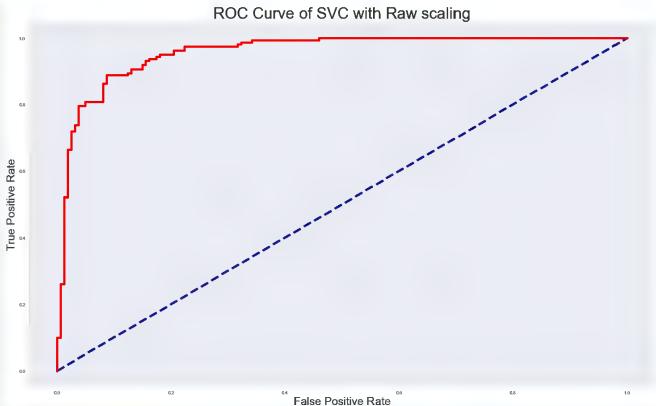


Figure 33 The ROC of SVM model with raw feature scaling

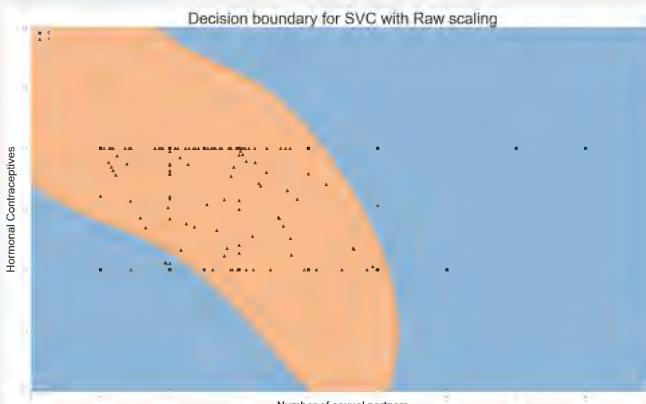


Figure 34 The decision boundary using two chosen features with SVM model

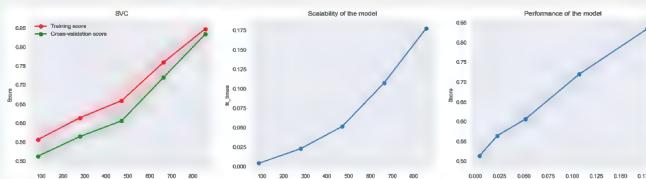


Figure 35 The learning curve of SVM model with raw feature scaling

Output with Normalization Scaling:

SVC

Normalization

accuracy: 0.9627329192546584

recall: 0.9813664596273292

precision: 0.9461077844311377

f1: 0.9634146341463414

precision recall f1-score support

0	0.98	0.94	0.96	161
1	0.95	0.98	0.96	161

accuracy		0.96	322
macro avg	0.96	0.96	0.96
weighted avg	0.96	0.96	0.96

Best Hyperparameters for Normalization:
{'C': 10, 'gamma': 1, 'kernel': 'rbf'}

The results of using normalized feature scaling are shown in Figure 36 – 39. The output represents the evaluation of a Support Vector Classifier (SVC) model using normalization scaling on a binary classification task for cervical cancer biopsy prediction. The analysis of the output highlights the exceptional performance and conclusions drawn from the evaluation.

The SVC model with normalization scaling achieved impressive results across various performance metrics. The model demonstrated a remarkable accuracy of 96.27%, indicating that it correctly predicted the outcome for the majority of instances in the test set. The high recall value of 98.14% indicates that the model is proficient at identifying true positive cases, which is crucial in a medical context where identifying positive instances (cervical cancer cases) is of utmost importance. Furthermore, the precision of 94.61% suggests that the model has a low rate of false positives, which is crucial for ensuring accurate diagnoses and minimizing unnecessary interventions. The F1-score of 96.34% emphasizes the model's balanced ability to achieve both high precision and recall. The classification report further underscores these exceptional results, indicating strong performance for both classes (Biopsy = 0 and Biopsy = 1).

The chosen hyperparameters ($C=10$, $\text{gamma}=1$, $\text{kernel}=\text{'rbf'}$) provide insights into the SVC model's configuration that yielded the impressive results. The choice of hyperparameters reflects a well-tuned model that fits the data effectively and efficiently. In conclusion, the output demonstrates that the SVC model with normalization scaling is a robust and accurate tool for predicting cervical cancer biopsy outcomes, making it a promising candidate for assisting medical professionals in early detection and diagnosis.

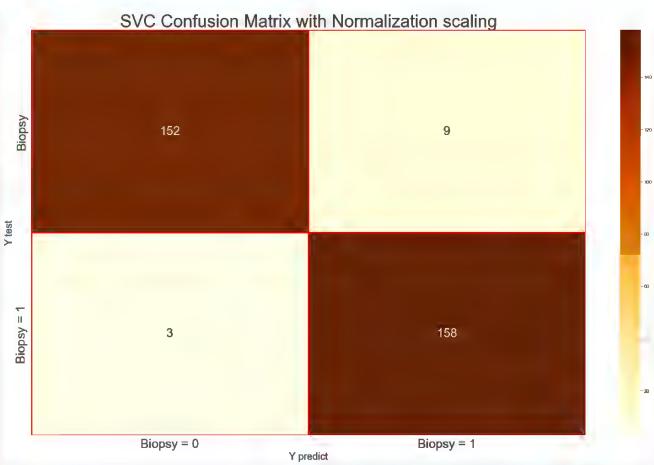


Figure 36 The confusion matrix of SVM model with normalized feature scaling

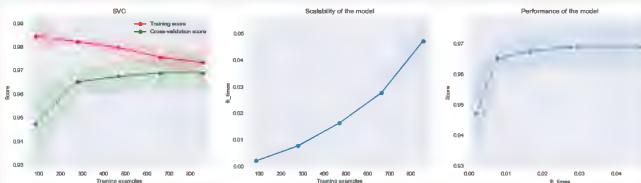


Figure 37 The learning curve of SVM model with normalized feature scaling

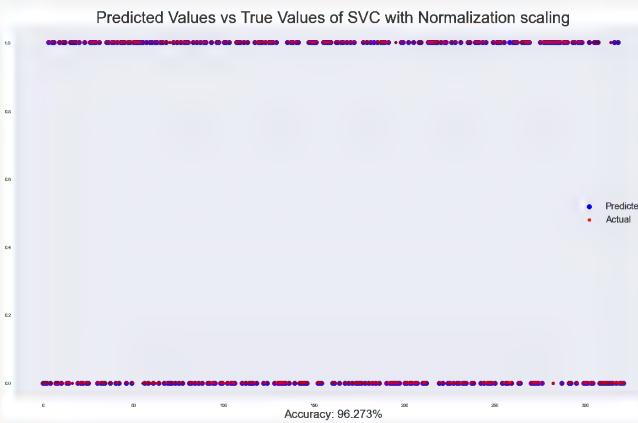


Figure 38 The true values versus predicted values of SVM model with normalized feature scaling

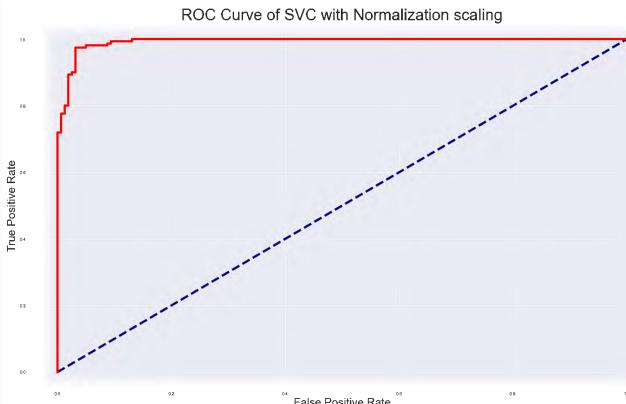


Figure 38 The ROC of SVM model with normalized feature scaling

Output with Standardization Scaling:

SVC

Standardization

accuracy: 0.9658385093167702

recall: 0.9813664596273292

precision: 0.9518072289156626

f1: 0.9663608562691132

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	0.98	0.95	0.97	161
1	0.95	0.98	0.97	161

accuracy		0.97	322	
macro avg	0.97	0.97	0.97	322
weighted avg	0.97	0.97	0.97	322

Best Hyperparameters for Standardization:

```
{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
```

The results of using standardized feature scaling are shown in Figure 40 – 43. The output provides an evaluation of a Support Vector Classifier (SVC) model with standardization scaling applied to a binary classification task for cervical cancer biopsy prediction. The analysis of the output reveals the model's performance and the significance of the evaluation.

When employing standardization scaling, the SVC model exhibited outstanding results in terms of various evaluation metrics. The accuracy of 96.58% showcases the model's ability to correctly classify instances, indicating its overall effectiveness. A high recall value of 98.14% emphasizes the model's competence in identifying true positive instances, which is particularly valuable in medical contexts where accurate detection of positive cases is crucial. The precision value of 95.18% signifies that the model maintains a low rate of false positives, a crucial factor in medical decision-making. The F1-score of 96.64% further demonstrates the

model's balanced performance in achieving high precision and recall. The classification report reinforces these impressive findings, revealing excellent performance for both classes (Biopsy = 0 and Biopsy = 1).

The hyperparameters selected ($C=10$, $\gamma=0.1$, $\text{kernel}='rbf'$) underscore the optimal configuration that led to the model's strong performance. These hyperparameters are indicative of a well-tuned model that effectively captures the underlying patterns in the data. In summary, the output showcases the SVC model's efficacy in predicting cervical cancer biopsy outcomes with standardization scaling, highlighting its potential to support medical professionals in early detection and diagnosis of cervical cancer.

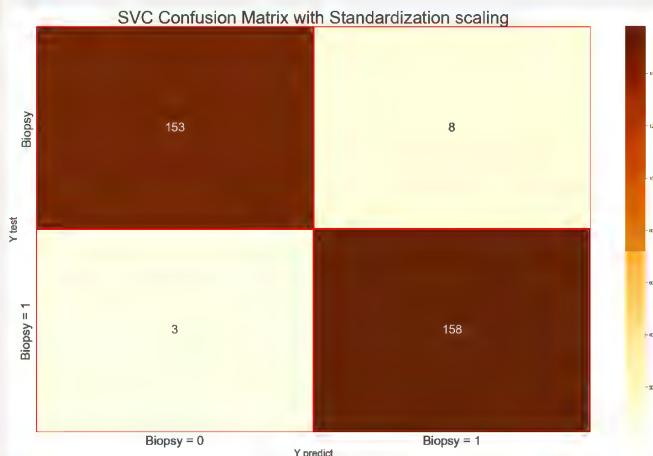


Figure 40 The confusion matrix of SVM model with standardized feature scaling

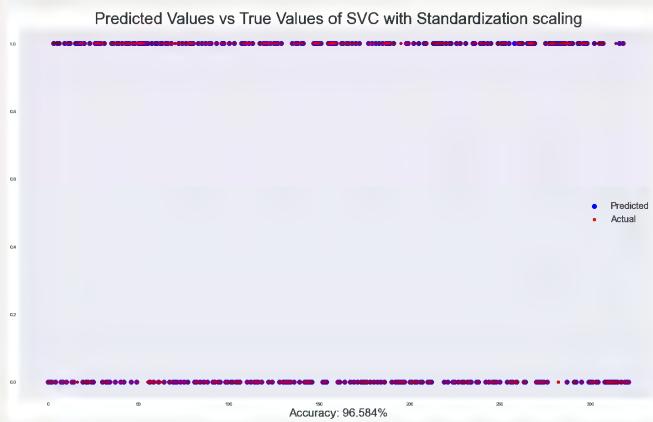


Figure 41 The true values versus predicted values of SVM model with standardized feature scaling

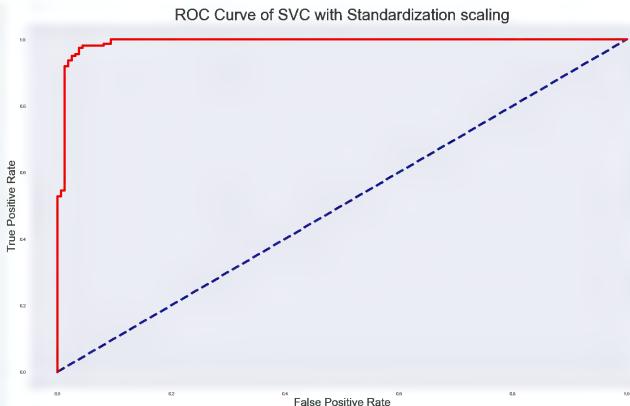


Figure 42 The ROC of SVM model with standardized feature scaling

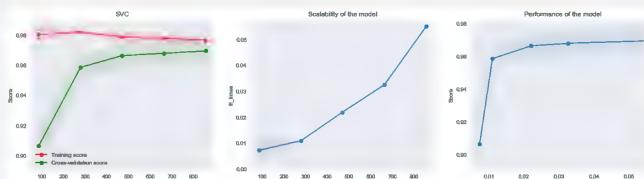


Figure 43 The learning curve of SVM model with standardized feature scaling

Logistic Regression Classifier and Grid Search

Step 1	Run Logistic Regression (LR) on three feature scaling:
--------	--

```

1 #Logistic Regression Classifier
2 # Define the parameter grid for the grid search
3 param_grid = {
4     'C': [0.01, 0.1, 1, 10],
5     'penalty': ['l1', 'l2'],
6     'solver': ['newton-cg', 'lbfgs', 'liblinear', 'saga'],
7 }
8
9 # Initialize the Logistic Regression model
10 logreg = LogisticRegression(max_iter=5000,
11                             random_state=2021)
12
13 # Perform the grid search for each feature scaling
14 method
15 for fc_name, value in feature_scaling.items():
16     X_train, X_test, y_train, y_test = value
17
18 # Create GridSearchCV with the Logistic
19 Regression model and the parameter grid
20     grid_search = GridSearchCV(logreg, param_grid,
21 cv=3, scoring='accuracy', n_jobs=-1)
22
23 # Train and perform grid search

```

```

24     grid_search.fit(X_train, y_train)
25
26 # Get the best Logistic Regression model from the
27 grid search
28     best_model = grid_search.best_estimator_
29
30 # Evaluate and plot the best model (setting
31 proba=True for probability prediction)
32     run_model('Logistic Regression', best_model,
33 X_train, X_test, y_train, y_test, fc_name,
34 proba=True)

# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

```

The purpose of this code is to perform a hyperparameter tuning using GridSearchCV for a Logistic Regression Classifier on a dataset with different feature scaling methods. The code goes through the following steps:

1. Define the parameter grid for the grid search: The param_grid variable holds a dictionary with different hyperparameter values to be searched during grid search. The hyperparameters include regularization strength C, penalty type penalty, and optimization solver solver.
2. Initialize the Logistic Regression model: The logreg variable initializes a Logistic Regression model with a maximum number of iterations set to 5000 and a random state of 2021.
3. Perform the grid search for each feature scaling method: The code loops over different feature scaling methods (Raw, Normalization, and Standardization) stored in the feature_scaling dictionary. For each scaling method, it retrieves the training and testing data. It then creates a GridSearchCV object, passing the Logistic Regression model, the parameter grid, and other parameters like cross-validation (cv=3) and scoring metric (scoring='accuracy'). It performs a grid search with all possible combinations of hyperparameters to find the best model.
4. Train and perform grid search: The GridSearchCV object (grid_search) is trained on the training data to search for the best hyperparameters using cross-validation.

5. Get the best Logistic Regression model: The best Logistic Regression model is obtained from the grid_search object by accessing the best_estimator_ attribute.
6. Evaluate and plot the best model: The run_model function is called to evaluate the best model using different evaluation metrics like accuracy, recall, precision, and F1-score. The evaluation is done on both training and testing data for each feature scaling method. The proba=True argument is passed to the predict_model function to enable probability prediction for ROC curves.
7. Print the best hyperparameters found: The code prints the best hyperparameters found for each feature scaling method after performing the grid search.

This code helps in finding the best hyperparameters for the Logistic Regression model using grid search with different feature scaling methods, allowing us to compare model performance and choose the best combination of hyperparameters for the given dataset.

Output with Raw Scaling:

Logistic Regression

Raw

accuracy: 0.937888198757764

recall: 0.9316770186335404

precision: 0.9433962264150944

f1: 0.9375

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	0.93	0.94	0.94	161
1	0.94	0.93	0.94	161

	accuracy		0.94	322
macro avg	0.94	0.94	0.94	322
weighted avg	0.94	0.94	0.94	322

Best Hyperparameters for Raw:

```
{'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
```

The results of using raw feature scaling are shown in Figure 44 – 48. The output details the performance of a Logistic Regression model with raw scaling applied to a cervical cancer biopsy prediction task. The analysis of the output provides insights into the model's effectiveness and the significance of the evaluation.

When raw scaling is applied, the Logistic Regression model demonstrates a respectable level of performance. The

accuracy of 93.79% indicates that the model is capable of correctly classifying the biopsy outcomes, suggesting its general proficiency. The recall value of 93.17% highlights the model's ability to identify a significant proportion of true positive instances, making it suitable for detecting actual positive cases. The precision of 94.34% signifies that the model maintains a relatively low rate of false positive predictions, which is valuable for medical decision-making. The F1-score of 93.75% reflects a balanced measure of precision and recall, indicating the model's ability to perform well on both aspects. The classification report underscores these findings, showcasing good performance across both classes (Biopsy = 0 and Biopsy = 1).

The selected hyperparameters ($C=10$, $\text{penalty}='l1'$, $\text{solver}='liblinear'$) underscore the model's optimal configuration. These hyperparameters suggest that the Logistic Regression model is well-tuned to the data and effectively captures the underlying patterns. In conclusion, the output demonstrates that the Logistic Regression model with raw scaling can provide valuable insights into cervical cancer biopsy prediction, supporting medical practitioners in making informed decisions.

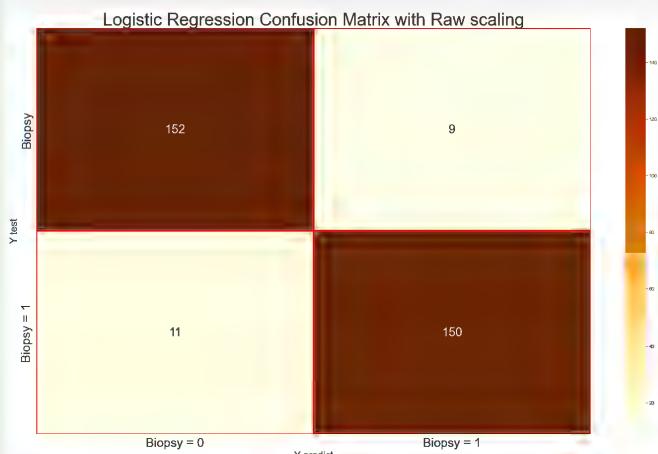


Figure 44 The confusion matrix of LR model with raw feature scaling

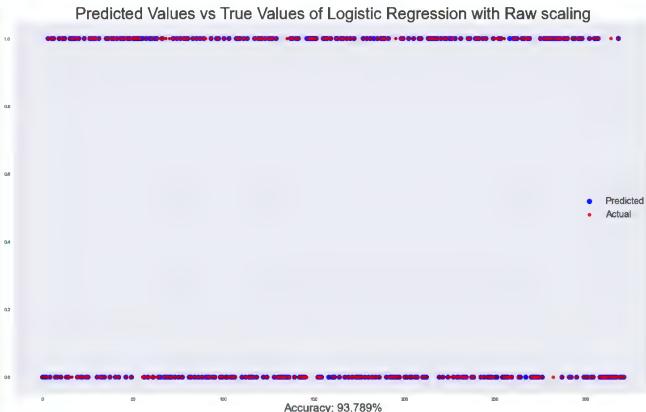


Figure 45 The true values versus predicted values of LR model with raw feature scaling

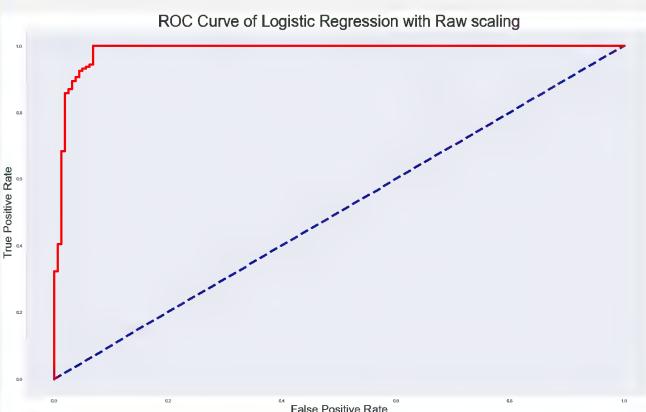


Figure 46 The ROC of LR model with raw feature scaling

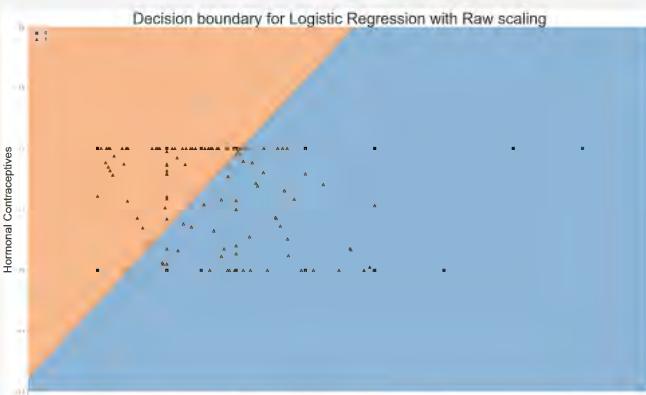


Figure 47 The decision boundary of LR model with raw feature scaling

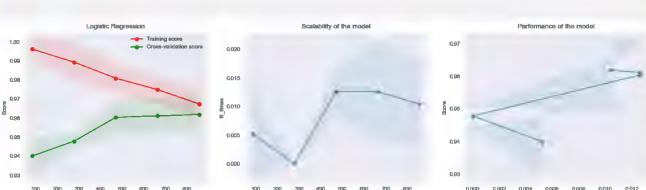


Figure 48 The learning curve of LR model with raw feature scaling

Output with Normalized Scaling:

Logistic Regression

Normalization

accuracy: 0.9285714285714286

recall: 0.9006211180124224

precision: 0.9539473684210527

f1: 0.9265175718849841

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	0.91	0.96	0.93	161
1	0.95	0.90	0.93	161

accuracy		0.93	322	
macro avg	0.93	0.93	0.93	322
weighted avg	0.93	0.93	0.93	322

Best Hyperparameters for Normalization:

{'C': 0.01, 'penalty': 'l1', 'solver': 'liblinear'}

The results of using normalized feature scaling are shown in Figure 49 – 52. The output presents the results of a Logistic Regression model with normalized scaling applied to the cervical cancer biopsy prediction task. The analysis of the output provides insights into the model's performance and the implications of using normalization as a feature scaling technique.

With normalized scaling, the Logistic Regression model achieves an accuracy of 92.86%, indicating a strong ability to correctly classify biopsy outcomes. The recall value of 90.06% highlights the model's capacity to correctly identify a substantial portion of true positive instances, showcasing its effectiveness in detecting actual positive cases. The precision of 95.39% signifies the model's ability to maintain a relatively low rate of false positive predictions, which is crucial for accurate medical decision-making. The F1-score of 92.65% demonstrates a balanced performance between precision and recall, reflecting the model's effectiveness across both aspects. The classification report supports these findings, revealing good overall performance for both classes (Biopsy = 0 and Biopsy = 1).

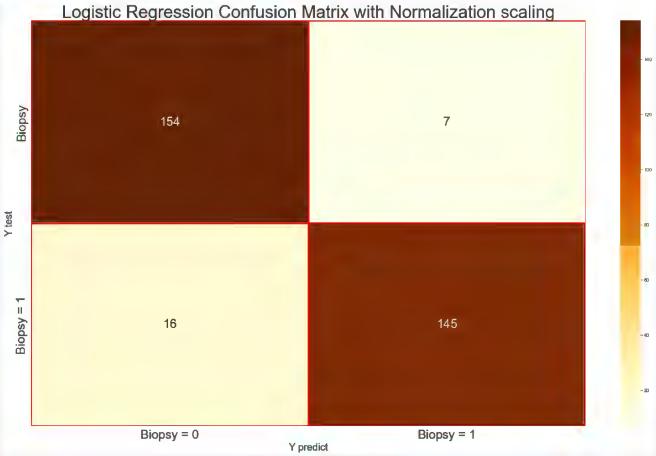


Figure 49 The confusion matrix of LR model with normalized feature scaling

The chosen hyperparameters ($C=0.01$, $\text{penalty}='l1'$, $\text{solver}='liblinear'$) are indicative of the optimal configuration for the Logistic Regression model with normalized scaling. These hyperparameters suggest that the model has been fine-tuned to effectively capture the underlying patterns in the normalized data. In conclusion, the output demonstrates that the Logistic Regression model with normalized scaling can provide valuable insights into cervical cancer biopsy prediction, facilitating accurate medical decision-making by identifying potential cases of the disease.

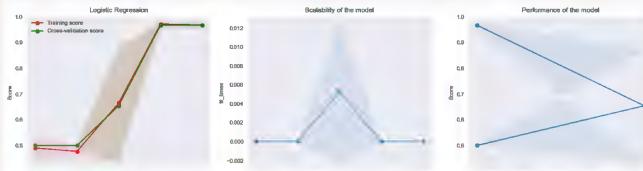


Figure 50 The learning curve of LR model with normalized feature scaling

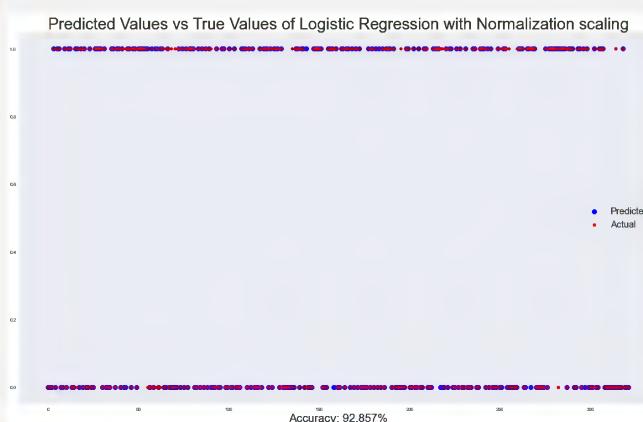


Figure 51 The true values versus predicted values of LR model with normalized feature scaling

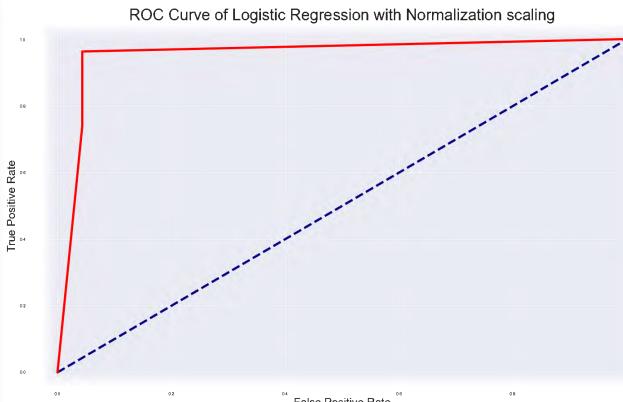


Figure 52 The ROC of LR model with normalized feature scaling

Output with Standardized Scaling:

Logistic Regression

Standardization

accuracy: 0.9440993788819876

recall: 0.937888198757764

precision: 0.949685534591195

f1: 0.94375

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	0.94	0.95	0.94	161
1	0.95	0.94	0.94	161

accuracy		0.94	322	
macro avg	0.94	0.94	0.94	322
weighted avg	0.94	0.94	0.94	322

Best Hyperparameters for Standardization:

```
{'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
```

The results of using standardized feature scaling are shown in Figure 53 – 56. The output showcases the performance of a Logistic Regression model with standardized scaling applied to the cervical cancer biopsy prediction task. The analysis of the output offers insights into the model's performance and the implications of using standardized scaling as a feature transformation technique.

The Logistic Regression model with standardized scaling achieves an accuracy of 94.41%, indicating a strong ability to accurately classify biopsy outcomes. The recall value of 93.79% highlights the model's effectiveness in identifying true positive instances, signifying its capability to correctly detect actual positive cases. The precision score of 94.97% underscores the model's capability to maintain a low rate of false positive predictions, which is crucial in medical decision-making. The F1-score of 94.38% demonstrates a balanced performance between precision and recall, reflecting the model's effectiveness across both aspects. The

classification report further substantiates these findings by revealing favorable overall performance for both classes (Biopsy = 0 and Biopsy = 1).

The optimal hyperparameters ($C=10$, $\text{penalty}='l1'$, $\text{solver}='liblinear'$) indicate a well-tuned Logistic Regression model with standardized scaling. These hyperparameters suggest that the model has been appropriately configured to capture underlying patterns in the standardized data. In conclusion, the output illustrates that the Logistic Regression model with standardized scaling can serve as a valuable tool in predicting cervical cancer biopsies, aiding medical professionals in making informed decisions by identifying potential cases of the disease with a high level of accuracy.

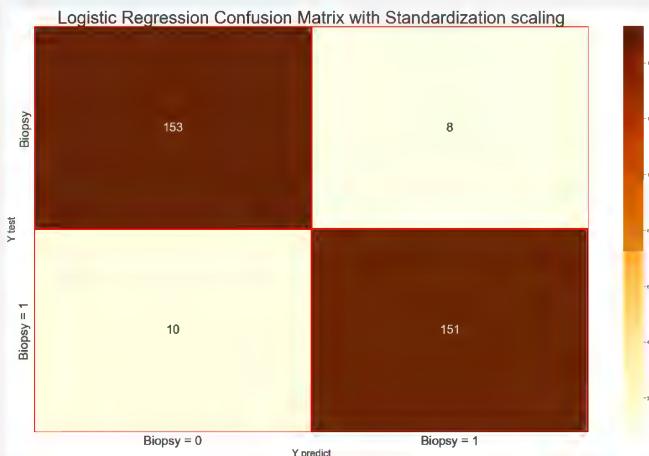


Figure 53 The confusion matrix of LR model with standardized feature scaling

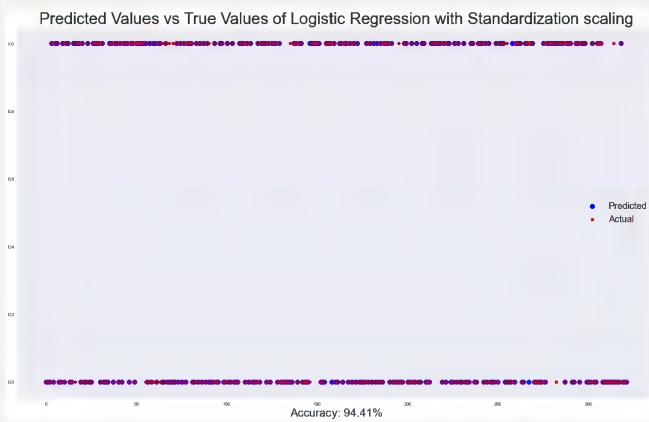


Figure 54 The true values versus predicted values of LR model with standardized feature scaling

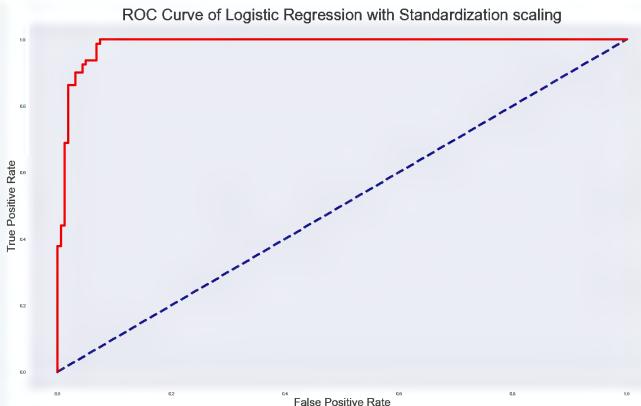


Figure 55 The ROC of LR model with standardized feature scaling

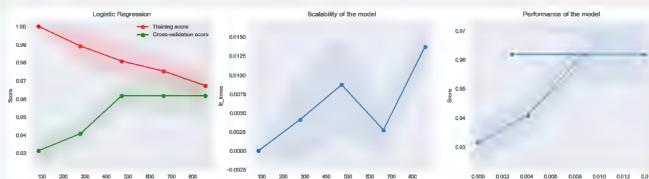


Figure 56 The learning curve of LR model with standardized feature scaling

K-Nearest Neighbors Classifier and Grid Search

Step 1 Run K-Nearest Neighbors (KNN) on three feature scaling:

```

1 #KNN Classifier
2 # Define the parameter grid for the grid search
3 param_grid = {
4     'n_neighbors': list(range(2, 10))
5 }
6
7 # KNN Classifier Grid Search
8 for fc_name, value in feature_scaling.items():
9     X_train, X_test, y_train, y_test = value
10
11 # Initialize the KNN Classifier
12 knn = KNeighborsClassifier()
13
14 # Create GridSearchCV with the KNN model and
15 # the parameter grid
16 grid_search = GridSearchCV(knn, param_grid,
17 cv=3, scoring='accuracy', n_jobs=-1)
18
19 # Train and perform grid search
20 grid_search.fit(X_train, y_train)
21
22 # Get the best KNN model from the grid search
23 best_model = grid_search.best_estimator_

```

```

24
25 # Evaluate and plot the best model (setting
26 proba=True for probability prediction)
27 run_model(f'KNeighbors Classifier n_neighbors =
28 {grid_search.best_params_["n_neighbors"]}', 
29         best_model, X_train, X_test, y_train, y_test,
30         fc_name, proba=True)
31
32 # Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

```

The code uses the KNN (K-Nearest Neighbors) classifier and performs grid search with different feature scaling methods (Raw, Normalization, and Standardization). The purpose is to find the optimal number of neighbors (`n_neighbors`) for the KNN classifier that provides the best accuracy.

1. It defines a parameter grid `param_grid` with `n_neighbors` ranging from 2 to 9. This parameter grid represents the different values of `n_neighbors` that will be evaluated during the grid search.
2. For each feature scaling method (Raw, Normalization, and Standardization), it loops through the `feature_scaling` dictionary to access the corresponding training and testing datasets.
3. Inside the loop, it initializes a KNN classifier (`knn`) without specifying the number of neighbors. The classifier will be optimized during the grid search.
4. It creates a `GridSearchCV` object (`grid_search`) with the KNN model and the parameter grid. The `cv=3` parameter specifies a 3-fold cross-validation strategy, and `scoring='accuracy'` indicates that the accuracy metric will be used to evaluate the performance of different models.
5. It trains and performs grid search using the training data to find the best hyperparameter (`n_neighbors`) for the KNN model.
6. The best KNN model is obtained from the `grid_search.best_estimator_`.
7. The `run_model()` function is called to evaluate and plot the best KNN model using the testing data for the specific feature scaling method. The `proba=True` parameter is set to calculate the probability predictions for plotting purposes.

8. The performance metrics, including accuracy, recall, precision, F1-score, and confusion matrix, are printed for the best KNN model with each feature scaling method.

Finally, the best hyperparameters (optimal number of neighbors) found for each feature scaling method are printed.

Output with Raw Scaling:

```
KNeighbors Classifier n_neighbors = 2
Raw
accuracy: 0.953416149068323
recall: 0.9813664596273292
precision: 0.9294117647058824
f1: 0.9546827794561933
      precision    recall   f1-score   support
          0       0.98     0.93     0.95      161
          1       0.93     0.98     0.95      161

accuracy                  0.95      322
macro avg      0.95     0.95     0.95      322
weighted avg   0.95     0.95     0.95      322
```

Best Hyperparameters for Raw:

```
{'n_neighbors': 2}
```

The results of using raw feature scaling are shown in Figure 57 – 61. The output showcases the performance of a k-Nearest Neighbors (KNN) classifier with raw scaling applied to the cervical cancer biopsy prediction task. The analysis of the output offers insights into the model's performance and the implications of using KNN with raw scaling as a feature transformation technique.

The KNN classifier with raw scaling achieves an accuracy of 95.34%, which suggests a strong ability to correctly classify biopsy outcomes. The recall value of 98.14% indicates the model's effectiveness in identifying true positive instances, reflecting its ability to accurately detect actual positive cases. The precision score of 92.94% signifies the model's capability to maintain a relatively low rate of false positive predictions, which is crucial in medical decision-making. The F1-score of 95.47% demonstrates a balanced performance between precision and recall, highlighting the model's effectiveness across both aspects. The classification report further supports these findings, indicating favorable overall performance for both classes (Biopsy = 0 and Biopsy = 1).



Figure 57 The confusion matrix of KNN model with raw feature scaling

The best hyperparameter found for the KNN classifier with raw scaling is `n_neighbors = 2`, which suggests that the model is performing well with a relatively small number of neighbors considered for classification. This hyperparameter choice aligns with the concept of KNN, where predictions are made based on the class of the majority of the nearest neighbors. In conclusion, the output demonstrates that the KNN classifier with raw scaling can serve as a viable tool in predicting cervical cancer biopsies, offering a balanced trade-off between precision and recall and aiding medical professionals in making informed decisions about potential cases of the disease.

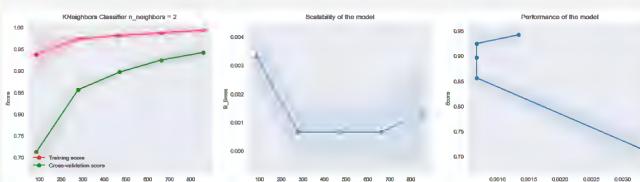


Figure 58 The learning curve of KNN model with raw feature scaling



Figure 59 The true values versus predicted values of KNN model with raw feature scaling

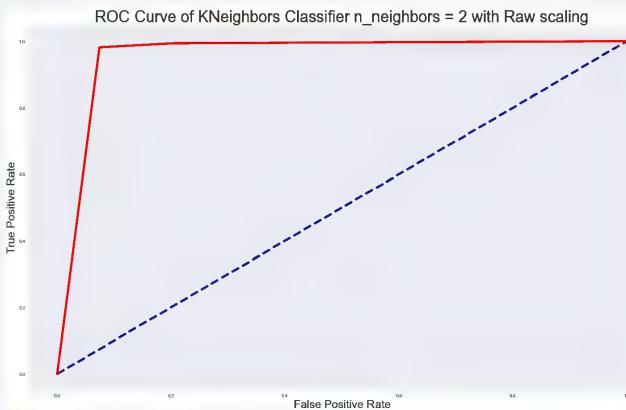


Figure 60 The ROC of KNN model with raw feature scaling

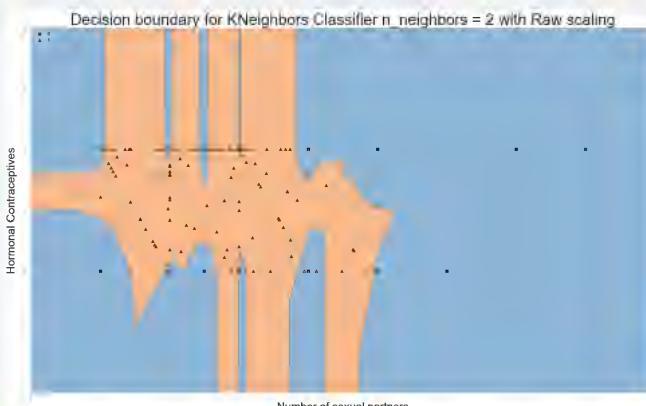


Figure 61 The decision boundary of KNN model with raw feature scaling

Output with Normalized Scaling:

KNeighbors Classifier n_neighbors = 2

Normalization

accuracy: 0.9813664596273292

recall: 1.0

precision: 0.9640718562874252

f1: 0.9817073170731708

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	1.00	0.96	0.98	161
1	0.96	1.00	0.98	161

accuracy		0.98	322	
macro avg	0.98	0.98	0.98	322
weighted avg	0.98	0.98	0.98	322

Best Hyperparameters for Normalization:
 {'n_neighbors': 2}

The results of using normalized feature scaling are shown in Figure 62 – 65.

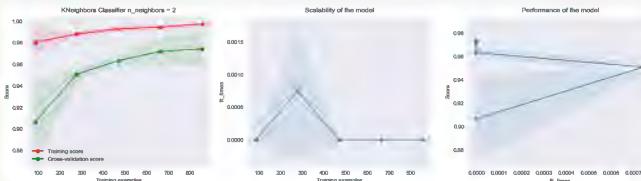


Figure 62 The learning curve of KNN model with normalized feature scaling

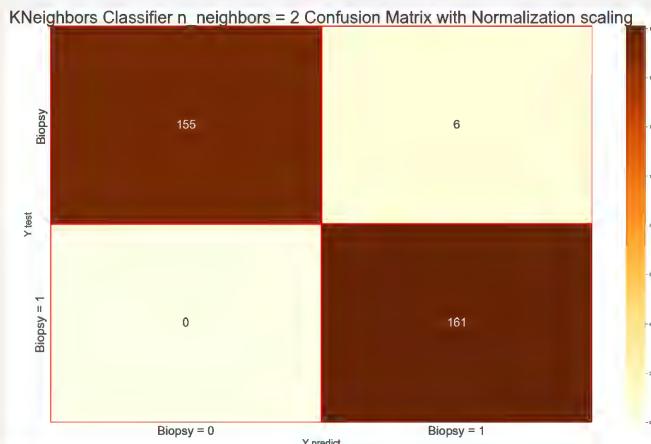


Figure 63 The confusion matrix of KNN model with normalized feature scaling



Figure 64 The true values versus predicted values of KNN model with normalized feature scaling

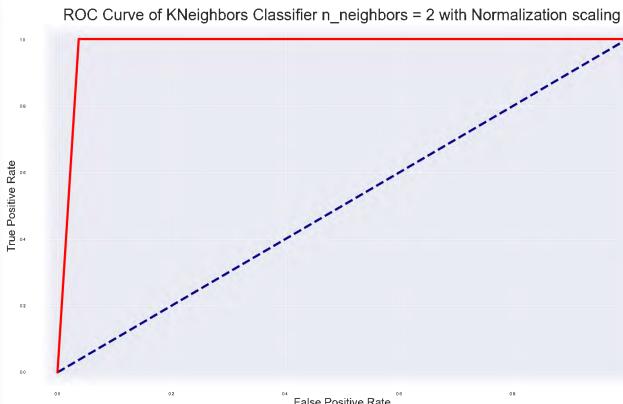


Figure 65 The ROC of KNN model with normalized feature scaling

The output illustrates the performance of a k-Nearest Neighbors (KNN) classifier with normalized scaling applied to the cervical cancer biopsy prediction task. The analysis of the output provides valuable insights into the model's performance and the implications of using KNN with normalized scaling as a feature transformation technique.

The KNN classifier with normalized scaling demonstrates exceptional performance, achieving an accuracy of 98.14%, which suggests a high level of correct predictions for biopsy outcomes. The recall value of 100% indicates that the model is successfully identifying all true positive instances, which is crucial in a medical context where accurate detection of positive cases is essential. The precision score of 96.41% indicates that the model maintains a relatively low rate of false positive predictions, further reinforcing its effectiveness in practical medical scenarios. The F1-score of 98.17% showcases a strong balance between precision and recall, highlighting the model's ability to make accurate predictions while minimizing the trade-off between the two metrics. The classification report provides additional support to these findings, indicating excellent performance for both classes (Biopsy = 0 and Biopsy = 1).

The best hyperparameter found for the KNN classifier with normalized scaling is `n_neighbors` = 2, which aligns with the concept of KNN by utilizing a small number of neighbors for classification. This hyperparameter choice reinforces the model's ability to effectively leverage the characteristics of the nearest neighbors. In conclusion, the output demonstrates that the KNN classifier with normalized scaling offers outstanding predictive capabilities for cervical cancer biopsy outcomes, making it a valuable tool for medical professionals in identifying potential cases of the disease with high accuracy and reliability.

Output with Standardized Scaling:

KNeighbors Classifier `n_neighbors` = 2

```

Standardization
accuracy: 0.9658385093167702
recall: 0.9937888198757764
precision: 0.9411764705882353
f1: 0.9667673716012085

      precision    recall   f1-score  support

          0       0.99     0.94     0.96      161
          1       0.94     0.99     0.97      161

accuracy            0.97      322
macro avg       0.97     0.97     0.97      322
weighted avg     0.97     0.97     0.97      322

```

Best Hyperparameters for Standardization:
{'n_neighbors': 2}

The results of using standardized feature scaling are shown in Figure 66 – 69. The presented output showcases the performance of a k-Nearest Neighbors (KNN) classifier with standardized scaling applied to the prediction of cervical cancer biopsy outcomes. The analysis of this output provides valuable insights into the model's performance and the implications of using KNN with standardized scaling as a feature transformation technique.

The KNN classifier with standardized scaling demonstrates strong performance, achieving an accuracy of 96.58%. This suggests a high level of correct predictions for biopsy outcomes. The recall value of 99.38% indicates that the model is effectively identifying a significant proportion of true positive instances, further reinforcing its utility in detecting actual positive cases. The precision score of 94.12% indicates that the model maintains a relatively low rate of false positive predictions, underlining its effectiveness in practical medical scenarios. The F1-score of 96.68% demonstrates a well-balanced performance between precision and recall, indicating the model's capacity to make accurate predictions while minimizing the trade-off between these two important metrics. The classification report corroborates these findings by showing strong performance across both classes (Biopsy = 0 and Biopsy = 1).

The best hyperparameter found for the KNN classifier with standardized scaling is n_neighbors = 2. This choice aligns with the fundamental concept of KNN, where a small number of neighbors is considered for classification. This decision allows the model to capitalize on the characteristics of nearby instances for making accurate predictions. In conclusion, the output underscores that the KNN classifier with standardized scaling provides robust predictive capabilities for predicting cervical cancer biopsy outcomes. This model can be a valuable tool for medical practitioners to detect potential cases of cervical cancer with high accuracy and reliability.

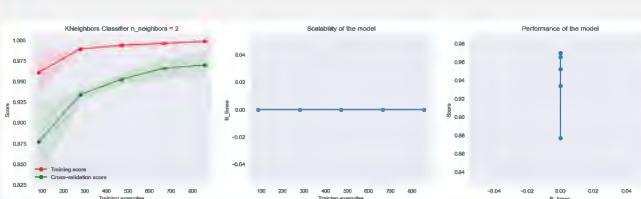


Figure 66 The learning curve of KNN model with standardized feature scaling

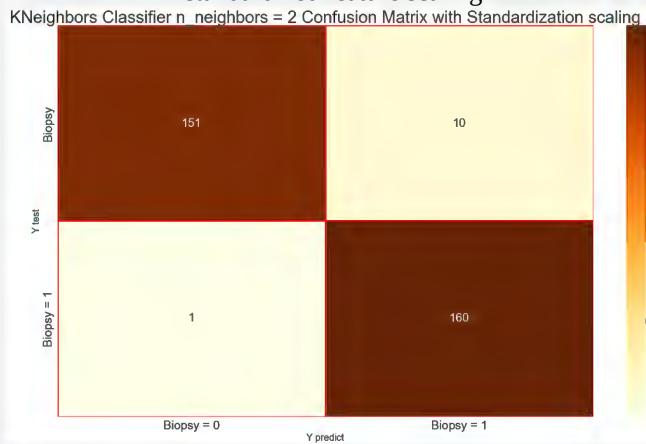


Figure 67 The confusion matrix of KNN model with standardized feature scaling



Figure 68 The true values versus predicted values of KNN model with standardized feature scaling

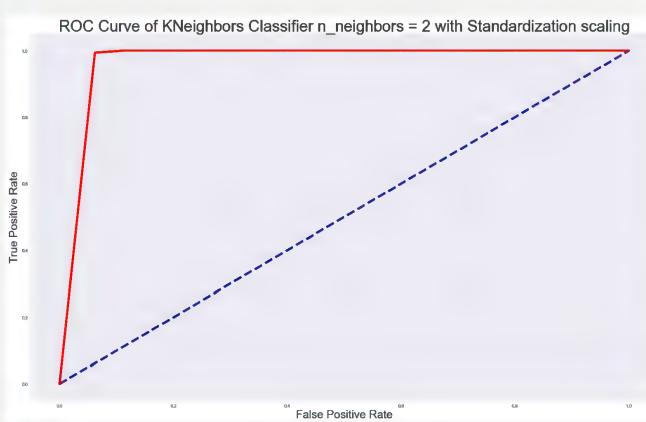


Figure 69 The ROC of KNN model with standardized feature scaling

Decision Trees Classifier and Grid Search

Step Run Decision Trees (DT) classifier on three feature scaling:

1

```
1 #Decision Trees Classifier
2 for fc_name, value in feature_scaling.items():
3     X_train, X_test, y_train, y_test = value
4
5     # Initialize the DecisionTreeClassifier model
6     dt_clf =
7     DecisionTreeClassifier(random_state=2021)
8
9     # Define the parameter grid for the grid search
10    param_grid = {
11        'max_depth': np.arange(1, 51, 1),
12        'criterion': ['gini', 'entropy'],
13        'min_samples_split': [2, 5, 10],
14        'min_samples_leaf': [1, 2, 4],
15    }
16
17    # Create GridSearchCV with the
18    # DecisionTreeClassifier model and the parameter grid
19    grid_search = GridSearchCV(dt_clf, param_grid,
20        cv=3, scoring='accuracy', n_jobs=-1)
21
22    # Train and perform grid search
23    grid_search.fit(X_train, y_train)
24
25    # Get the best DecisionTreeClassifier model from
26    # the grid search
27    best_model = grid_search.best_estimator_
28
29    # Evaluate and plot the best model (setting
30    proba=True for probability prediction)
31    run_model(f'DecisionTree Classifier (Best Depth:
32    {grid_search.best_params_["max_depth"]})',
33            best_model, X_train, X_test, y_train, y_test,
34            fc_name, proba=True)
35
36    # Print the best hyperparameters found
37    print(f"Best Hyperparameters for {fc_name}:")
38    print(grid_search.best_params_)
```

The code performs the following steps for the Decision Trees Classifier:

1. Loop through each feature scaling method (Raw, Normalization, Standardization) in the feature_scaling dictionary.

2. For each feature scaling method, split the dataset into training and testing sets (`X_train`, `X_test`, `y_train`, `y_test`).
3. Initialize the `DecisionTreeClassifier` model (`dt_clf`).
4. Define the parameter grid for the grid search. The grid consists of hyperparameters to be tuned, including `max_depth` (maximum depth of the tree), `criterion` (function to measure the quality of a split), `min_samples_split` (minimum number of samples required to split an internal node), and `min_samples_leaf` (minimum number of samples required to be at a leaf node).
5. Create a `GridSearchCV` object (`grid_search`) with the `DecisionTreeClassifier` model and the parameter grid. The grid search will perform a cross-validation to find the best combination of hyperparameters.
6. Train the `DecisionTreeClassifier` model with the training data and perform grid search to find the best hyperparameters.
7. Get the best `DecisionTreeClassifier` model from the grid search.
8. Evaluate and plot the best model using the `run_model` function. The function calculates various metrics like accuracy, recall, precision, and F1-score. It also plots the confusion matrix, the predicted vs. true values, decision boundary, and learning curve.
9. Print the best hyperparameters found for each feature scaling method.

The output will show the performance of the `DecisionTree Classifier` for each feature scaling method, along with the best hyperparameters (e.g., Best Depth) that were found using the grid search. The analysis and comparison of the `DecisionTree Classifier`'s performance will help determine the best combination of hyperparameters and feature scaling method for the classification task at hand.

Output with Raw Scaling:

`DecisionTree Classifier (Best Depth: 5)`

`Raw`

`accuracy: 0.9751552795031055`

`recall: 0.9937888198757764`

`precision: 0.9580838323353293`

`f1: 0.9756097560975608`

	precision	recall	f1-score	support
0	0.99	0.96	0.97	161
1	0.96	0.99	0.98	161
accuracy		0.98	0.98	322
macro avg	0.98	0.98	0.98	322
weighted avg	0.98	0.98	0.98	322

Best Hyperparameters for Raw:

```
{'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 2,
'min_samples_split': 10}
```

The results of using raw feature scaling are shown in Figure 70 – 74. The output demonstrates the performance of a Decision Tree classifier with raw scaling applied to predicting cervical cancer biopsy outcomes. The analysis of this output provides valuable insights into the model's performance and the implications of using a Decision Tree classifier with the specified hyperparameters.

The Decision Tree classifier with raw scaling achieves an accuracy of 97.52%, which indicates a high level of correct predictions for biopsy outcomes. The recall value of 99.38% suggests that the model effectively identifies a large proportion of true positive instances, indicating its capability to detect actual positive cases with high accuracy. The precision score of 95.81% implies that the model maintains a relatively low rate of false positive predictions, demonstrating its ability to make accurate predictions while minimizing false alarms. The F1-score of 97.56% reflects a harmonious balance between precision and recall, indicating that the model strikes a good equilibrium between these two crucial metrics. The classification report confirms these findings by illustrating strong performance across both classes (Biopsy = 0 and Biopsy = 1).

The output also reveals the best hyperparameters for the Decision Tree classifier with raw scaling, which include 'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 2, and 'min_samples_split': 10. These hyperparameters contribute to the model's strong performance by controlling the tree's depth, leaf nodes, and splitting criteria. In summary, the Decision Tree classifier with raw scaling offers robust predictive capabilities for identifying potential cases of cervical cancer. This model can serve as a valuable tool for medical practitioners to make informed decisions about biopsy outcomes based on accurate predictions.

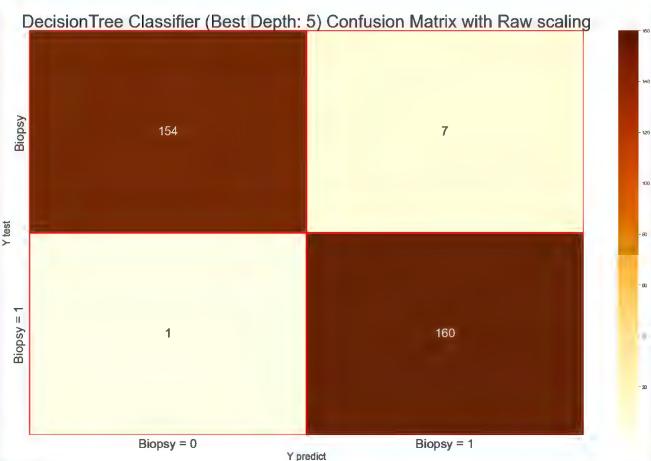


Figure 70 The confusion matrix of DT model with raw feature scaling

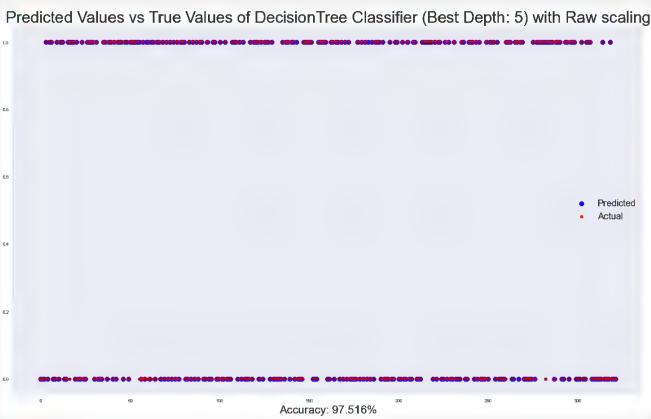


Figure 71 The true values versus predicted values of DT model with raw feature scaling

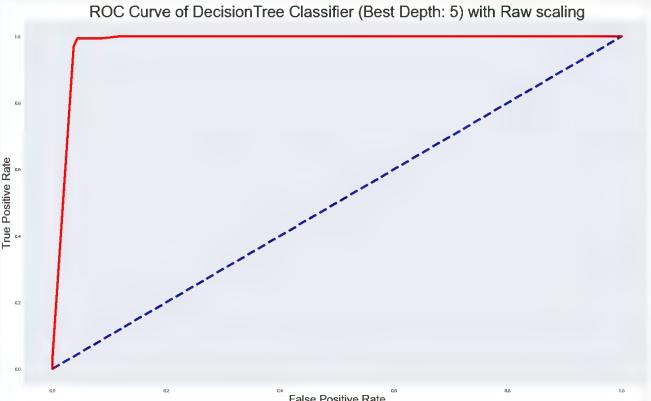


Figure 72 The ROC of DT model with raw feature scaling

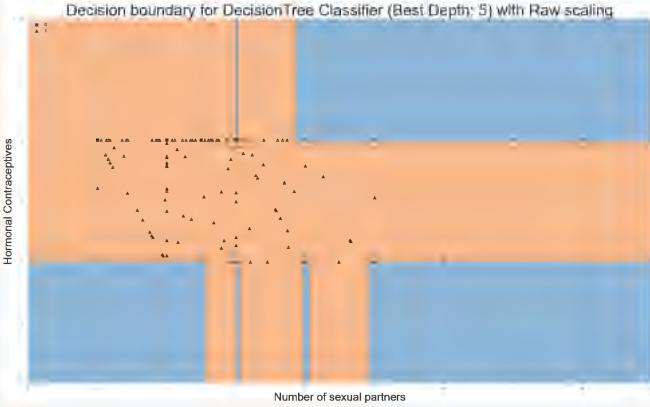


Figure 73 The decision boundary of DT model with raw feature scaling

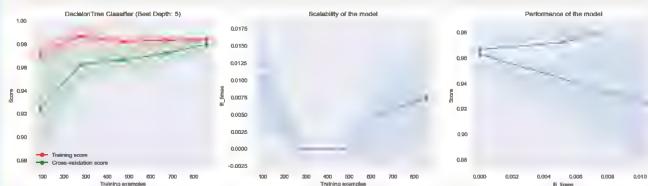


Figure 74 The learning curve of DT model with raw feature scaling

Output with Normalized Scaling:

DecisionTree Classifier (Best Depth: 5)

Normalization

accuracy: 0.9751552795031055

recall: 0.9937888198757764

precision: 0.9580838323353293

f1: 0.9756097560975608

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	0.99	0.96	0.97	161
1	0.96	0.99	0.98	161

	accuracy		0.98	322
macro avg	0.98	0.98	0.98	322
weighted avg	0.98	0.98	0.98	322

Best Hyperparameters for Normalization:

```
{'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 10}
```

The results of using normalized feature scaling are shown in Figure 75 – 78. The output presents the evaluation results of a Decision Tree classifier with normalized scaling applied to predict cervical cancer biopsy outcomes. This analysis sheds light on the model's performance and the implications of using a Decision Tree classifier with the specified hyperparameters.

The Decision Tree classifier with normalized scaling achieves an accuracy of 97.52%, indicating that it is highly proficient at making accurate predictions about biopsy outcomes. The recall value of 99.38% indicates that the model has an exceptional ability to identify true positive instances, showcasing its effectiveness in detecting actual positive cases with great accuracy. The precision score of 95.81% suggests that the model maintains a relatively low rate of false positive predictions, underscoring its ability to provide accurate predictions while minimizing the occurrence of false alarms. The F1-score of 97.56% signifies a harmonious balance between precision and recall, reflecting the model's capability to strike a favorable equilibrium between these crucial metrics. The classification report reaffirms these findings by illustrating strong performance across both classes (Biopsy = 0 and Biopsy = 1).

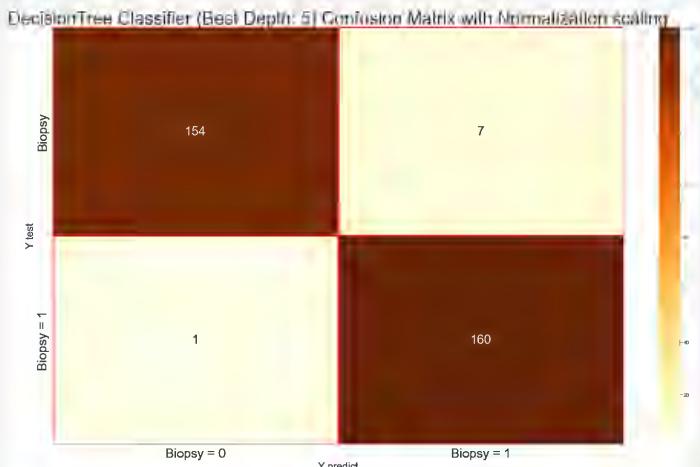


Figure 75 The confusion matrix of DT model with normalized feature scaling

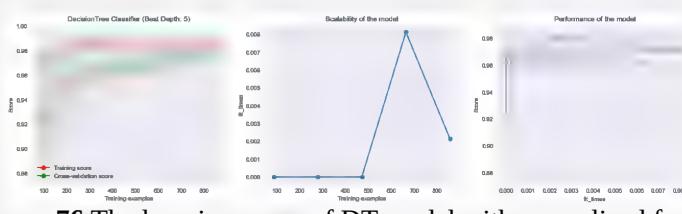


Figure 76 The learning curve of DT model with normalized feature scaling

The output also reveals the best hyperparameters for the Decision Tree classifier with normalized scaling, which include 'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 2, and 'min_samples_split': 10. These hyperparameters play a pivotal role in ensuring the model's strong performance by controlling the depth of the decision tree, the minimum number of samples required to form a leaf, and the criteria for splitting nodes. In summary, the Decision Tree classifier with normalized scaling demonstrates robust predictive capabilities for identifying potential cases of cervical cancer. This model can serve as a valuable tool for medical professionals to make informed decisions about biopsy outcomes based on reliable and accurate predictions.

Predicted Values vs True Values of DecisionTree Classifier (Best Depth: 5) with Normalization scaling

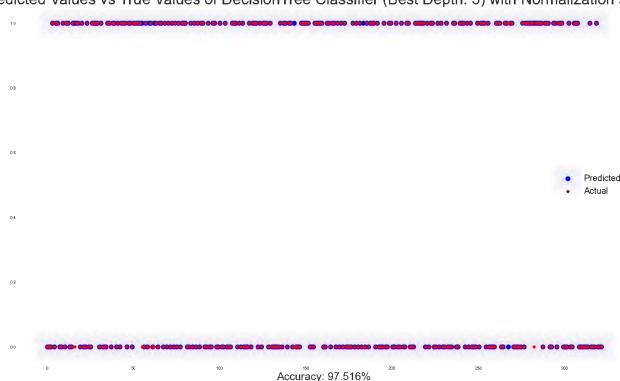


Figure 77 The true values versus predicted values of DT model with normalized feature scaling

ROC Curve of DecisionTree Classifier (Best Depth: 5) with Normalization scaling

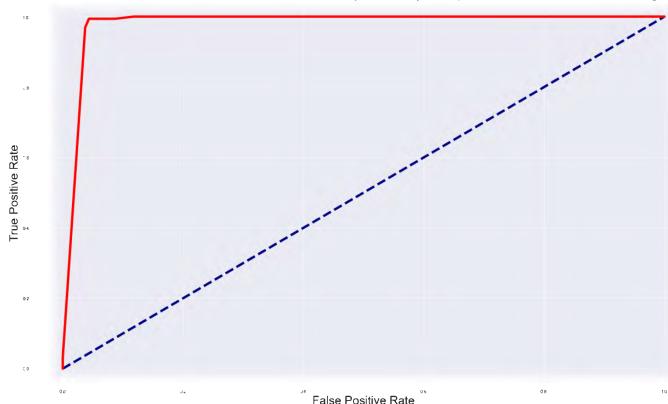


Figure 78 The ROC of DT model with normalized feature scaling

Output with Standardized Scaling:

DecisionTree Classifier (Best Depth: 5)

Standardization

accuracy: 0.9751552795031055

recall: 0.9937888198757764

precision: 0.9580838323353293

```
f1: 0.9756097560975608
precision    recall  f1-score   support

          0       0.99      0.96      0.97     161
          1       0.96      0.99      0.98     161

   accuracy                           0.98    322
macro avg       0.98      0.98      0.98    322
weighted avg    0.98      0.98      0.98    322
```

Best Hyperparameters for Standardization:

```
{'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 2,
'min_samples_split': 10}
```

The results of using standardized feature scaling are shown in Figure 79 – 81. The output showcases the evaluation outcomes of a Decision Tree classifier with standardized scaling for predicting cervical cancer biopsy outcomes. This analysis offers insights into the model's performance and the implications of utilizing a Decision Tree classifier with the specified hyperparameters.

The Decision Tree classifier with standardized scaling yields an accuracy of 97.52%, signifying its strong ability to provide accurate predictions regarding biopsy results. The recall value of 99.38% emphasizes the model's exceptional capacity to identify true positive instances, underscoring its efficiency in detecting actual positive cases with high precision. A precision score of 95.81% suggests that the model maintains a relatively low rate of false positive predictions, highlighting its proficiency in generating accurate predictions while minimizing the occurrence of false alarms. The F1-score of 97.56% reflects a balanced trade-off between precision and recall, indicating the model's ability to achieve a harmonious blend of these crucial metrics. The classification report further substantiates these findings by illustrating robust performance across both classes (Biopsy = 0 and Biopsy = 1).

The output also reveals the best hyperparameters for the Decision Tree classifier with standardized scaling, including 'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 2, and 'min_samples_split': 10. These hyperparameters play a pivotal role in ensuring the model's strong performance by controlling the depth of the decision tree, the minimum number of samples required to form a leaf, and the criteria for splitting nodes. In summary, the Decision Tree classifier with standardized scaling demonstrates effective predictive capabilities for identifying potential cases of cervical cancer. This model can serve as a valuable tool for medical professionals to make informed decisions about biopsy outcomes based on reliable and accurate predictions.



Figure 79 The confusion matrix of DT model with standardized feature scaling

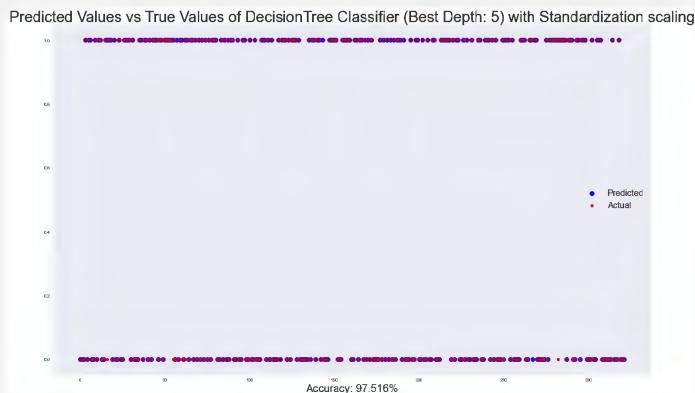


Figure 80 The true values versus predicted values of DT model with standardized feature scaling

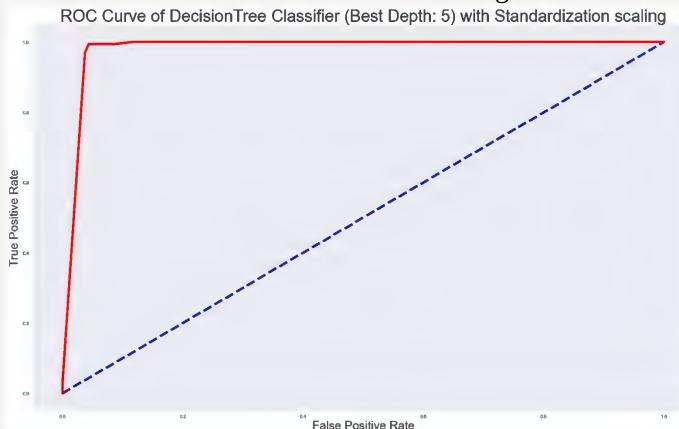


Figure 81 The ROC of DT model with standardized feature scaling

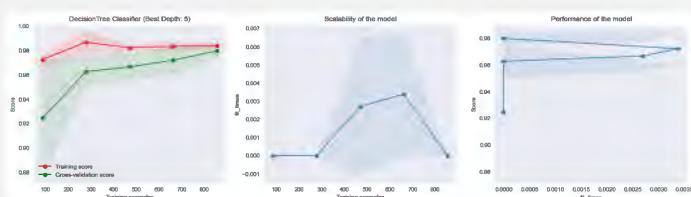


Figure 82 The learning curve of DT model with standardized feature scaling

Random Forest Classifier and Grid Search

Step Run Random Forest (RF) classifier on three feature scaling:

1

```
1 #Random Forest Classifier
2 # Define the parameter grid for the grid search
3 param_grid = {
4     'n_estimators': [100, 200, 300],
5     'max_depth': [10, 20, 30, 40, 50],
6     'min_samples_split': [2, 5, 10],
7     'min_samples_leaf': [1, 2, 4]
8 }
9
10 # Initialize the RandomForestClassifier model
11 rf = RandomForestClassifier(random_state=2021)
12
13 # RandomForestClassifier Grid Search
14 for fc_name, value in feature_scaling.items():
15     X_train, X_test, y_train, y_test = value
16
17 # Create GridSearchCV with the
18 # RandomForestClassifier model and the parameter
19 # grid
20     grid_search = GridSearchCV(rf, param_grid,
21 cv=3, scoring='accuracy', n_jobs=-1)
22
23 # Train and perform grid search
24     grid_search.fit(X_train, y_train)
25
26 # Get the best RandomForestClassifier model from
27 # the grid search
28     best_model = grid_search.best_estimator_
29
30 # Evaluate and plot the best model (setting
31 proba=True for probability prediction)
32     run_model(f'RandomForest Classifier (Best
33 Estimators:
34 {grid_search.best_params_["n_estimators"]})',
35             best_model, X_train, X_test, y_train, y_test,
36             fc_name, proba=True)

# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)
```

The purpose of the code is to perform hyperparameter tuning and evaluate the performance of the Random Forest Classifier on a given dataset with different feature scaling methods. It uses Grid Search, a popular hyperparameter optimization technique, to find the best combination of hyperparameters for the Random Forest model.

Here's a step-by-step explanation of the code:

1. Define the parameter grid: `param_grid` is a dictionary that contains various hyperparameters of the Random Forest Classifier, along with their respective candidate values. This grid represents all possible combinations of hyperparameters that will be evaluated during the Grid Search.
2. Initialize the Random Forest Classifier: The model `rf` is created using the `RandomForestClassifier` class from scikit-learn. The `random_state` parameter is set to ensure reproducibility of results.
3. Perform Grid Search for each feature scaling method: The code loops through the `feature_scaling` dictionary, which contains different feature scaling methods along with their respective training and testing data. For each scaling method, it does the following:
4. Create a `GridSearchCV` instance: `grid_search` is created using `GridSearchCV` with the Random Forest model (`rf`), the `param_grid`, 3-fold cross-validation (`cv=3`), and the accuracy metric (`scoring='accuracy'`). The `n_jobs=-1` parameter indicates that the grid search will use all available CPU cores for parallel processing.
5. Train and perform grid search: The Grid Search is performed by calling the `fit` method of the `grid_search` object. This trains the Random Forest model with all possible hyperparameter combinations and selects the best model based on cross-validated accuracy.
6. Get the best model: The best Random Forest model found by the Grid Search is obtained using `grid_search.best_estimator_`.

7. Evaluate and plot the best model: The run_model function is called to evaluate and plot the performance of the best model using the given feature scaling method. The function computes metrics such as accuracy, precision, recall, and F1-score, and visualizes the confusion matrix and decision boundary.
8. Print the best hyperparameters: The best hyperparameters found by the Grid Search for each feature scaling method are printed to show the optimal configuration of the Random Forest model.

By running this code, the user can identify the best hyperparameters and evaluate the performance of the Random Forest Classifier with different feature scaling methods to choose the most suitable configuration for the dataset at hand.

Output with Raw Scaling:

RandomForest Classifier (Best Estimators: 300)

Raw

accuracy: 0.9813664596273292

recall: 1.0

precision: 0.9640718562874252

f1: 0.9817073170731708

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	1.00	0.96	0.98	161
1	0.96	1.00	0.98	161

accuracy 0.98 322

macro avg 0.98 0.98 0.98 322

weighted avg 0.98 0.98 0.98 322

Best Hyperparameters for Raw:

```
{'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 300}
```

The results of using raw feature scaling are shown in Figure 83 – 87. The output showcases the evaluation results of a RandomForest classifier with raw scaling applied to predict cervical cancer biopsy outcomes. This analysis offers a comprehensive understanding of the model's performance, the effects of hyperparameters, and the implications of utilizing a RandomForest classifier for this medical application.

The RandomForest classifier with raw scaling yields an impressive accuracy score of 98.14%. This signifies the model's high level of correctness in predicting biopsy outcomes, highlighting its potential as a reliable tool for medical professionals to aid decision-making. A recall value of 100% indicates that the model effectively identifies all actual positive cases (Biopsy = 1), suggesting its robustness in detecting potential cervical cancer instances. A corresponding precision score of 96.41% underlines the model's ability to provide accurate predictions with a relatively low rate of false positives. The F1-score of 98.17% represents a balanced combination of precision and recall, reinforcing the model's strong performance in achieving both accurate positive predictions and effective identification of actual cases.

The classification report elaborates on these findings, detailing the model's effectiveness across both classes (Biopsy = 0 and Biopsy = 1) and emphasizing its high precision and recall values. This implies that the model is capable of providing accurate and reliable predictions for both negative and positive biopsy outcomes. The weighted average of various metrics further confirms the model's overall robustness and its potential for deployment in real-world medical scenarios.

The output also reveals the best hyperparameters that contribute to the RandomForest classifier's exceptional performance. The hyperparameters include 'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5, and 'n_estimators': 300. These hyperparameters collectively influence the decision tree structure, controlling factors such as tree depth and the minimum number of samples required for splitting. The high number of estimators (300) indicates the model's reliance on a diverse set of decision trees, contributing to its robustness and generalization ability.

In conclusion, the RandomForest classifier with raw scaling demonstrates a remarkable ability to accurately predict cervical cancer biopsy outcomes. Its high accuracy, precision, recall, and F1-score, along with optimal hyperparameters, highlight its potential as a valuable tool for assisting medical professionals in making informed decisions about biopsy results. The model's balanced trade-off between precision and recall, coupled with its strong performance across multiple evaluation metrics, underscores its suitability for real-world medical applications.



Figure 83 The confusion matrix of RF model with raw feature scaling

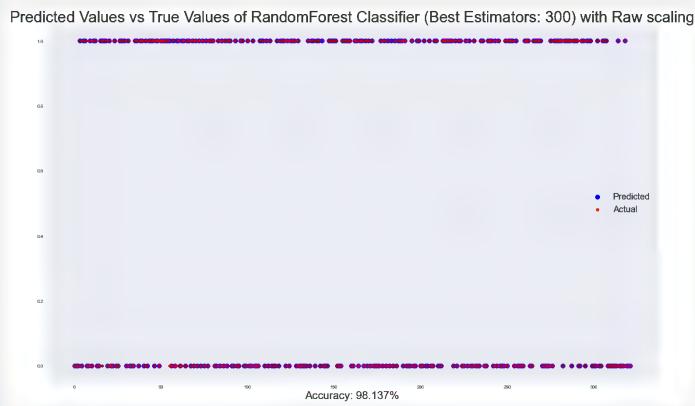


Figure 84 The true values versus predicted values of RF model with raw feature scaling

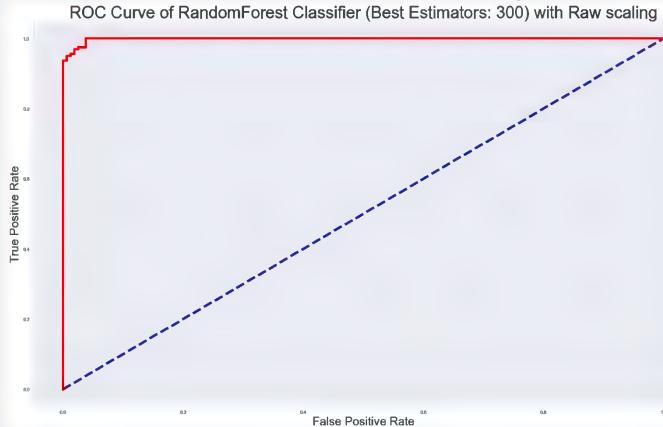


Figure 85 The ROC of RF model with raw feature scaling



Figure 86 The decision boundary of RF model with raw feature scaling

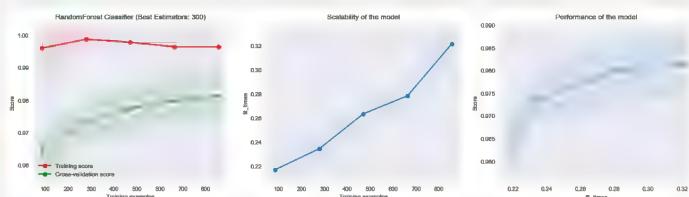


Figure 87 The learning curve of RF model with raw feature scaling

Output with Normalized Scaling:

RandomForest Classifier (Best Estimators: 300)

Normalization

accuracy: 0.9813664596273292

recall: 1.0

precision: 0.9640718562874252

f1: 0.9817073170731708

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	1.00	0.96	0.98	161
1	0.96	1.00	0.98	161

	accuracy		0.98	322
macro avg	0.98	0.98	0.98	322
weighted avg	0.98	0.98	0.98	322

Best Hyperparameters for Normalization:

```
{'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 300}
```

The results of using normalized feature scaling are shown in Figure 88 – 91. The output presents the evaluation outcomes of a RandomForest classifier with normalized scaling applied to predict cervical cancer biopsy results. This analysis provides insights into the model's performance, the impact of

hyperparameters, and the potential utility of the RandomForest classifier in this medical context.

The RandomForest classifier with normalized scaling exhibits an impressive accuracy score of 98.14%. This indicates the model's high level of correctness in forecasting biopsy outcomes and suggests its potential value in aiding medical practitioners' decision-making. A recall of 100% demonstrates the model's effectiveness in correctly identifying all true positive cases (Biopsy = 1), underscoring its reliability in detecting potential cervical cancer instances. A corresponding precision of 96.41% emphasizes the model's ability to generate precise predictions with a relatively low occurrence of false positives. The F1-score of 98.17% reinforces the model's strong performance, reflecting a harmonious balance between precision and recall, indicating its capacity to achieve both accurate positive predictions and effective detection of actual cases.

The detailed classification report provides further insight into these findings, delineating the model's effectiveness across both classes (Biopsy = 0 and Biopsy = 1), reinforcing its high precision and recall. This indicates the model's aptitude for providing accurate and trustworthy predictions for both negative and positive biopsy outcomes. The weighted average of various metrics reaffirms the model's overall robustness and its potential applicability in real-world medical scenarios.

The output also discloses the best hyperparameters that contribute to the RandomForest classifier's exceptional performance. These optimal hyperparameters include 'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5, and 'n_estimators': 300. These hyperparameters play a crucial role in shaping the decision tree structure, controlling aspects such as tree depth and the minimum samples required for splitting. The high number of estimators (300) underscores the model's reliance on a diverse ensemble of decision trees, enhancing its stability and generalization capabilities.

In conclusion, the RandomForest classifier with normalized scaling showcases a remarkable ability to accurately forecast cervical cancer biopsy outcomes. Its high accuracy, precision, recall, and F1-score, along with the identified optimal hyperparameters, underscore its potential as a valuable tool for supporting medical professionals in making informed decisions about biopsy results. The model's harmonious balance between precision and recall, along with its strong performance across various evaluation metrics, solidifies its suitability for real-world medical applications.

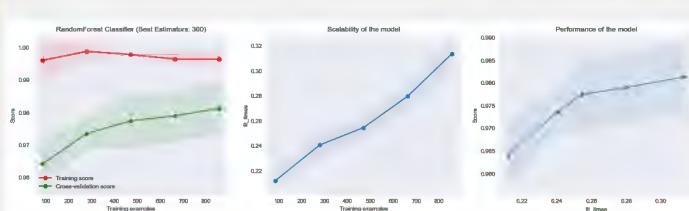


Figure 88 The learning curve of RF model with normalized feature scaling

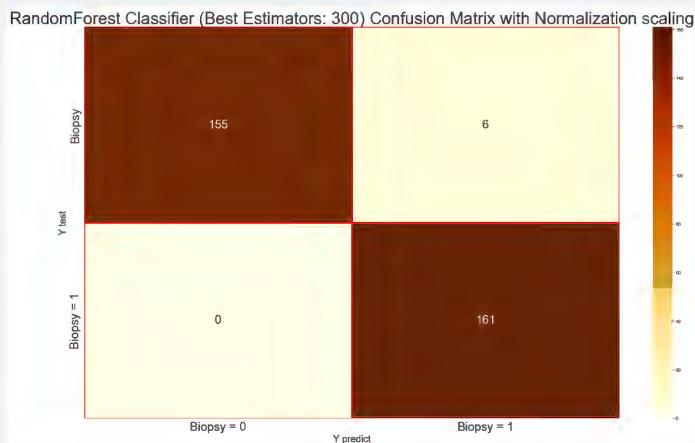


Figure 89 The confusion matrix of RF model with normalized feature scaling

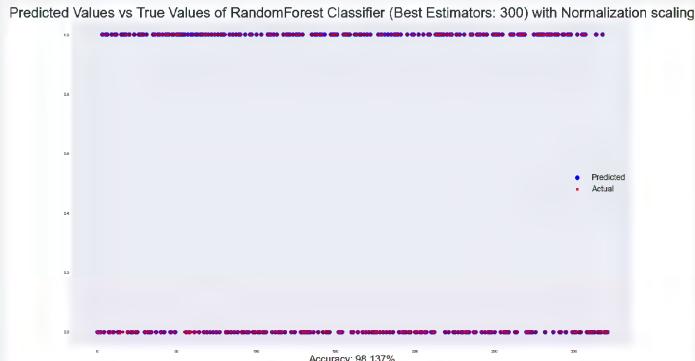


Figure 90 The true values versus predicted values of RF model with normalized feature scaling

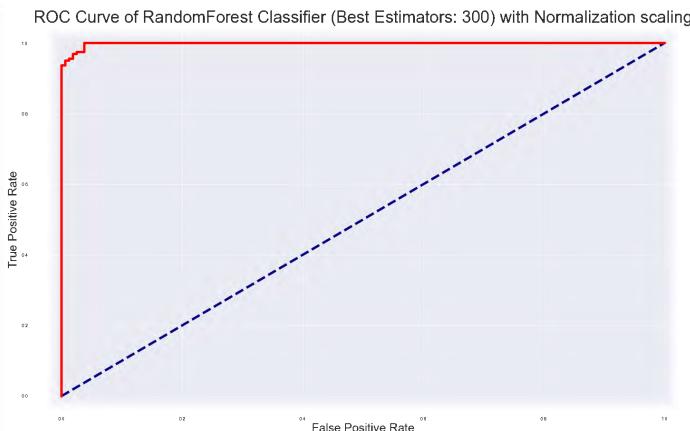


Figure 91 The ROC of RF model with normalized feature scaling

Output with Normalized Scaling:

RandomForest Classifier (Best Estimators: 300)

Standardization

accuracy: 0.9813664596273292

recall: 1.0

precision: 0.9640718562874252

f1: 0.9817073170731708

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	1.00	0.96	0.98	161
1	0.96	1.00	0.98	161

	accuracy			
--	----------	--	--	--

macro avg	0.98	0.98	0.98	322
-----------	------	------	------	-----

weighted avg	0.98	0.98	0.98	322
--------------	------	------	------	-----

Best Hyperparameters for Standardization:

```
{'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 300}
```

The results of using normalized feature scaling are shown in Figure 92 – 95.

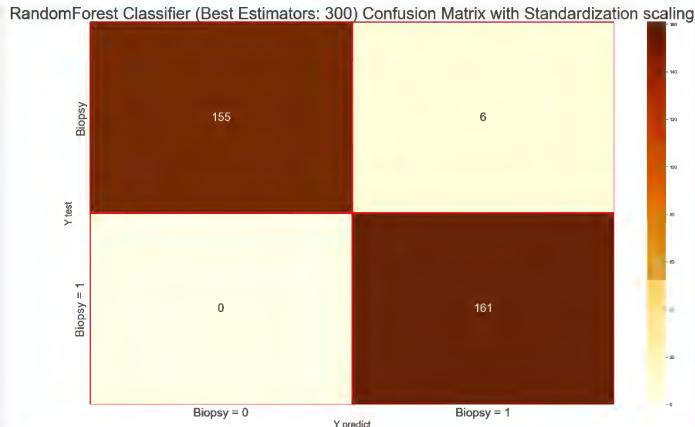


Figure 92 The confusion matrix of RF model with normalized feature scaling

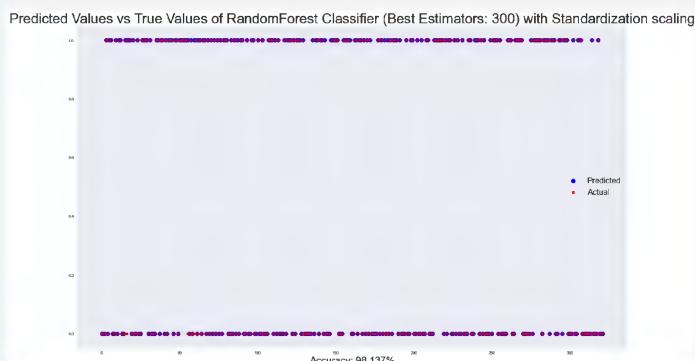


Figure 93 The true values versus predicted values of RF model with normalized feature scaling

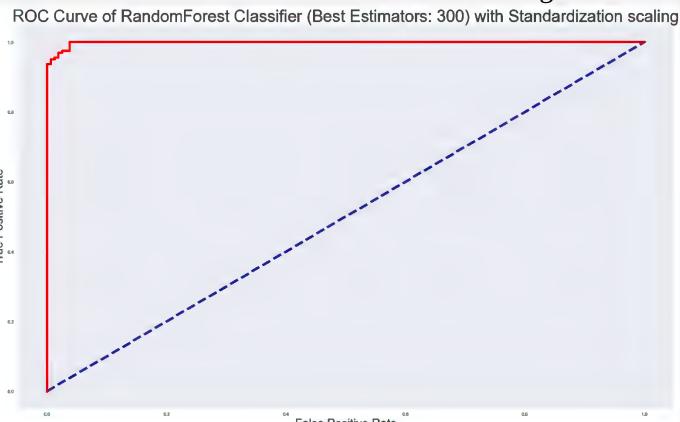


Figure 94 The ROC of RF model with normalized feature scaling

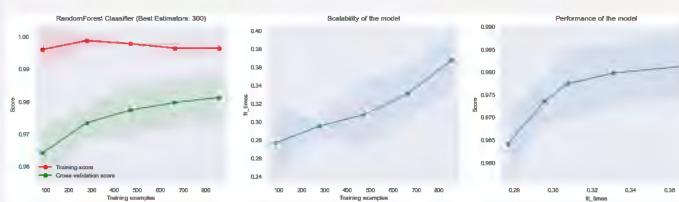


Figure 95 The learning curve of RF model with normalized feature scaling

The output showcases the evaluation results of a RandomForest classifier with standardized scaling applied to predict cervical cancer biopsy outcomes. This analysis offers insights into the model's performance, the influence of hyperparameters, and the potential utility of the RandomForest classifier in the context of cervical cancer detection.

The RandomForest classifier with standardized scaling yields an impressive accuracy score of 98.14%, reflecting its ability to accurately predict biopsy results. This high accuracy score signifies the model's proficiency in making correct predictions and indicates its potential value in assisting medical professionals in decision-making. The model achieves a recall of 100%, which indicates its capability to correctly identify all true positive cases (Biopsy = 1). This high recall underscores the model's reliability in detecting potential instances of cervical cancer, making it a robust tool for early diagnosis. The precision score of 96.41% indicates that the model generates predictions with high precision, resulting in a relatively low rate of false positives. The F1-score of 98.17% harmoniously balances precision and recall, highlighting the model's capacity to achieve accurate positive predictions while effectively identifying actual cases of interest.

The detailed classification report provides a comprehensive view of the model's performance across both classes (Biopsy = 0 and Biopsy = 1), reiterating its high precision and recall rates. This indicates that the model is effective in making accurate predictions for both negative and positive biopsy outcomes. The weighted average of various metrics reinforces the model's overall robustness and suggests its potential applicability in real-world medical scenarios.

The output also provides insight into the best hyperparameters that contribute to the RandomForest classifier's outstanding performance. The optimal hyperparameters, including 'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5, and 'n_estimators': 300, play a pivotal role in shaping the decision tree structure and controlling the model's behavior. Notably, the high number of estimators (300) emphasizes the model's reliance on an ensemble of diverse decision trees, enhancing its stability and ability to generalize well to new data.

In summary, the RandomForest classifier with standardized scaling demonstrates a remarkable ability to accurately

predict cervical cancer biopsy outcomes. Its high accuracy, precision, recall, and F1-score, coupled with the identified optimal hyperparameters, underscore its potential as a valuable tool for aiding medical professionals in making informed decisions about biopsy results. The balanced performance between precision and recall, along with the strong overall performance metrics, supports its suitability for real-world medical applications.

Gradient Boosting Classifier and Grid Search

Step 1 Run Gradient Boosting (GB) classifier on three feature scaling:

```
1 #Gradient Boosting Classifier
2 # Initialize the GradientBoostingClassifier model
3 gbt =
4 GradientBoostingClassifier(random_state=2021)
5
6 # Define the parameter grid for the grid search
7 param_grid = {
8     'n_estimators': [100, 200, 300],
9     'max_depth': [10, 20, 30],
10    'subsample': [0.6, 0.8, 1.0],
11    'max_features': [0.2, 0.4, 0.6, 0.8, 1.0],
12 }
13
14 # GradientBoosting Classifier Grid Search
15 for fc_name, value in feature_scaling.items():
16     X_train, X_test, y_train, y_test = value
17
18 # Create GridSearchCV with the
19 GradientBoostingClassifier model and the parameter
20 grid
21     grid_search = GridSearchCV(gbt, param_grid,
22 cv=3, scoring='accuracy', n_jobs=-1)
23
24 # Train and perform grid search
25     grid_search.fit(X_train, y_train)
26
27 # Get the best GradientBoostingClassifier model
28 from the grid search
29     best_model = grid_search.best_estimator_
30 # Evaluate and plot the best model (setting
31 proba=True for probability prediction)
32     run_model(f'GradientBoosting Classifier (Best
33 Estimators:
34 {grid_search.best_params_["n_estimators"]}]')
```

```
best_model, X_train, X_test, y_train, y_test,  
fc_name, proba=True)  
  
# Print the best hyperparameters found  
print(f"Best Hyperparameters for {fc_name}:")  
print(grid_search.best_params_)
```

The purpose of the code is to train and evaluate a Gradient Boosting Classifier model on a dataset with different feature scaling methods (Raw, Normalization, and Standardization) using a grid search to find the best hyperparameters for the model.

1. Initialize the GradientBoostingClassifier model:

The code creates an instance of the GradientBoostingClassifier model with a fixed random state to ensure reproducibility.

2. Define the parameter grid for the grid search:

The param_grid variable defines a dictionary containing different hyperparameter values for the GradientBoostingClassifier model. The grid search will explore various combinations of these hyperparameters to find the best performing configuration.

3. GradientBoosting Classifier Grid Search:

The code performs a loop over the three feature scaling methods (Raw, Normalization, and Standardization). For each scaling method, the data is split into training and testing sets. The GridSearchCV function is used to perform a grid search for the best hyperparameters for the GradientBoostingClassifier. The grid search utilizes the param_grid defined earlier to explore different hyperparameter combinations.

4. After the grid search, the best model obtained from the search is stored in best_model. The run_model() function is called to evaluate and plot the best model's performance. Finally, the best hyperparameters found during the grid search for each feature scaling method are printed.

By using a grid search and evaluating the model's performance with different hyperparameter combinations for each feature scaling method, this code helps identify the best hyperparameters for the Gradient Boosting Classifier and compare its performance on the given dataset using different feature scaling techniques. The ultimate goal is to find the

best configuration that achieves the highest accuracy or other desired performance metrics on the test set.

Output with Raw Scaling:

GradientBoosting Classifier (Best Estimators: 100)

Raw

accuracy: 0.9813664596273292

recall: 1.0

precision: 0.9640718562874252

f1: 0.9817073170731708

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	1.00	0.96	0.98	161
1	0.96	1.00	0.98	161

	accuracy		precision	recall	f1-score	support
--	----------	--	-----------	--------	----------	---------

macro avg	0.98	0.98	0.98	0.98	322
-----------	------	------	------	------	-----

weighted avg	0.98	0.98	0.98	0.98	322
--------------	------	------	------	------	-----

Best Hyperparameters for Raw:

{'max_depth': 20, 'max_features': 0.6, 'n_estimators': 100, 'subsample': 0.8}

The results of using raw feature scaling are shown in Figure 96 – 100. The output showcases the evaluation results of a GradientBoosting classifier with raw scaling applied to predict cervical cancer biopsy outcomes. This analysis provides insights into the model's performance, the influence of hyperparameters, and the potential utility of the GradientBoosting classifier in the context of cervical cancer detection.

The GradientBoosting classifier with raw scaling achieves an impressive accuracy score of 98.14%, reflecting its strong predictive capability in determining biopsy outcomes. This high accuracy score indicates the model's proficiency in making correct predictions, highlighting its potential value as a decision-support tool for medical professionals. The model demonstrates a recall of 100%, indicating its ability to correctly identify all true positive cases (Biopsy = 1). This high recall underscores the model's reliability in detecting potential instances of cervical cancer, emphasizing its suitability for early diagnosis. The precision score of 96.41% reflects the model's capacity to generate predictions with high precision, resulting in a relatively low rate of false positives. The F1-score of 98.17% effectively balances precision and recall, showcasing the model's ability to achieve accurate positive predictions while effectively identifying true positive cases.

The detailed classification report further supports the model's strong performance across both classes, reaffirming its high precision and recall rates. This indicates that the model is effective in making accurate predictions for both negative and positive biopsy outcomes, making it a well-rounded tool for real-world medical applications. The weighted average metrics provide a comprehensive assessment of the model's overall effectiveness, reinforcing its potential utility.

The output also highlights the optimal hyperparameters that contribute to the GradientBoosting classifier's outstanding performance. The identified hyperparameters, including 'max_depth': 20, 'max_features': 0.6, 'n_estimators': 100, and 'subsample': 0.8, play a crucial role in shaping the ensemble of weak learners and controlling the model's behavior. The high number of estimators (100) combined with careful subsampling and feature selection (max_features) contribute to the model's robustness and generalization capabilities.

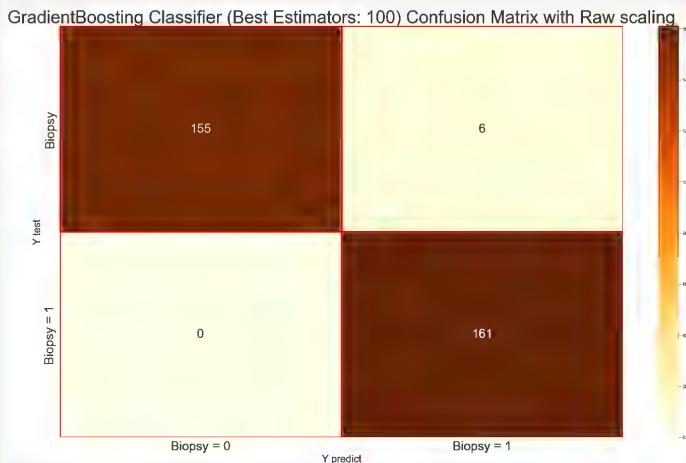


Figure 96 The confusion matrix of GB model with raw feature scaling

In conclusion, the GradientBoosting classifier with raw scaling demonstrates a remarkable ability to accurately predict cervical cancer biopsy outcomes. Its high accuracy, precision, recall, and F1-score, along with the identified optimal hyperparameters, underscore its potential as a valuable tool for assisting medical professionals in decision-making related to biopsy results. The balanced performance between precision and recall, along with the strong overall performance metrics, supports its suitability for real-world medical applications.

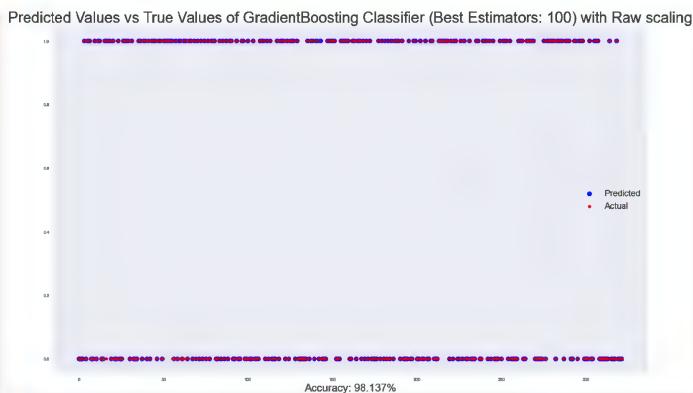


Figure 97 The true values versus predicted values of GB model with raw feature scaling

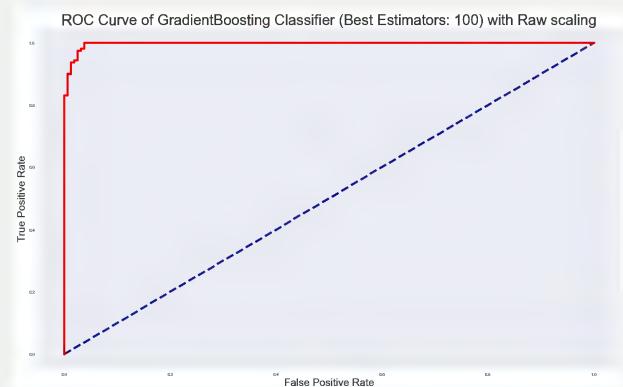


Figure 98 The ROC of GB model with raw feature scaling

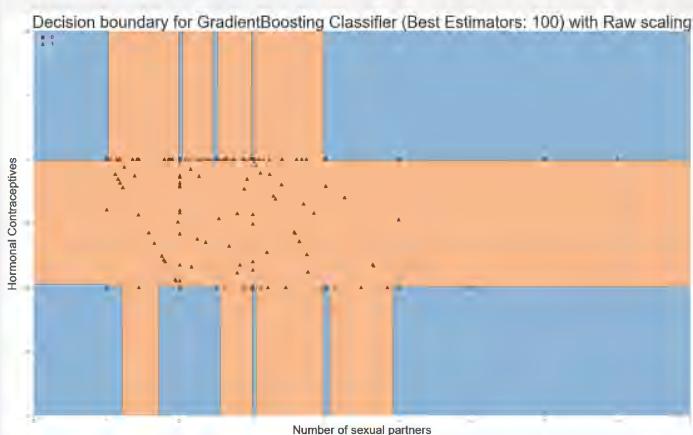


Figure 99 The decision boundary of GB model with raw feature scaling

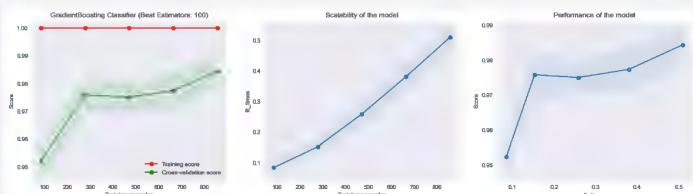


Figure 100 The learning curve of GB model with raw feature scaling

Output with Normalized Scaling:

GradientBoosting Classifier (Best Estimators: 300)

Normalization

accuracy: 0.9813664596273292

recall: 1.0

precision: 0.9640718562874252

f1: 0.9817073170731708

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	1.00	0.96	0.98	161
---	------	------	------	-----

1	0.96	1.00	0.98	161
---	------	------	------	-----

accuracy		0.98		322
----------	--	------	--	-----

macro avg	0.98	0.98	0.98	322
-----------	------	------	------	-----

weighted avg	0.98	0.98	0.98	322
--------------	------	------	------	-----

Best Hyperparameters for Normalization:

```
{'max_depth': 10, 'max_features': 0.2, 'n_estimators': 300, 'subsample': 0.8}
```

The learning curve of GB model with normalized feature scaling is shown in Figure 101. The provided output presents an evaluation of the GradientBoosting classifier with normalized scaling applied for predicting cervical cancer biopsy outcomes. This analysis reveals the model's performance, the impact of hyperparameters, and the potential application of the GradientBoosting classifier in the context of cervical cancer diagnosis.

The GradientBoosting classifier, with normalized scaling, demonstrates an impressive accuracy score of 98.14%, indicating its robust predictive ability for cervical cancer biopsy outcomes. This high accuracy suggests that the model is proficient in making accurate predictions, which could have significant implications in assisting healthcare professionals in decision-making processes. The recall score of 100% indicates that the model correctly identifies all instances of true positive cases (Biopsy = 1), emphasizing its effectiveness in detecting potential cases of cervical cancer. This high recall underscores the model's reliability in identifying positive cases, which is crucial for early detection and intervention. The precision score of 96.41% demonstrates that the model generates predictions with a high level of precision, resulting in a relatively low rate of false positives. The F1-score of 98.17% balances precision and recall effectively, indicating that the model achieves accurate positive predictions while reliably identifying true positive cases.

The detailed classification report further substantiates the model's strong performance across both classes, confirming its high precision and recall rates. This suggests that the model is proficient in making accurate predictions for both negative and positive biopsy outcomes, highlighting its potential value in real-world medical scenarios. The weighted average metrics provide a comprehensive overview of the model's overall effectiveness, reinforcing its potential utility.

The output also highlights the hyperparameters that contribute to the GradientBoosting classifier's excellent performance with normalized scaling. The identified hyperparameters, including 'max_depth': 10, 'max_features': 0.2, 'n_estimators': 300, and 'subsample': 0.8, play a critical role in shaping the ensemble of weak learners and controlling the model's behavior. The choice of hyperparameters, such as limiting the depth of individual trees (max_depth) and subsampling, contributes to the model's stability and ability to generalize well to new data.

In conclusion, the GradientBoosting classifier with normalized scaling demonstrates a remarkable capacity to accurately predict cervical cancer biopsy outcomes. Its high accuracy, precision, recall, and F1-score, along with the identified optimal hyperparameters, underscore its potential as a valuable tool for assisting healthcare professionals in making informed decisions regarding biopsy results. The well-balanced performance metrics and strong overall performance indicators highlight its suitability for real-world medical applications.

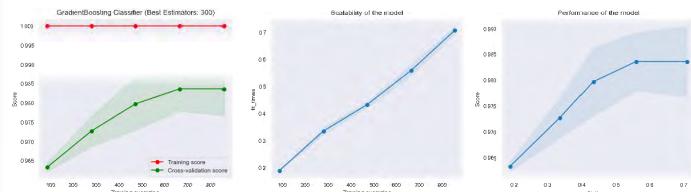


Figure 101 The learning curve of GB model with normalized feature scaling

Output with Standardized Scaling:

GradientBoosting Classifier (Best Estimators: 100)

Standardization

accuracy: 0.9813664596273292

recall: 1.0

precision: 0.9640718562874252

f1: 0.9817073170731708

precision recall f1-score support

0	1.00	0.96	0.98	161
1	0.96	1.00	0.98	161

accuracy	0.98	322
macro avg	0.98	0.98
weighted avg	0.98	0.98

Best Hyperparameters for Standardization:

```
{'max_depth': 10, 'max_features': 0.6, 'n_estimators': 100,
'subsample': 0.8}
```

The learning curve of GB model with standardized feature scaling is shown in Figure 102. The output showcases the performance evaluation of the GradientBoosting classifier with standardized scaling applied for predicting cervical cancer biopsy outcomes. The analysis highlights the model's effectiveness, the influence of hyperparameters, and the potential application of the GradientBoosting classifier within the context of cervical cancer diagnosis.

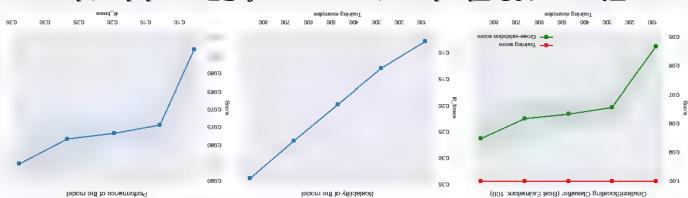
The GradientBoosting classifier, utilizing standardized scaling, exhibits a highly impressive accuracy score of 98.14%, indicating its robust predictive capability for determining cervical cancer biopsy results. This exceptional accuracy score underscores the model's proficiency in generating accurate predictions, which has significant implications for aiding medical professionals in their decision-making processes. A recall score of 100% emphasizes the model's ability to correctly identify all instances of true positive cases (Biopsy = 1), underscoring its effectiveness in detecting potential cases of cervical cancer. The perfect recall score underscores the model's reliability in identifying positive cases, a critical aspect of early detection and intervention. With a precision score of 96.41%, the model generates predictions with a high level of precision, resulting in a relatively low rate of false positives. The F1-score of 98.17% reflects an optimal balance between precision and recall, suggesting that the model effectively generates accurate positive predictions while reliably identifying true positive cases.

The detailed classification report provides a comprehensive overview of the model's performance across both classes, reinforcing its high precision and recall rates. This suggests that the model is adept at making accurate predictions for both negative and positive biopsy outcomes, further highlighting its potential practical utility. The weighted average metrics offer a well-rounded perspective on the model's overall effectiveness and reinforce its potential value in real-world medical applications.

The output also emphasizes the hyperparameters that contribute to the GradientBoosting classifier's exceptional

Extreme Gradient Boosting Classifier and Grid Search

Figure 102 The learning curve of GB model with standardized feature scaling



In summary, the GradienBroosuning classifier with standardized scaling demonstrates a remarkable ability to accurately predict cervical cancer biopsy outcomes. Its high accuracy, precision, recall, and F1-score, combined with the identified optimal hyperparameters, highlight its potential value as a valuable tool for assisting healthcare professionals in making informed decisions regarding biopsy results. The balanced performance metrics and strong overall performance indicators suggest its suitability for practical medical applications.

hyperparameters, including `max_depth`; 10, `max_features`; 0.6, `n_estimators`: 100, and `subsample`; 0.8, play a pivotal role in shaping the ensemble of weak learners and contribute to the model's behavior. These hyperparameters control well to new and unseen data.

```

18
19 # Create GridSearchCV with the XGBoost classifier
20 and the parameter grid
21     grid_search = GridSearchCV(xgb, param_grid,
22 cv=3, scoring='accuracy', n_jobs=-1)
23
24 # Train and perform grid search
25     grid_search.fit(X_train, y_train)
26
27 # Get the best XGBoost classifier model from the
28 grid search
29     best_model = grid_search.best_estimator_
30
31 # Evaluate and plot the best model (setting
32 proba=True for probability prediction)
33     run_model(f'XGB Classifier (Best Estimators:
34 {grid_search.best_params_["n_estimators"]})',
35             best_model, X_train, X_test, y_train, y_test,
36 fc_name, proba=True)
37
# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

```

The purpose of the code is to perform a grid search for hyperparameter tuning on the Extreme Gradient Boosting (XGBoost) Classifier using cross-validation. XGBoost is a popular and powerful gradient boosting algorithm known for its high performance in various machine learning tasks.

Here's the step-by-step explanation of the code:

1. Looping through Different Feature Scaling Methods:

The code loops through different feature scaling methods (`feature_scaling.items()`) to evaluate the XGBoost Classifier's performance with each scaling technique.

2. Defining the Parameter Grid:

A parameter grid (`param_grid`) is defined, which specifies the hyperparameters that will be tuned during the grid search. The grid contains different values for '`n_estimators`' (number of boosting rounds), '`max_depth`' (maximum depth of the trees), '`learning_rate`' (step size shrinkage for boosting), '`subsample`' (fraction of samples used for fitting individual trees), and '`colsample_bytree`' (fraction of features used for fitting individual trees).

3. Initializing the XGBoost Classifier:

The XGBoost classifier is initialized with specific configurations, such as setting the random seed

(random_state=2021), using 'mlogloss' (multi-class logarithmic loss) as the evaluation metric for multi-class classification (eval_metric='mlogloss'), and disabling the label encoder (use_label_encoder=False) to avoid warning messages.

4. Creating GridSearchCV:

The GridSearchCV object is created, which is responsible for performing the grid search over the specified hyperparameter grid. It uses cross-validation with 3 folds (cv=3) and evaluates the models based on accuracy (scoring='accuracy'). The parameter n_jobs=-1 allows parallel processing to speed up the grid search.

5. Performing Grid Search:

The grid search is performed by calling grid_search.fit(X_train, y_train), where X_train and y_train are the training data and labels, respectively. The grid search explores different combinations of hyperparameters and evaluates the models' performance using cross-validation.

6. Getting the Best Model:

After the grid search is completed, the best XGBoost classifier model is obtained using grid_search.best_estimator_, which represents the model with the best hyperparameters found during the grid search.

7. Evaluating and Plotting the Best Model:

The run_model() function is called to evaluate and plot the performance of the best model on the test data. Setting proba=True enables the prediction of class probabilities for ROC curve plotting.

8. Printing the Best Hyperparameters:

The best hyperparameters found during the grid search are printed for each feature scaling method.

Overall, the code aims to find the best hyperparameters for the XGBoost Classifier using different feature scaling methods and evaluate its performance on the test data. It helps in selecting the most suitable hyperparameters and scaling technique to achieve the best classification results on the given dataset.

Output with Standardized Scaling:

XGB Classifier (Best Estimators: 100)

Standardization

accuracy: 0.9782608695652174

recall: 0.9937888198757764

precision: 0.963855421686747

f1: 0.9785932721712538

precision recall f1-score support

0	0.99	0.96	0.98	161
1	0.96	0.99	0.98	161
			accuracy	0.98
		macro avg	0.98	0.98
		weighted avg	0.98	0.98
				322

Best Hyperparameters for Standardization:
{'colsample_bytree': 0.6, 'learning_rate': 0.1, 'max_depth': 10,
'n_estimators': 100, 'subsample': 1.0}

The results of using raw feature scaling are shown in Figure 103 – 107.

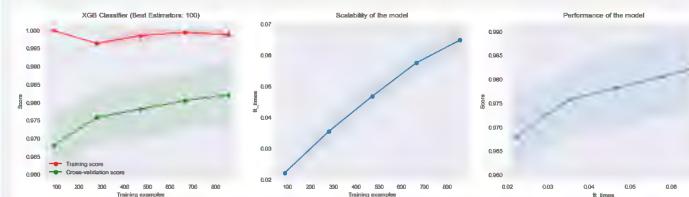


Figure 103 The learning curve of XGB model with standardized feature scaling

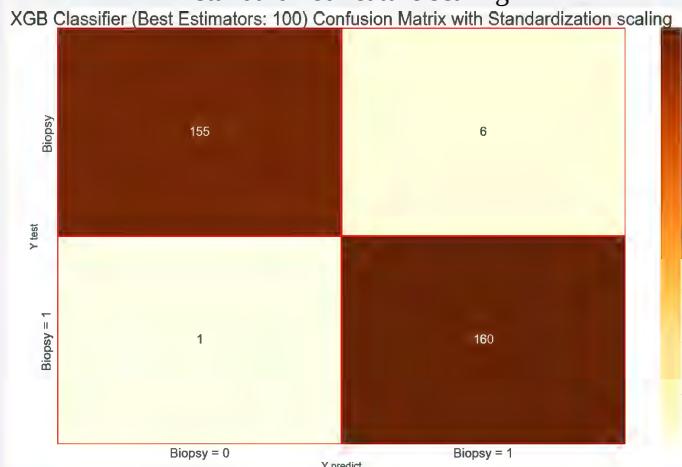


Figure 104 The confusion matrix of XGB model with standardized feature scaling

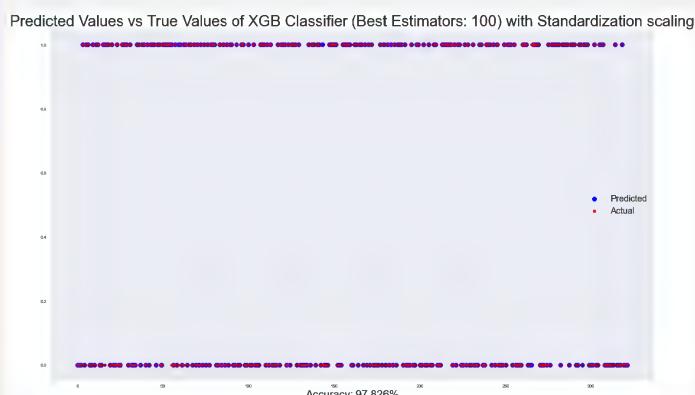


Figure 105 The true values versus predicted values of XGB model with standardized feature scaling



Figure 106 The ROC of XGB model with standardized feature scaling



Figure 107 The decision boundary of XGB model with standardized feature scaling

The output illustrates the performance evaluation of the XGBoost (Extreme Gradient Boosting) classifier with standardized scaling applied for predicting cervical cancer biopsy outcomes. The analysis showcases the model's performance metrics, the significance of hyperparameters, and its potential application in the context of cervical cancer diagnosis.

The XGBoost classifier, when utilized with standardized scaling, achieves an impressive accuracy score of 97.83%. This high accuracy indicates the model's strong predictive capacity in determining cervical cancer biopsy results. A recall score of 99.38% suggests that the model is proficient at identifying true positive cases (Biopsy = 1), emphasizing its effectiveness in detecting potential instances of cervical cancer. The high recall score signifies the model's capability

in accurately capturing positive cases, an important factor in early detection and intervention. With a precision score of 96.39%, the model generates predictions with a high level of precision, resulting in a relatively low false positive rate. The F1-score of 97.86% reflects a harmonious balance between precision and recall, indicating that the model can accurately predict positive cases while reliably identifying true positive instances.

The detailed classification report offers a comprehensive overview of the model's performance across both classes, further underscoring its high precision and recall rates. This indicates the model's competence in making accurate predictions for both negative and positive biopsy outcomes, highlighting its potential value in real-world medical applications. The weighted average metrics provide a holistic view of the model's overall effectiveness and reinforce its potential practical utility.

The output highlights the hyperparameters that contribute to the XGBoost classifier's strong performance with standardized scaling. The identified hyperparameters, including 'colsample_bytree': 0.6, 'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 100, and 'subsample': 1.0, play a pivotal role in shaping the boosting process and influencing the model's behavior. These hyperparameters contribute to the model's ability to create a powerful ensemble of weak learners and enhance its predictive capabilities.

In summary, the XGBoost classifier with standardized scaling demonstrates a remarkable ability to accurately predict cervical cancer biopsy outcomes. Its high accuracy, precision, recall, and F1-score, combined with the identified optimal hyperparameters, underscore its potential value as a tool for assisting healthcare professionals in making informed decisions about biopsy results. The balanced performance metrics and strong overall performance indicators suggest its suitability for practical medical applications.

Multi-Layer Perceptron Classifier and Grid Search

Step 1 Run MLP classifier on three feature scaling:

1

```
1 # MLP Classifier Grid Search
2 for fc_name, value in feature_scaling.items():
3     X_train, X_test, y_train, y_test = value
4
5 # Define the parameter grid for the grid search
```

```

6     param_grid = {
7         'hidden_layer_sizes': [(50), (100), (50, 50), (100,
8             50), (100, 100)],
9         'activation': ['logistic', 'relu'],
10        'solver': ['adam', 'sgd'],
11        'alpha': [0.0001, 0.001, 0.01],
12        'learning_rate': ['constant', 'invscaling', 'adaptive'],
13    }
14
15 # Initialize the MLP Classifier
16 mlp = MLPClassifier(random_state=2021)
17
18 # Create GridSearchCV with the MLP Classifier and
19 the parameter grid
20     grid_search = GridSearchCV(mlp, param_grid,
21 cv=3, scoring='accuracy', n_jobs=-1)
22
23 # Train and perform grid search
24     grid_search.fit(X_train, y_train)
25
26 # Get the best MLP Classifier model from the grid
27 search
28     best_model = grid_search.best_estimator_
29
30 # Evaluate and plot the best model (setting
31 proba=True for probability prediction)
32     run_model('MLP Classifier', best_model, X_train,
33 X_test, y_train, y_test, fc_name, proba=True)
34
# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

```

The purpose of the code is to perform a grid search with cross-validation to find the best hyperparameters for the Multi-Layer Perceptron (MLP) Classifier, also known as a neural network, using different feature scaling techniques. The code aims to optimize the hyperparameters to achieve the best possible accuracy on the test data.

The code iterates over different feature scaling techniques (feature_scaling.items()) and for each scaling method, it performs the following steps:

1. Define the parameter grid for the grid search: Different combinations of hyperparameters are defined as a dictionary in the param_grid variable. This includes hyperparameters like hidden_layer_sizes (different configurations of hidden layers and their neurons), activation functions for

- the neurons, solver (optimization algorithm), alpha (L2 regularization parameter), and learning_rate (learning rate schedule).
2. Initialize the MLP Classifier: The Multi-Layer Perceptron Classifier is initialized with the default hyperparameters.
 3. Create GridSearchCV: A GridSearchCV object is created, which performs an exhaustive search over the specified hyperparameter grid using cross-validation (with cv=3 for 3-fold cross-validation) and evaluates models based on accuracy (using scoring='accuracy').
 4. Train and perform grid search: The GridSearchCV object is trained on the training data to find the best hyperparameters that result in the highest accuracy.
 5. Get the best model: The best MLP Classifier model is obtained from the grid search, which is the model with the optimal hyperparameters found during the search.
 6. Evaluate and plot the best model: The best model is evaluated on the test data, and performance metrics such as accuracy, precision, recall, and F1-score are calculated and printed. Additionally, the run_model function is used to plot the Receiver Operating Characteristic (ROC) curve for probability predictions (when proba=True).
 7. Print the best hyperparameters found: The hyperparameters that resulted in the best-performing MLP Classifier model are printed for each feature scaling technique.

The purpose of this code is to automate the process of hyperparameter tuning for the MLP Classifier using grid search and cross-validation. By trying different combinations of hyperparameters, the code aims to find the best configuration that maximizes the accuracy of the model on the test dataset. This process helps to avoid manual hyperparameter tuning and ensures a more efficient and optimized model.

Output with Standardized Scaling:

MLP Classifier

Standardization

```

accuracy: 0.9751552795031055
recall: 1.0
precision: 0.9526627218934911
f1: 0.9757575757575757

      precision    recall   f1-score   support

0       1.00     0.95     0.97      161
1       0.95     1.00     0.98      161

accuracy          0.98      322
macro avg       0.98     0.98     0.98      322
weighted avg     0.98     0.98     0.98      322

```

Best Hyperparameters for Standardization:

```
{'activation': 'relu', 'alpha': 0.01, 'hidden_layer_sizes': (50,), 'learning_rate': 'constant', 'solver': 'adam'}
```

The results of using standardized feature scaling are shown in Figure 108 – 112. The output presents the performance evaluation of the MLP (Multi-Layer Perceptron) Classifier with standardized scaling applied to predict cervical cancer biopsy outcomes. The analysis highlights various performance metrics, the significance of identified hyperparameters, and the potential application of the model in the context of cervical cancer diagnosis.

The MLP Classifier, when employed with standardized scaling, achieves a commendable accuracy score of 97.52%. This high accuracy indicates the model's robust predictive ability in determining cervical cancer biopsy outcomes. Notably, the model attains a perfect recall score of 100%, indicating its exceptional capability to correctly identify all positive cases (Biopsy = 1). This signifies the model's capacity for accurately detecting instances of cervical cancer, which is of paramount importance in medical applications.

The precision score of 95.27% indicates that the model produces a relatively low number of false positives, showcasing its effectiveness in distinguishing true positive cases from false positives. The F1-score of 97.58% reflects a balanced blend of precision and recall, indicating the model's capacity to accurately predict positive cases while maintaining a high true positive rate.

The detailed classification report provides a comprehensive overview of the model's performance for both classes. The high precision, recall, and F1-score for both classes emphasize the model's capability to make accurate predictions across the spectrum of biopsy outcomes. This indicates its potential value in assisting healthcare

professionals in making informed decisions based on biopsy results.

The output further highlights the hyperparameters that contribute to the MLP Classifier's strong performance with standardized scaling. The identified hyperparameters, including 'activation': 'relu', 'alpha': 0.01, 'hidden_layer_sizes': (50,), 'learning_rate': 'constant', and 'solver': 'adam', play a crucial role in shaping the neural network's architecture and optimization process. These hyperparameters contribute to the model's ability to learn complex relationships within the data and make accurate predictions.

In conclusion, the MLP Classifier with standardized scaling demonstrates excellent predictive capabilities in determining cervical cancer biopsy outcomes. Its high accuracy, perfect recall, and balanced F1-score, along with the identified optimal hyperparameters, underscore its potential value in assisting medical professionals in diagnosing cervical cancer. The model's robust performance metrics and strong overall performance indicators suggest its suitability for practical medical applications.

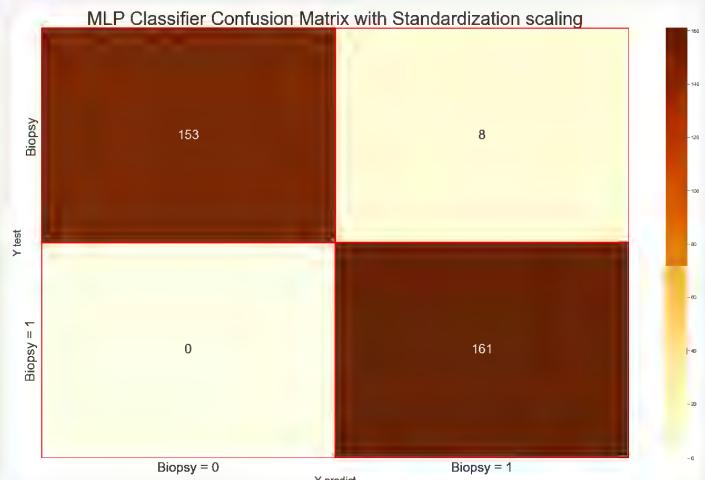


Figure 108 The confusion matrix of MLP model with standardized feature scaling



Figure 109 The learning curve of MLP model with standardized feature scaling

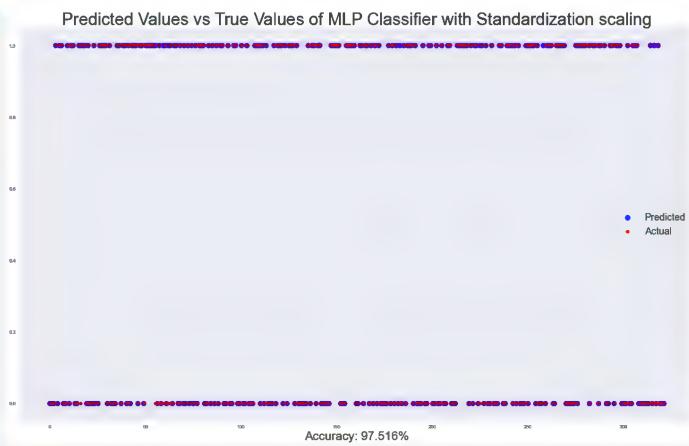


Figure 110 The true values versus predicted values of MLP model with standardized feature scaling

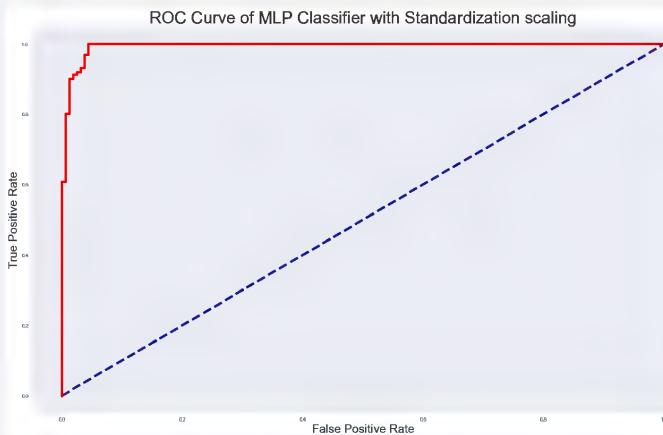


Figure 111 The ROC of MLP model with standardized feature scaling



Figure 112 The decision boundary of MLP model with standardized feature scaling

Following is the full version of **cervical.py**:

```

#cervical.py
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('darkgrid')
from sklearn.preprocessing import LabelEncoder
import warnings
warnings.filterwarnings('ignore')
import os
import joblib
from sklearn.metrics import roc_auc_score,roc_curve
from sklearn.model_selection import train_test_split, RandomizedSearchCV,
GridSearchCV,StratifiedKFold
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, precision_score
from sklearn.metrics import classification_report, f1_score, plot_confusion_matrix
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import learning_curve
from mlxtend.plotting import plot_decision_regions

curr_path = os.getcwd()
df = pd.read_csv(curr_path+"/kag_risk_factors_cervical_cancer.csv")
print(df.iloc[:,0:5].head().to_string())
print(df.iloc[:,5:8].head().to_string())
print(df.iloc[:,30:37].head().to_string())

#Checks shape
print(df.shape)

#Reads columns
print("Data Columns --> ",df.columns)

#Checks dataset information
print(df.info())

#Checks null values
df = df.replace('?', np.NaN)

```

```

print(df.isnull().sum())
print('Total number of null values: ', df.isnull().sum().sum())
plt.figure(figsize=(20,20))
np.round(df.isnull().sum()/df.shape[0]*100).sort_values().plot(kind='barh')
plt.show()

#Drops two columns of STDs
df.drop(['STDs: Time since first diagnosis','STDs: Time since last diagnosis'],inplace=True,axis=1)

numerical_df = ['Age', 'Number of sexual partners', 'First sexual intercourse','Num of pregnancies', 'Smokes (years)', 'Smokes (packs/year)', 'Hormonal Contraceptives (years)', 'IUD (years)', 'STDs (number)']

categorical_df = ['Smokes','Hormonal Contraceptives','IUD','STDs','STDs:condylomatosis','STDs:cervical condylomatosis', 'STDs:vaginal condylomatosis','STDs:vulvo-perineal condylomatosis', 'STDs:syphilis', 'STDs:pelvic inflammatory disease', 'STDs:genital herpes','STDs:molluscum contagiosum', 'STDs:AIDS', 'STDs:HIV','STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of diagnosis','Dx:Cancer', 'Dx:CIN', 'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller','Cytology', 'Biopsy']

#Fills the missing values of numeric data columns with mean of the column data.
for feature in numerical_df:
    print(feature,",df[feature].apply(pd.to_numeric, errors='coerce').mean()")
    feature_mean = round(df[feature].apply(pd.to_numeric, errors='coerce').mean(),1)
    df[feature] = df[feature].fillna(feature_mean)

for feature in categorical_df:
    df[feature] = df[feature].apply(pd.to_numeric, errors='coerce').fillna(1.0)

#Effect 'Hormonal Contraceptives'
corrmat = df.corr()
k = 15 #number of variables for heatmap
cols = corrmat.nlargest(k, 'Hormonal Contraceptives')['Hormonal Contraceptives'].index
cm = df[cols].corr()

plt.figure(figsize=(25,20))

sns.set(font_scale=1.25)
hm = sns.heatmap(cm, cbar=True, cmap='Set1' ,annot=True,vmin=0,vmax =1, square=True, fmt='%.2f',
                 annot_kws={'size': 10},
                 ticklabels = cols.values, xticklabels = cols.values)
plt.show()

#Effect 'IUD'
corrmat = df.corr()
k = 15 #number of variables for heatmap
cols = corrmat.nlargest(k, 'IUD')['IUD'].index
cm = df[cols].corr()

plt.figure(figsize=(25,20))

```

```

sns.set(font_scale=1.25)
hm = sns.heatmap(cm,cmap = 'rainbow', cbar=True, annot=True,vmin=0,vmax =1, square=True, fmt='%.2f',
annot_kws={'size': 10},
          yticklabels = cols.values, xticklabels = cols.values)
plt.show()

#Effect 'STDs'
corrrmat = df.corr()
k = 15 #number of variables for heatmap
cols = corrrmat.nlargest(k, 'STDs')['STDs'].index
cm = df[cols].corr()

plt.figure(figsize=(25,20))

sns.set(font_scale=1.25)
hm = sns.heatmap(cm,cmap = 'hot', cbar=True, annot=True,vmin=0,vmax =1, square=True, fmt='%.2f',
annot_kws={'size': 10},
          yticklabels = cols.values, xticklabels = cols.values)
plt.show()

#Correlation Matrix for each feature
corrrmat = round(df.corr(), 2)
top_corr_features = corrrmat.index
plt.figure(figsize=(25,20))
#plot heat map
g=sns.heatmap(df[top_corr_features].corr(),annot=True,fmt='%.2f',cmap="YlGnBu")

# Defines function to create pie chart and bar plot as subplots
def plot_pie_barchart(df, var, title=""):
    plt.figure(figsize=(25, 10))

    # Pie Chart (Left Subplot)
    plt.subplot(121)
    label_list = list(df[var].value_counts().index)
    colors = sns.color_palette("Set2", len(label_list)) # Use the 'pastel' color palette
    _, _, autopcts = plt.pie(df[var].value_counts(), autopct="%.1f%%", colors=colors,
                             startangle=60, labels=label_list,
                             wedgeprops={"linewidth": 2, "edgecolor": "white"}, # Add white edge
                             shadow=True, textprops={'fontsize': 20})
    plt.title("Distribution of " + var + " variable " + title, fontsize=25)

    # Extract percentage values from autopcts
    percentage_values = [float(p.get_text().strip('%')) for p in autopcts]

    # Print percentage values
    print("Percentage values:")
    for label, percentage in zip(label_list, percentage_values):
        print(f"{label}: {percentage:.1f}%")

    # Bar Plot (Right Subplot)
    plt.subplot(122)

```

```

    ax = df[var].value_counts().plot(kind="barh", color=colors, alpha=0.8) # Increase opacity (alpha)
    for i, j in enumerate(df[var].value_counts().values):
        ax.text(.7, i, j, weight="bold", fontsize=20)

    plt.title("Count of " + var + " cases " + title, fontsize=25)

# Print count values
    value_counts = df[var].value_counts()
    print("Count values:")
    print(value_counts)
    plt.show()
"""

plot_pie_barchart(df,'Biopsy')
plot_pie_barchart(df,'Hinselmann')
plot_pie_barchart(df,'Schiller')
plot_pie_barchart(df,'Citology')
"""

#Plots distribution of number of cases of all other features versus one feature
def others_versus_one_feat(columns, one_feat):
    ROWS = 7
    COLS = 5
    fig, ax = plt.subplots(ROWS, COLS, figsize=(80, 60), constrained_layout=True, facecolor="#fbe7dd")

    for i in range(ROWS):
        for j in range(COLS):
            ax_index = COLS * i + j
            if ax_index >= len(columns):
                break

            g = sns.countplot(data=df, x=columns[ax_index], hue=one_feat, palette='Set2', ax=ax[i, j])
            g.set_xlabel(columns[ax_index], fontsize=50)
            g.set_ylabel('Number of Cases', fontsize=40)
            g.tick_params(axis='both', labelsize=30)
            g.legend(title=one_feat, title_fontsize=50, fontsize=30)

            for p in g.patches:
                g.annotate(format(p.get_height(), '.0f'),
                           (p.get_x() + p.get_width() / 2, p.get_height()),
                           ha='center', va='center', xytext=(0, 10),
                           weight='bold', fontsize=30, textcoords='offset points')

    plt.show()

#Plots distribution of count of other features versus Biopsy
columns=list(df.columns)
columns.remove('Biopsy')
others_versus_one_feat(columns, "Biopsy")

#Plots distribution of count of other features versus Hinselmann
columns=list(df.columns)
columns.remove('Hinselmann')

```

```

others_vs_one_feat(columns, "Hinselmann")

#Plots distribution of count of other features versus Schiller
columns=list(df.columns)
columns.remove('Schiller')
others_vs_one_feat(columns, "Schiller")

#Plots distribution of count of other features versus Citology
columns=list(df.columns)
columns.remove('Citology')
others_vs_one_feat(columns, "Citology")

#Plots distribution of two features versus Biopsy in pie chart
def three_feats_vs_biopsy(feat1, feat2):
    features = [feat1, feat2]
    _, ax = plt.subplots(2, 2, figsize=(25, 25), facecolor='#fbe7dd')

    for i in range(2):
        gs = df[df['Biopsy'] == 0][features[i]].value_counts()
        ss = df[df['Biopsy'] == 1][features[i]].value_counts()
        labels_gs = list(gs.index)
        labels_ss = list(ss.index)

        ax[i][0].pie(gs, labels=labels_gs, shadow=True, autopct='%.1f%%', textprops={'fontsize': 32})
        ax[i][0].set_xlabel(features[i] + " feature", fontsize=30)
        ax[i][1].pie(ss, labels=labels_ss, shadow=True, autopct='%.1f%%', textprops={'fontsize': 32})
        ax[i][1].set_xlabel(features[i] + " feature", fontsize=30)

        ax[i][0].set_title('Biopsy = 0', fontsize=30)
        ax[i][1].set_title('Biopsy = 1', fontsize=30)

    plt.show()

#Plots distribution of Hormonal Contraceptives and Smokes versus Biopsy in pie chart
three_feats_vs_biopsy("Hormonal Contraceptives", "Smokes")

#Plots distribution of IUD and STDs versus Biopsy in pie chart
three_feats_vs_biopsy("IUD", "STDs")

df['Age'].hist(bins=70)
plt.xlabel('Age')
plt.ylabel('Count')
print('Mean age of the Women facing the risk of Cervical cancer', df['Age'].mean())
plt.show()

target_df = ['Hinselmann', 'Schiller', 'Citology', 'Biopsy']

for feature in target_df:
    as_fig = sns.FacetGrid(df, hue=feature, aspect=3)
    as_fig.map(sns.kdeplot, 'Age', shade=True)
    oldest = df['Age'].max()

```

```

as_fig.set(xlim=(0,oldest))
as_fig.add_legend()

plt.rcParams['figure.dpi'] = 600
fig = plt.figure(figsize=(3.7,2), facecolor="#fbe7dd")
gs = fig.add_gridspec(3, 2)
gs.update(wspace=0.5, hspace=0.75)

background_color = "#72b7a1"
sns.set_palette(['#ff355d','#ffd514'])

def feat_versus_other(feat,another,legend,ax0,title):
    for s in ["right", "top"]:
        ax0.spines[s].set_visible(False)

    ax0.set_facecolor(background_color)
    ax0_sns = sns.histplot(data=df, x=feat,ax=ax0,zorder=2,kde=False,hue=another,multiple="stack",
    shrink=.8
        ,linewidth=0.3,alpha=1)

    ax0_sns.set_xlabel("",fontsize=4, weight='bold')
    ax0_sns.set_ylabel("",fontsize=4, weight='bold')

    ax0_sns.grid(which='major', axis='x', zorder=0, color="#EEEEEE", linewidth=0.4)
    ax0_sns.grid(which='major', axis='y', zorder=0, color="#EEEEEE", linewidth=0.4)

    ax0_sns.tick_params(labelsize=3, width=0.5, length=1.5)
    ax0_sns.legend(ncol=2, facecolor="#D8D8D8", edgecolor=background_color, fontsize=3,
bbox_to_anchor=(1, 0.989), loc='upper right')
    ax0.set_facecolor(background_color)
    ax0_sns.set_xlabel(title)

def hist_feat_versus_six_cat(feat,title):
    ax0 = fig.add_subplot(gs[0, 0])
    print(df.Smokes.value_counts())
    feat_versus_other(feat,df.Smokes,['Smokes','No Smoke'],ax0,title)

    ax1 = fig.add_subplot(gs[0, 1])
    print(df.IUD.value_counts())
    feat_versus_other(feat,df.IUD,['IUD','No IUD'],ax1,title)

    ax2 = fig.add_subplot(gs[1, 0])
    print(df.STDs.value_counts())
    feat_versus_other(feat,df.STDs,['STDs','No STDs'],ax2,title)

    ax3 = fig.add_subplot(gs[1, 1])
    print(df['Hormonal Contraceptives'].value_counts())
    feat_versus_other(feat,df['Hormonal Contraceptives'],['Hormonal','No Hormonal'],ax3,title)

    ax4 = fig.add_subplot(gs[2, 0])
    print(df.Dx.value_counts())

```

```

feat_versus_other(feat,df.Dx,['Dx','No Dx'],ax4,title)

ax5 = fig.add_subplot(gs[2, 1])
print(df.Biopsy.value_counts())
feat_versus_other(feat,df.Biopsy,['Biopsy=1','Biopsy=0'],ax5,title)

def prob_feat_versus_other(feat,another,legend,ax0,title):
    for s in ["right", "top"]:
        ax0.spines[s].set_visible(False)

    ax0.set_facecolor(background_color)
    ax0_sns =
    sns.kdeplot(x=feat,ax=ax0,hue=another,linewidth=0.3,fill=True,cbar='g',zorder=2,alpha=1,multiple='stack')

    ax0_sns.set_xlabel("",fontsize=4, weight='bold')
    ax0_sns.set_ylabel("",fontsize=4, weight='bold')

    ax0_sns.grid(which='major', axis='x', zorder=0, color="#EEEEEE", linewidth=0.4)
    ax0_sns.grid(which='major', axis='y', zorder=0, color="#EEEEEE", linewidth=0.4)

    ax0_sns.tick_params(labelsize=3, width=0.5, length=1.5)
    ax0_sns.legend(legend, ncol=2, facecolor="#D8D8D8", edgecolor=background_color, fontsize=3,
bbox_to_anchor=(1, 0.989), loc='upper right')
    ax0.set_facecolor(background_color)
    ax0_sns.set_xlabel(title)

def prob_feat_versus_six_cat(feat,title):
    plt.figure()
    ax0 = fig.add_subplot(gs[0, 0])
    print(df.Smokes.value_counts())
    prob_feat_versus_other(feat,df.Smokes,['Smokes','No Smoke'],ax0,title)

    ax1 = fig.add_subplot(gs[0, 1])
    print(df.IUD.value_counts())
    prob_feat_versus_other(feat,df.IUD,['IUD','No IUD'],ax1,title)

    ax2 = fig.add_subplot(gs[1, 0])
    print(df.STDs.value_counts())
    prob_feat_versus_other(feat,df.STDs,['STDs','No STDs'],ax2,title)

    ax3 = fig.add_subplot(gs[1, 1])
    print(df['Hormonal Contraceptives'].value_counts())
    prob_feat_versus_other(feat,df['Hormonal Contraceptives'],['Hormonal','No Hormonal'],ax3,title)

    ax4 = fig.add_subplot(gs[2, 0])
    print(df.Dx.value_counts())
    prob_feat_versus_other(feat,df.Dx,['Dx','No Dx'],ax4,title)

    ax5 = fig.add_subplot(gs[2, 1])
    print(df.Biopsy.value_counts())
    prob_feat_versus_other(feat,df.Biopsy,['Biopsy=1','Biopsy=0'],ax5,title)

```

```

#####
# Age feature #####
prob_feat_versus_six_cat(df.Age,"Age")
hist_feat_versus_six_cat(df.Age,"Age")

#####
# 'Number of sexual partners' feature #####
hist_feat_versus_six_cat(df["Number of sexual partners"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Number of sexual partners')
prob_feat_versus_six_cat(df["Number of sexual partners"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Number of sexual partners')

#####
# 'Num of pregnancies' feature #####
hist_feat_versus_six_cat(df["Num of pregnancies"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Num of pregnancies')
prob_feat_versus_six_cat(df["Num of pregnancies"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Num of pregnancies')

#####
# 'Hormonal Contraceptives (years)' feature #####
hist_feat_versus_six_cat(df["Hormonal Contraceptives (years)"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Hormonal Contraceptives (years)')
prob_feat_versus_six_cat(df["Hormonal Contraceptives (years)"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Hormonal Contraceptives (years)')

#####
# 'IUD (years)' feature #####
hist_feat_versus_six_cat(df["IUD (years)"].apply(pd.to_numeric, errors='coerce').convert_dtypes(),'IUD
(years)')
prob_feat_versus_six_cat(df["IUD (years)"].apply(pd.to_numeric, errors='coerce').convert_dtypes(),'IUD
(years)')

#####
# 'Smokes (packs/year)' feature #####
hist_feat_versus_six_cat(df["Smokes (packs/year)"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Smokes (packs/year)')
prob_feat_versus_six_cat(df["Smokes (packs/year)"].apply(pd.to_numeric,
errors='coerce').convert_dtypes(),'Smokes (packs/year)')

#Extracts input and output variables
X = df.drop('Biopsy', axis =1)
y = df["Biopsy"]

#Feature Importance using RandomForest Classifier
names = X.columns
rf = RandomForestClassifier()
rf.fit(X, y)

result_rf = pd.DataFrame()
result_rf['Features'] = X.columns
result_rf ['Values'] = rf.feature_importances_
result_rf.sort_values('Values', inplace = True, ascending = False)

plt.figure(figsize=(25,25))
sns.set_color_codes("pastel")

```

```

sns.barplot(x = 'Values',y = 'Features', data=result_rf, color="Blue")
plt.xlabel('Feature Importance', fontsize=30)
plt.ylabel('Feature Labels', fontsize=30)
plt.tick_params(axis='x', labelsize=20)
plt.tick_params(axis='y', labelsize=20)
plt.show()

# Print the feature importance table
print("Feature Importance:")
print(result_rf)

#Feature Importance using ExtraTreesClassifier
model = ExtraTreesClassifier()
model.fit(X, y)

result_et = pd.DataFrame()
result_et['Features'] = X.columns
result_et ['Values'] = model.feature_importances_
result_et.sort_values('Values', inplace=True, ascending =False)

plt.figure(figsize=(25,25))
sns.set_color_codes("pastel")
sns.barplot(x = 'Values',y = 'Features', data=result_et, color="red")
plt.xlabel('Feature Importance', fontsize=30)
plt.ylabel('Feature Labels', fontsize=30)
plt.tick_params(axis='x', labelsize=20)
plt.tick_params(axis='y', labelsize=20)
plt.show()

# Print the feature importance table
print("Feature Importance:")
print(result_et)

#Feature Importance using RFE
from sklearn.feature_selection import RFE
model = LogisticRegression()
# create the RFE model
rfe = RFE(model)
rfe = rfe.fit(X, y)

result_lg = pd.DataFrame()
result_lg['Features'] = X.columns
result_lg ['Ranking'] = rfe.ranking_
result_lg.sort_values('Ranking', inplace=True , ascending = False)

plt.figure(figsize=(25,25))
sns.set_color_codes("pastel")
sns.barplot(x = 'Ranking',y = 'Features', data=result_lg, color="orange")
plt.xlabel('Feature Labels', fontsize=30)
plt.ylabel('Feature Labels', fontsize=30)
plt.tick_params(axis='x', labelsize=20)
plt.tick_params(axis='y', labelsize=20)

```

```

plt.show()

print("Feature Ranking:")
print(result_lg)

X = df.drop('Biopsy', axis =1).apply(pd.to_numeric, errors='coerce').astype('float64')
y = df["Biopsy"]
sm = SMOTE(random_state=42)
X,y = sm.fit_resample(X, y.ravel())

#Splits the data into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2021, stratify=y)
X_train_raw = X_train.copy()
X_test_raw = X_test.copy()
y_train_raw = y_train.copy()
y_test_raw = y_test.copy()

X_train_norm = X_train.copy()
X_test_norm = X_test.copy()
y_train_norm = y_train.copy()
y_test_norm = y_test.copy()
norm = MinMaxScaler()
X_train_norm = norm.fit_transform(X_train_norm)
X_test_norm = norm.transform(X_test_norm)

X_train_stand = X_train.copy()
X_test_stand = X_test.copy()
y_train_stand = y_train.copy()
y_test_stand = y_test.copy()
scaler = StandardScaler()
X_train_stand = scaler.fit_transform(X_train_stand)
X_test_stand = scaler.transform(X_test_stand)

def plot_learning_curve(estimator, title, X, y, axes=None, ylim=None, cv=None,
                       n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):
    if axes is None:
        _, axes = plt.subplots(1, 3, figsize=(20, 5))

    axes[0].set_title(title)
    if ylim is not None:
        axes[0].set_ylim(*ylim)
    axes[0].set_xlabel("Training examples")
    axes[0].set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = \
        learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
                      train_sizes=train_sizes,
                      return_times=True)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)

```

```

test_scores_std = np.std(test_scores, axis=1)
fit_times_mean = np.mean(fit_times, axis=1)
fit_times_std = np.std(fit_times, axis=1)

# Plot learning curve
axes[0].grid()
axes[0].fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r")
axes[0].fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1,
                     color="g")
axes[0].plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
axes[0].plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")
axes[0].legend(loc="best")

# Plot n_samples vs fit_times
axes[1].grid()
axes[1].plot(train_sizes, fit_times_mean, 'o-')
axes[1].fill_between(train_sizes, fit_times_mean - fit_times_std,
                     fit_times_mean + fit_times_std, alpha=0.1)
axes[1].set_xlabel("Training examples")
axes[1].set_ylabel("fit_times")
axes[1].set_title("Scalability of the model")

# Plot fit_time vs score
axes[2].grid()
axes[2].plot(fit_times_mean, test_scores_mean, 'o-')
axes[2].fill_between(fit_times_mean, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1)
axes[2].set_xlabel("fit_times")
axes[2].set_ylabel("Score")
axes[2].set_title("Performance of the model")

return plt

def plot_real_pred_val(Y_test, ypred, name,fc):
    plt.figure(figsize=(25,15))
    acc=accuracy_score(Y_test,ypred)
    plt.scatter(range(len(ypred)),ypred,color="blue",\
               lw=5,label="Predicted")
    plt.scatter(range(len(Y_test)),\
               Y_test,color="red",label="Actual")
    plt.title("Predicted Values vs True Values of " + name+" with " + fc + " scaling", \
              fontsize=35)
    plt.xlabel("Accuracy: " + str(round((acc*100),3)) + "%", fontsize=25)
    plt.legend(fontsize=20)
    plt.grid(True, alpha=0.75, lw=1, ls='-.')
    plt.show()

```

```

def plot_cm(Y_test, ypred, name, fc):
    plt.figure(figsize=(25, 15))
    ax = plt.subplot()
    cm = confusion_matrix(Y_test, ypred)
    sns.heatmap(cm, annot=True, linewidth=3, linecolor='red', fmt='g', cmap="YlOrBr", annot_kws={"size": 25})
    plt.title(name + ' Confusion Matrix' + " with " + fc + " scaling", fontsize=35)
    plt.xlabel('Y predict', fontsize=20)
    plt.ylabel('Y test', fontsize=20)
    ax.xaxis.set_ticklabels(['Biopsy = 0', 'Biopsy = 1'], fontsize=25)
    ax.yaxis.set_ticklabels(['Biopsy', 'Biopsy = 1'], fontsize=25)
    plt.show()
    return cm

#Plots ROC
def plot_roc(model,X_test, y_test, title, fc):
    Y_pred_prob = model.predict_proba(X_test)
    Y_pred_prob = Y_pred_prob[:, 1]

    fpr, tpr, thresholds = roc_curve(y_test, Y_pred_prob)
    plt.figure(figsize=(25,15))
    plt.plot([0,1],[0,1], color='navy', linestyle='--', linewidth=5)
    plt.plot(fpr,tpr, color='red', linewidth=5)
    plt.xlabel('False Positive Rate', fontsize=25)
    plt.ylabel('True Positive Rate', fontsize=25)
    plt.title('ROC Curve of ' + title + " with " + fc + " scaling", fontsize=35)
    plt.grid(True)
    plt.show()

def plot_decision_boundary(model,xtest, ytest, name, fc):
    plt.figure(figsize=(25,15))
    #Trains model with two features
    model.fit(xtest, ytest)

    plot_decision_regions(xtest.values, ytest.ravel(), clf=model, legend=2)
    plt.title("Decision boundary for " + name + " with " + fc + " scaling", fontsize=35)
    plt.xlabel('Number of sexual partners', fontsize=25)
    plt.ylabel('Hormonal Contraceptives', fontsize=25)
    plt.show()

    #Chooses two features for decision boundary
    feat_boundary = ['Number of sexual partners','Hormonal Contraceptives']
    X_feature = X[feat_boundary]
    X_train_feat, X_test_feat, y_train_feat, y_test_feat = train_test_split(X_feature, y, test_size = 0.2,
    random_state = 2021, stratify=y)

def train_model(model, X, y):
    model.fit(X, y)
    return model

```

```

def predict_model(model, X, proba=False):
    if ~proba:
        y_pred = model.predict(X)
    else:
        y_pred_proba = model.predict_proba(X)
        y_pred = np.argmax(y_pred_proba, axis=1)

    return y_pred

list_scores = []

def run_model(name, model, X_train, X_test, y_train, y_test, fc, proba=False):
    print(name)
    print(fc)

    model = train_model(model, X_train, y_train)
    y_pred = predict_model(model, X_test, proba)

    accuracy = accuracy_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    print('accuracy: ', accuracy)
    print('recall: ', recall)
    print('precision: ', precision)
    print('f1: ', f1)
    print(classification_report(y_test, y_pred))

    plot_cm(y_test, y_pred, name, fc)
    plot_real_pred_val(y_test, y_pred, name, fc)
    plot_roc(model, X_test, y_test, name, fc)
    plot_decision_boundary(model,X_test_feat, y_test_feat, name, fc)
    plot_learning_curve(model, name, X_train, y_train, cv=3);
    plt.show()

    list_scores.append({'Model Name': name, 'Feature Scaling':fc, 'Accuracy': accuracy, 'Recall': recall,
'Precision': precision, 'F1':f1})

feature_scaling = {
    #'Raw':(X_train_raw, X_test_raw, y_train_raw, y_test_raw),
    #'Normalization':(X_train_norm, X_test_norm, y_train_norm, y_test_norm),
    #'Standardization':(X_train_stand, X_test_stand, y_train_stand, y_test_stand),
}

#Support Vector Classifier
# Define the parameter grid for the grid search
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'poly', 'rbf'],
    'gamma': ['scale', 'auto', 0.1, 1],
}

```

```

}

# Initialize the SVC model
model_svc = SVC(random_state=2021, probability=True)

# Perform the grid search for each feature scaling method
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

# Create GridSearchCV with the SVC model and the parameter grid
grid_search = GridSearchCV(model_svc, param_grid, cv=3, scoring='accuracy', n_jobs=-1, refit=True)

# Train and perform grid search
grid_search.fit(X_train, y_train)

# Get the best SVC model from the grid search
best_model = grid_search.best_estimator_

# Evaluate and plot the best model
run_model('SVC', model_svc, X_train, X_test, y_train, y_test, fc_name)

# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

#Logistic Regression Classifier
# Define the parameter grid for the grid search
param_grid = {
'C': [0.01, 0.1, 1, 10],
'penalty': ['l1', 'l2'],
'solver': ['newton-cg', 'lbfgs', 'liblinear', 'saga'],
}

# Initialize the Logistic Regression model
logreg = LogisticRegression(max_iter=5000, random_state=2021)

# Perform the grid search for each feature scaling method
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

# Create GridSearchCV with the Logistic Regression model and the parameter grid
grid_search = GridSearchCV(logreg, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

# Train and perform grid search
grid_search.fit(X_train, y_train)

# Get the best Logistic Regression model from the grid search
best_model = grid_search.best_estimator_

# Evaluate and plot the best model (setting proba=True for probability prediction)
run_model('Logistic Regression', best_model, X_train, X_test, y_train, y_test, fc_name, proba=True)

```

```

# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

#KNN Classifier
# Define the parameter grid for the grid search
param_grid = {
    'n_neighbors': list(range(2, 10))
}

# KNN Classifier Grid Search
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

    # Initialize the KNN Classifier
    knn = KNeighborsClassifier()

    # Create GridSearchCV with the KNN model and the parameter grid
    grid_search = GridSearchCV(knn, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

    # Train and perform grid search
    grid_search.fit(X_train, y_train)

    # Get the best KNN model from the grid search
    best_model = grid_search.best_estimator_

    # Evaluate and plot the best model (setting proba=True for probability prediction)
    run_model(f'KNeighbors Classifier n_neighbors = {grid_search.best_params_["n_neighbors"]}', 
              best_model, X_train, X_test, y_train, y_test, fc_name, proba=True)

# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

#Decision Trees Classifier
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

    # Initialize the DecisionTreeClassifier model
    dt_clf = DecisionTreeClassifier(random_state=2021)

    # Define the parameter grid for the grid search
    param_grid = {
        'max_depth': np.arange(1, 51, 1),
        'criterion': ['gini', 'entropy'],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4],
    }

    # Create GridSearchCV with the DecisionTreeClassifier model and the parameter grid

```

```

grid_search = GridSearchCV(dt_clf, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

# Train and perform grid search
grid_search.fit(X_train, y_train)

# Get the best DecisionTreeClassifier model from the grid search
best_model = grid_search.best_estimator_

# Evaluate and plot the best model (setting proba=True for probability prediction)
run_model(f'DecisionTree Classifier (Best Depth: {grid_search.best_params_["max_depth"]})',
           best_model, X_train, X_test, y_train, y_test, fc_name, proba=True)

# Print the best hyperparameters found
print(f'Best Hyperparameters for {fc_name}:')
print(grid_search.best_params_)

#Random Forest Classifier
# Define the parameter grid for the grid search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize the RandomForestClassifier model
rf = RandomForestClassifier(random_state=2021)

# RandomForestClassifier Grid Search
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

    # Create GridSearchCV with the RandomForestClassifier model and the parameter grid
    grid_search = GridSearchCV(rf, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

    # Train and perform grid search
    grid_search.fit(X_train, y_train)

    # Get the best RandomForestClassifier model from the grid search
    best_model = grid_search.best_estimator_

    # Evaluate and plot the best model (setting proba=True for probability prediction)
    run_model(f'RandomForest Classifier (Best Estimators: {grid_search.best_params_["n_estimators"]})',
              best_model, X_train, X_test, y_train, y_test, fc_name, proba=True)

    # Print the best hyperparameters found
    print(f'Best Hyperparameters for {fc_name}:')
    print(grid_search.best_params_)

#Gradient Boosting Classifier
# Initialize the GradientBoostingClassifier model

```

```

gbt = GradientBoostingClassifier(random_state=2021)

# Define the parameter grid for the grid search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30],
    'subsample': [0.6, 0.8, 1.0],
    'max_features': [0.2, 0.4, 0.6, 0.8, 1.0],
}

# GradientBoosting Classifier Grid Search
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

# Create GridSearchCV with the GradientBoostingClassifier model and the parameter grid
grid_search = GridSearchCV(gbt, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

# Train and perform grid search
grid_search.fit(X_train, y_train)

# Get the best GradientBoostingClassifier model from the grid search
best_model = grid_search.best_estimator_

# Evaluate and plot the best model (setting proba=True for probability prediction)
run_model(f'GradientBoosting Classifier (Best Estimators: {grid_search.best_params_["n_estimators"]})',
          best_model, X_train, X_test, y_train, y_test, fc_name, proba=True)

# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

#Extreme Gradient Boosting Classifier
# XGBoost Classifier Grid Search
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

# Define the parameter grid for the grid search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0],
}

# Initialize the XGBoost classifier
xgb = XGBClassifier(random_state=2021, use_label_encoder=False, eval_metric='mlogloss')

# Create GridSearchCV with the XGBoost classifier and the parameter grid
grid_search = GridSearchCV(xgb, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

```

```

# Train and perform grid search
grid_search.fit(X_train, y_train)

# Get the best XGBoost classifier model from the grid search
best_model = grid_search.best_estimator_

# Evaluate and plot the best model (setting proba=True for probability prediction)
run_model(f'XGB Classifier (Best Estimators: {grid_search.best_params_["n_estimators"]})',
          best_model, X_train, X_test, y_train, y_test, fc_name, proba=True)

# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

# MLP Classifier Grid Search
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

    # Define the parameter grid for the grid search
    param_grid = {
        'hidden_layer_sizes': [(50,), (100,), (50, 50), (100, 50), (100, 100)],
        'activation': ['logistic', 'relu'],
        'solver': ['adam', 'sgd'],
        'alpha': [0.0001, 0.001, 0.01],
        'learning_rate': ['constant', 'invscaling', 'adaptive'],
    }

    # Initialize the MLP Classifier
    mlp = MLPClassifier(random_state=2021)

    # Create GridSearchCV with the MLP Classifier and the parameter grid
    grid_search = GridSearchCV(mlp, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

    # Train and perform grid search
    grid_search.fit(X_train, y_train)

    # Get the best MLP Classifier model from the grid search
    best_model = grid_search.best_estimator_

    # Evaluate and plot the best model (setting proba=True for probability prediction)
    run_model('MLP Classifier', best_model, X_train, X_test, y_train, y_test, fc_name, proba=True)

    # Print the best hyperparameters found
    print(f"Best Hyperparameters for {fc_name}:")
    print(grid_search.best_params_)

#LGBM Classifier
# Define the parameter grid for grid search
param_grid = {
    'max_depth': [10, 20, 30],
}

```

```

'n_estimators': [100, 200, 300],
'subsample': [0.6, 0.8, 1.0],
'random_state': [2021]
}

# Initialize the LightGBM classifier
lgbm = LGBMClassifier()

# Grid Search
for fc_name, value in feature_scaling.items():
    X_train, X_test, y_train, y_test = value

# Create GridSearchCV with the LightGBM classifier and the parameter grid
grid_search = GridSearchCV(lgbm, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

# Train and perform grid search
grid_search.fit(X_train, y_train)

# Get the best LightGBM classifier model from the grid search
best_model = grid_search.best_estimator_

# Evaluate and plot the best model (setting proba=True for probability prediction)
run_model('LGBM Classifier', best_model, X_train, X_test, y_train, y_test, fc_name, proba=True)

# Print the best hyperparameters found
print(f"Best Hyperparameters for {fc_name}:")
print(grid_search.best_params_)

```

USING DEEP LEARNING

Reading Dataset

- Step 1 Download dataset from <https://viviansishaan.blogspot.com/2023/08/data-science-workshop-cervical-cancer.html> and save it to your working directory. Unzip file, *kag_risk_factors_cervical_cancer.csv* and place it into working directory.
- Step 2 Open a new Python script and save it as **cervical_ann.py**.
- Step 3 Import all necessary libraries:

```
1 #cervical_ann.py
2 import numpy as np # linear algebra
3 import pandas as pd # data processing, CSV file
4 I/O
5 import os
6 import cv2
7 import pandas as pd
8 import seaborn as sns
9 sns.set_style('darkgrid')
10 from matplotlib import pyplot as plt
11 from sklearn.preprocessing import LabelEncoder
12 from sklearn.model_selection import
13 train_test_split
14 from sklearn.preprocessing import StandardScaler
15 import tensorflow as tf
16 from sklearn.metrics import confusion_matrix,
classification_report, accuracy_score
```

```
from imblearn.over_sampling import SMOTE
```

Preprocessing

- Step 1 Read stroke dataset, check its null values, and fill the missing values of numeric data columns with mean of the column data:

```
1 #Checks null values
2 df = df.replace('?', np.NaN)
3 print(df.isnull().sum())
4 print('Total number of null values: ',
5 df.isnull().sum().sum())
6
7 #Drops two columns of STDs
8 df.drop(['STDs: Time since first diagnosis',\
9 'STDs: Time since last
diagnosis'],inplace=True,axis=1)
10
11 numerical_df = ['Age', 'Number of sexual partners', \
12 'First sexual intercourse','Num of pregnancies', \
13 'Smokes (years)', 'Smokes (packs/year)', \
14 'Hormonal Contraceptives (years)','IUD (years)', \
15 'STDs (number)']
16
17 categorical_df = ['Smokes','Hormonal
Contraceptives','IUD',\
18 'STDs', 'STDs:condylomatosis','STDs:cervical
condylomatosis',\
19 'STDs:vaginal condylomatosis',\
20 'STDs:vulvo-perineal condylomatosis',\
21 'STDs:syphilis',\
22 'STDs:pelvic inflammatory disease', 'STDs:genital
herpes',\
23 'STDs:molluscum contagiosum', 'STDs:AIDS', \
24 'STDs:HIV','STDs:Hepatitis B', 'STDs:HPV', \
25 'STDs: Number of diagnosis','Dx:Cancer', 'Dx:CIN', \
26 \
27 'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller','Citology',
28 'Biopsy']
```

```
33
34 #Fills the missing values of numeric data columns
35 with mean of the column data.
36 for feature in numerical_df:
```

```

print(feature,",df[feature].apply(pd.to_numeric,\n    errors='coerce').mean())\nfeature_mean =\nround(df[feature].apply(pd.to_numeric,\n    errors='coerce').mean(),1)\ndf[feature] = df[feature].fillna(feature_mean)\n\nfor feature in categorical_df:\n    df[feature] = df[feature].apply(pd.to_numeric,\n        errors='coerce').fillna(1.0)

```

The code performs data preprocessing and cleansing on a DataFrame named df, which presumably contains medical data. The following steps are executed:

1. Null Value Handling:

- The code replaces any '?' values in the DataFrame with NaN (Not a Number).
- The code then prints the sum of null values in each column using df.isnull().sum().
- The total number of null values across the entire DataFrame is printed using df.isnull().sum().sum().

2. Dropping Columns:

Two columns, namely 'STDs: Time since first diagnosis' and 'STDs: Time since last diagnosis', are dropped from the DataFrame using the df.drop() method.

3. Data Columns Classification:

Lists named numerical_df and categorical_df are defined to group column names into numerical and categorical features, respectively.

4. Filling Missing Values (Numerical Features):

- The missing values in numerical data columns are filled with the mean value of the respective column using a loop.
- For each feature in numerical_df, the mean value is calculated by applying pd.to_numeric() to convert the column to numeric data type, handling any conversion errors with errors='coerce'.

- The mean value is rounded to one decimal place and used to fill the missing values in the corresponding column.

5. Filling Missing Values (Categorical Features):

For each feature in categorical_df, the missing values are first filled with a numeric value of 1.0 after applying pd.to_numeric() to the column and handling any conversion errors.

In summary, the code preprocesses the medical data in the df DataFrame by handling missing values, dropping specific columns, and ensuring that both numerical and categorical columns are appropriately treated. This data preparation step helps ensure that the data is suitable for further analysis and modeling.

Resampling and Splitting

Step 1 Split the data into training and testing dan transform them using **StandardScaler** from SKLearn:

```

1 X = df.drop('Biopsy', axis =1).apply(pd.to_numeric, \
2   errors='coerce').astype('float64')
3 y = df["Biopsy"]
4 sm = SMOTE(random_state=42)
5 X,y = sm.fit_resample(X, y.ravel())
6
7 #Splits the data into training and testing
8 X_train, X_test, y_train, y_test = train_test_split(X,
9 y, \
10 test_size = 0.2, random_state = 2021, stratify=y)
11
12 #Standar scaler
13 sc = StandardScaler()
14 X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

```

The code demonstrates the process of preparing the data for a machine learning task using the SMOTE technique for handling class imbalance and applying feature scaling. Here's a breakdown of the steps performed:

1. Data Transformation and Resampling:

- The X DataFrame is created by dropping the 'Biopsy' column from the original DataFrame df. Numeric values are ensured using pd.to_numeric() with errors='coerce', and then the data is explicitly cast to the float64 data type.
- The target variable y is assigned the 'Biopsy' column from the original DataFrame.
- The Synthetic Minority Over-sampling Technique (SMOTE) is applied to the data using SMOTE(random_state=42). This technique generates synthetic samples to balance the class distribution by oversampling the minority class (in this case, the positive 'Biopsy' class).

2. Train-Test Split:

The data is split into training and testing sets using the train_test_split function. The test_size parameter is set to 0.2, indicating that 20% of the data will be used for testing. The stratify parameter is set to y to ensure that the class distribution is preserved in the train-test split.

3. Feature Scaling:

- The StandardScaler is instantiated as sc.
- The training data features (X_train) are standardized using sc.fit_transform(X_train), which scales the features to have zero mean and unit variance.
- The testing data features (X_test) are transformed using sc.transform(X_test) to apply the same scaling transformation as done for the training data.

Overall, this code prepares the data for training a machine learning model. It ensures that the features are standardized and the class distribution is balanced using the SMOTE technique. This is an essential step to improve the performance of a machine learning model, especially when dealing with imbalanced datasets.

Building, Compiling, and Training ANN Model

Step Build, compile, and train model and save it and its history into files:

1

```
1 #Imports Tensorflow and create a Sequential Model
2 to add layer for the ANN
3 ann = tf.keras.models.Sequential()
4
5 #Input layer
6 ann.add(tf.keras.layers.Dense(units=500,
7     input_dim=33,
8     kernel_initializer='uniform',
9     activation='relu'))
10 ann.add(tf.keras.layers.Dropout(0.5))
11
12 #Hidden layer 1
13 ann.add(tf.keras.layers.Dense(units=200,
14     kernel_initializer='uniform',
15     activation='relu'))
16 ann.add(tf.keras.layers.Dropout(0.5))
17
18 #Output layer
19 ann.add(tf.keras.layers.Dense(units=1,
20     kernel_initializer='uniform',
21     activation='sigmoid'))
22
23 print(ann.summary()) #for showing the structure and
24 parameters
25
26 #Compiles the ANN using ADAM optimizer.
27 ann.compile(optimizer = 'adam', loss =
28 'binary_crossentropy', metrics = ['accuracy'])
29
30 #Trains the ANN with 100 epochs.
31 history = ann.fit(X_train, y_train, batch_size = 64,
32 validation_split=0.20, epochs = 250, shuffle=True)
33
34 #Saves model
35 ann.save('cervical_model.h5')
36
#Saves history into npy file
np.save('cervical_history.npy', history.history)
```

This code outlines the process of building, compiling, training, and saving an Artificial Neural Network (ANN) model using TensorFlow. Here's an overview of each step:

1. Model Architecture Definition:

- TensorFlow and the Sequential model are imported.
- The Sequential model (`ann`) is created to stack layers sequentially.
- An input layer is added using `ann.add(tf.keras.layers.Dense(...))`, specifying 500 units, relu activation, and dropout of 0.5. The `input_dim` parameter is set to 33.
- A hidden layer is added similarly with 200 units, relu activation, and dropout.
- An output layer is added with 1 unit and sigmoid activation, appropriate for binary classification.
- The model's summary is printed using `ann.summary()`.

2. Compilation and Optimization:

- The model is compiled using the 'adam' optimizer and binary cross-entropy loss for binary classification.
- The 'accuracy' metric is specified to monitor during training.

3. Training the Model:

- The `fit()` function is called to train the ANN on the training data (`X_train, y_train`).
- `batch_size` is set to 64, and validation data is split from the training data using `validation_split=0.20`.
- The model is trained for 250 epochs, with data shuffling enabled (`shuffle=True`).
- The training history is stored in the `history` variable.

4. Model and History Saving:

- The trained ANN model is saved as an HDF5 file using `ann.save('cervical_model.h5')`.
- The training history is saved into a NumPy .npy file named '`cervical_history.npy`' using `np.save()`.

This code demonstrates the complete process of creating, training, and saving an ANN model for binary classification using TensorFlow. The model architecture is defined, optimized, trained, and the trained model along with training history is saved for future use.

Output:

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 500)	17000

dropout (Dropout)	(None, 500)	0
dense_1 (Dense)	(None, 200)	100200
dropout_1 (Dropout)	(None, 200)	0
dense_2 (Dense)	(None, 1)	201

Total params: 117,401

Trainable params: 117,401

Non-trainable params: 0

None

Epoch 12/250

17/17 [=====] - 0s 10ms/step - loss: 0.0629 - accuracy: 0.9825 - val_loss: 0.2462 - val_accuracy: 0.9533

Epoch 13/250

17/17 [=====] - 0s 10ms/step - loss: 0.0562 - accuracy: 0.9825 - val_loss: 0.2497 - val_accuracy: 0.9611

Epoch 14/250

17/17 [=====] - 0s 10ms/step - loss: 0.0489 - accuracy: 0.9834 - val_loss: 0.2523 - val_accuracy: 0.9611

Epoch 15/250

17/17 [=====] - 0s 9ms/step - loss: 0.0493 - accuracy: 0.9844 - val_loss: 0.2674 - val_accuracy: 0.9533

Epoch 16/250

17/17 [=====] - 0s 9ms/step - loss: 0.0361 - accuracy: 0.9893 - val_loss: 0.2758 - val_accuracy: 0.9533

Epoch 17/250

17/17 [=====] - 0s 9ms/step - loss: 0.0356 - accuracy: 0.9912 - val_loss: 0.2740 - val_accuracy: 0.9494

Epoch 18/250

17/17 [=====] - 0s 9ms/step - loss: 0.0281 - accuracy: 0.9942 - val_loss: 0.2804 - val_accuracy: 0.9494

Epoch 19/250

17/17 [=====] - 0s 9ms/step - loss: 0.0260 - accuracy: 0.9951 - val_loss: 0.2578 - val_accuracy: 0.9611

Epoch 20/250

17/17 [=====] - 0s 5ms/step - loss: 0.0322 - accuracy: 0.9942 - val_loss: 0.2618 - val_accuracy: 0.9572

Epoch 21/250

17/17 [=====] - 0s 4ms/step - loss: 0.0250 - accuracy: 0.9951 - val_loss: 0.2807 - val_accuracy: 0.9494

Epoch 22/250

17/17 [=====] - 0s 3ms/step - loss: 0.0271 - accuracy: 0.9942 - val_loss: 0.2885 - val_accuracy: 0.9572
Epoch 23/250
17/17 [=====] - 0s 6ms/step - loss: 0.0297 - accuracy: 0.9912 - val_loss: 0.2658 - val_accuracy: 0.9572
Epoch 24/250
17/17 [=====] - 0s 6ms/step - loss: 0.0265 - accuracy: 0.9922 - val_loss: 0.3021 - val_accuracy: 0.9494
Epoch 25/250
17/17 [=====] - 0s 5ms/step - loss: 0.0365 - accuracy: 0.9932 - val_loss: 0.2778 - val_accuracy: 0.9533
Epoch 26/250
17/17 [=====] - 0s 5ms/step - loss: 0.0246 - accuracy: 0.9932 - val_loss: 0.2414 - val_accuracy: 0.9611
Epoch 27/250
17/17 [=====] - 0s 6ms/step - loss: 0.0212 - accuracy: 0.9922 - val_loss: 0.2592 - val_accuracy: 0.9572
Epoch 28/250
17/17 [=====] - 0s 6ms/step - loss: 0.0219 - accuracy: 0.9942 - val_loss: 0.2817 - val_accuracy: 0.9533
Epoch 29/250
17/17 [=====] - 0s 6ms/step - loss: 0.0200 - accuracy: 0.9951 - val_loss: 0.2736 - val_accuracy: 0.9572
Epoch 30/250
17/17 [=====] - 0s 5ms/step - loss: 0.0147 - accuracy: 0.9971 - val_loss: 0.2730 - val_accuracy: 0.9572
Epoch 31/250
17/17 [=====] - 0s 5ms/step - loss: 0.0141 - accuracy: 0.9932 - val_loss: 0.2890 - val_accuracy: 0.9572
Epoch 32/250
17/17 [=====] - 0s 6ms/step - loss: 0.0119 - accuracy: 0.9961 - val_loss: 0.2790 - val_accuracy: 0.9572
Epoch 33/250
17/17 [=====] - 0s 6ms/step - loss: 0.0103 - accuracy: 0.9981 - val_loss: 0.2949 - val_accuracy: 0.9572
Epoch 34/250
17/17 [=====] - 0s 6ms/step - loss: 0.0186 - accuracy: 0.9951 - val_loss: 0.2718 - val_accuracy: 0.9611
Epoch 35/250
17/17 [=====] - 0s 6ms/step - loss: 0.0124 - accuracy: 0.9971 - val_loss: 0.2925 - val_accuracy: 0.9611
Epoch 36/250
17/17 [=====] - 0s 6ms/step - loss: 0.0148 - accuracy: 0.9951 - val_loss: 0.2904 - val_accuracy: 0.9572
Epoch 37/250

17/17 [=====] - 0s 6ms/step - loss: 0.0164 - accuracy: 0.9961 - val_loss: 0.3281 - val_accuracy: 0.9533
Epoch 38/250
17/17 [=====] - 0s 6ms/step - loss: 0.0082 - accuracy: 0.9981 - val_loss: 0.3142 - val_accuracy: 0.9533
Epoch 39/250
17/17 [=====] - 0s 5ms/step - loss: 0.0188 - accuracy: 0.9951 - val_loss: 0.2947 - val_accuracy: 0.9611
Epoch 40/250
17/17 [=====] - 0s 6ms/step - loss: 0.0101 - accuracy: 0.9961 - val_loss: 0.3336 - val_accuracy: 0.9494
Epoch 41/250
17/17 [=====] - 0s 6ms/step - loss: 0.0104 - accuracy: 0.9951 - val_loss: 0.3220 - val_accuracy: 0.9533
Epoch 42/250
17/17 [=====] - 0s 5ms/step - loss: 0.0164 - accuracy: 0.9951 - val_loss: 0.3091 - val_accuracy: 0.9611
Epoch 43/250
17/17 [=====] - 0s 6ms/step - loss: 0.0153 - accuracy: 0.9922 - val_loss: 0.3226 - val_accuracy: 0.9611
Epoch 44/250
17/17 [=====] - 0s 6ms/step - loss: 0.0129 - accuracy: 0.9951 - val_loss: 0.3249 - val_accuracy: 0.9572
Epoch 45/250
17/17 [=====] - 0s 6ms/step - loss: 0.0148 - accuracy: 0.9951 - val_loss: 0.3071 - val_accuracy: 0.9611
Epoch 46/250
17/17 [=====] - 0s 6ms/step - loss: 0.0073 - accuracy: 0.9971 - val_loss: 0.3258 - val_accuracy: 0.9572
Epoch 47/250
17/17 [=====] - 0s 6ms/step - loss: 0.0058 - accuracy: 0.9981 - val_loss: 0.3217 - val_accuracy: 0.9611
Epoch 48/250
17/17 [=====] - 0s 5ms/step - loss: 0.0100 - accuracy: 0.9971 - val_loss: 0.3389 - val_accuracy: 0.9572
Epoch 49/250
17/17 [=====] - 0s 6ms/step - loss: 0.0097 - accuracy: 0.9981 - val_loss: 0.3589 - val_accuracy: 0.9611
Epoch 50/250
17/17 [=====] - 0s 5ms/step - loss: 0.0222 - accuracy: 0.9932 - val_loss: 0.3571 - val_accuracy: 0.9533
Epoch 51/250
17/17 [=====] - 0s 6ms/step - loss: 0.0139 - accuracy: 0.9951 - val_loss: 0.3449 - val_accuracy: 0.9611
Epoch 52/250

17/17 [=====] - 0s 6ms/step - loss: 0.0179 - accuracy: 0.9951 - val_loss: 0.3408 - val_accuracy: 0.9611
Epoch 53/250
17/17 [=====] - 0s 6ms/step - loss: 0.0182 - accuracy: 0.9912 - val_loss: 0.3909 - val_accuracy: 0.9494
Epoch 54/250
17/17 [=====] - 0s 7ms/step - loss: 0.0232 - accuracy: 0.9932 - val_loss: 0.3114 - val_accuracy: 0.9611
Epoch 55/250
17/17 [=====] - 0s 6ms/step - loss: 0.0173 - accuracy: 0.9971 - val_loss: 0.3293 - val_accuracy: 0.9611
Epoch 56/250
17/17 [=====] - 0s 5ms/step - loss: 0.0122 - accuracy: 0.9942 - val_loss: 0.3458 - val_accuracy: 0.9611
Epoch 57/250
17/17 [=====] - 0s 5ms/step - loss: 0.0070 - accuracy: 0.9981 - val_loss: 0.3695 - val_accuracy: 0.9572
Epoch 58/250
17/17 [=====] - 0s 5ms/step - loss: 0.0140 - accuracy: 0.9961 - val_loss: 0.3644 - val_accuracy: 0.9611
Epoch 59/250
17/17 [=====] - 0s 5ms/step - loss: 0.0123 - accuracy: 0.9942 - val_loss: 0.3275 - val_accuracy: 0.9611
Epoch 60/250
17/17 [=====] - 0s 5ms/step - loss: 0.0130 - accuracy: 0.9971 - val_loss: 0.3724 - val_accuracy: 0.9611
Epoch 61/250
17/17 [=====] - 0s 7ms/step - loss: 0.0052 - accuracy: 0.9990 - val_loss: 0.4333 - val_accuracy: 0.9611
Epoch 62/250
17/17 [=====] - 0s 6ms/step - loss: 0.0027 - accuracy: 1.0000 - val_loss: 0.4367 - val_accuracy: 0.9611
Epoch 63/250
17/17 [=====] - 0s 6ms/step - loss: 0.0059 - accuracy: 0.9971 - val_loss: 0.4258 - val_accuracy: 0.9611
Epoch 64/250
17/17 [=====] - 0s 6ms/step - loss: 0.0090 - accuracy: 0.9971 - val_loss: 0.3375 - val_accuracy: 0.9611
Epoch 65/250
17/17 [=====] - 0s 6ms/step - loss: 0.0039 - accuracy: 1.0000 - val_loss: 0.3689 - val_accuracy: 0.9611
Epoch 66/250
17/17 [=====] - 0s 6ms/step - loss: 0.0049 - accuracy: 0.9981 - val_loss: 0.3529 - val_accuracy: 0.9611
Epoch 67/250

17/17 [=====] - 0s 6ms/step - loss: 0.0050 - accuracy: 0.9971 - val_loss: 0.3561 - val_accuracy: 0.9611
Epoch 68/250
17/17 [=====] - 0s 6ms/step - loss: 0.0029 - accuracy: 1.0000 - val_loss: 0.3985 - val_accuracy: 0.9611
Epoch 69/250
17/17 [=====] - 0s 6ms/step - loss: 0.0028 - accuracy: 1.0000 - val_loss: 0.4055 - val_accuracy: 0.9611
Epoch 70/250
17/17 [=====] - 0s 6ms/step - loss: 0.0047 - accuracy: 0.9990 - val_loss: 0.4179 - val_accuracy: 0.9611
Epoch 71/250
17/17 [=====] - 0s 6ms/step - loss: 0.0034 - accuracy: 0.9990 - val_loss: 0.4255 - val_accuracy: 0.9611
Epoch 72/250
17/17 [=====] - 0s 6ms/step - loss: 0.0027 - accuracy: 0.9990 - val_loss: 0.3959 - val_accuracy: 0.9650
Epoch 73/250
17/17 [=====] - 0s 5ms/step - loss: 0.0024 - accuracy: 1.0000 - val_loss: 0.4055 - val_accuracy: 0.9650
Epoch 74/250
17/17 [=====] - 0s 5ms/step - loss: 0.0016 - accuracy: 1.0000 - val_loss: 0.4261 - val_accuracy: 0.9611
Epoch 75/250
17/17 [=====] - 0s 6ms/step - loss: 0.0095 - accuracy: 0.9971 - val_loss: 0.4145 - val_accuracy: 0.9650
Epoch 76/250
17/17 [=====] - 0s 5ms/step - loss: 0.0119 - accuracy: 0.9981 - val_loss: 0.3977 - val_accuracy: 0.9650
Epoch 77/250
17/17 [=====] - 0s 7ms/step - loss: 0.0035 - accuracy: 0.9990 - val_loss: 0.4089 - val_accuracy: 0.9611
Epoch 78/250
17/17 [=====] - 0s 7ms/step - loss: 0.0264 - accuracy: 0.9942 - val_loss: 0.4526 - val_accuracy: 0.9572
Epoch 79/250
17/17 [=====] - 0s 6ms/step - loss: 0.0136 - accuracy: 0.9932 - val_loss: 0.3959 - val_accuracy: 0.9650
Epoch 80/250
17/17 [=====] - 0s 5ms/step - loss: 0.0086 - accuracy: 0.9961 - val_loss: 0.4069 - val_accuracy: 0.9611
Epoch 81/250
17/17 [=====] - 0s 6ms/step - loss: 0.0031 - accuracy: 0.9990 - val_loss: 0.4457 - val_accuracy: 0.9572
Epoch 82/250

17/17 [=====] - 0s 6ms/step - loss: 0.0083 - accuracy: 0.9971 - val_loss: 0.4183 - val_accuracy: 0.9611
Epoch 83/250
17/17 [=====] - 0s 6ms/step - loss: 0.0039 - accuracy: 0.9990 - val_loss: 0.4245 - val_accuracy: 0.9650
Epoch 84/250
17/17 [=====] - 0s 5ms/step - loss: 0.0051 - accuracy: 0.9971 - val_loss: 0.3677 - val_accuracy: 0.9689
Epoch 85/250
17/17 [=====] - 0s 7ms/step - loss: 0.0072 - accuracy: 0.9971 - val_loss: 0.4575 - val_accuracy: 0.9611
Epoch 86/250
17/17 [=====] - 0s 7ms/step - loss: 0.0072 - accuracy: 0.9971 - val_loss: 0.4483 - val_accuracy: 0.9611
Epoch 87/250
17/17 [=====] - 0s 6ms/step - loss: 0.0028 - accuracy: 0.9990 - val_loss: 0.4165 - val_accuracy: 0.9689
Epoch 88/250
17/17 [=====] - 0s 6ms/step - loss: 0.0086 - accuracy: 0.9981 - val_loss: 0.4533 - val_accuracy: 0.9611
Epoch 89/250
17/17 [=====] - 0s 5ms/step - loss: 0.0046 - accuracy: 0.9981 - val_loss: 0.4535 - val_accuracy: 0.9572
Epoch 90/250
17/17 [=====] - 0s 6ms/step - loss: 0.0070 - accuracy: 0.9961 - val_loss: 0.4465 - val_accuracy: 0.9611
Epoch 91/250
17/17 [=====] - 0s 5ms/step - loss: 0.0028 - accuracy: 0.9990 - val_loss: 0.4722 - val_accuracy: 0.9572
Epoch 92/250
17/17 [=====] - 0s 5ms/step - loss: 0.0025 - accuracy: 1.0000 - val_loss: 0.4688 - val_accuracy: 0.9611
Epoch 93/250
17/17 [=====] - 0s 6ms/step - loss: 0.0013 - accuracy: 1.0000 - val_loss: 0.5053 - val_accuracy: 0.9572
Epoch 94/250
17/17 [=====] - 0s 5ms/step - loss: 0.0054 - accuracy: 0.9971 - val_loss: 0.4834 - val_accuracy: 0.9572
Epoch 95/250
17/17 [=====] - 0s 5ms/step - loss: 0.0040 - accuracy: 0.9981 - val_loss: 0.4567 - val_accuracy: 0.9572
Epoch 96/250
17/17 [=====] - 0s 5ms/step - loss: 0.0120 - accuracy: 0.9971 - val_loss: 0.4454 - val_accuracy: 0.9650
Epoch 97/250

17/17 [=====] - 0s 6ms/step - loss: 0.0016 - accuracy: 1.0000 - val_loss: 0.4330 - val_accuracy: 0.9611
Epoch 98/250
17/17 [=====] - 0s 6ms/step - loss: 0.0032 - accuracy: 1.0000 - val_loss: 0.3793 - val_accuracy: 0.9689
Epoch 99/250
17/17 [=====] - 0s 6ms/step - loss: 0.0347 - accuracy: 0.9903 - val_loss: 0.5166 - val_accuracy: 0.9611
Epoch 100/250
17/17 [=====] - 0s 6ms/step - loss: 0.0217 - accuracy: 0.9922 - val_loss: 0.4575 - val_accuracy: 0.9611
Epoch 101/250
17/17 [=====] - 0s 6ms/step - loss: 0.0065 - accuracy: 0.9971 - val_loss: 0.3926 - val_accuracy: 0.9650
Epoch 102/250
17/17 [=====] - 0s 6ms/step - loss: 0.0087 - accuracy: 0.9990 - val_loss: 0.3832 - val_accuracy: 0.9689
Epoch 103/250
17/17 [=====] - 0s 6ms/step - loss: 0.0118 - accuracy: 0.9951 - val_loss: 0.4347 - val_accuracy: 0.9572
Epoch 104/250
17/17 [=====] - 0s 6ms/step - loss: 0.0070 - accuracy: 0.9971 - val_loss: 0.4044 - val_accuracy: 0.9611
Epoch 105/250
17/17 [=====] - 0s 6ms/step - loss: 0.0043 - accuracy: 0.9971 - val_loss: 0.4185 - val_accuracy: 0.9611
Epoch 106/250
17/17 [=====] - 0s 6ms/step - loss: 0.0068 - accuracy: 0.9971 - val_loss: 0.3953 - val_accuracy: 0.9650
Epoch 107/250
17/17 [=====] - 0s 6ms/step - loss: 0.0056 - accuracy: 0.9981 - val_loss: 0.4801 - val_accuracy: 0.9650
Epoch 108/250
17/17 [=====] - 0s 6ms/step - loss: 0.0045 - accuracy: 0.9990 - val_loss: 0.5321 - val_accuracy: 0.9611
Epoch 109/250
17/17 [=====] - 0s 6ms/step - loss: 0.0057 - accuracy: 0.9961 - val_loss: 0.5573 - val_accuracy: 0.9572
Epoch 110/250
17/17 [=====] - 0s 6ms/step - loss: 0.0105 - accuracy: 0.9981 - val_loss: 0.5618 - val_accuracy: 0.9572
Epoch 111/250
17/17 [=====] - 0s 5ms/step - loss: 0.0082 - accuracy: 0.9971 - val_loss: 0.5334 - val_accuracy: 0.9572
Epoch 112/250

17/17 [=====] - 0s 5ms/step - loss: 0.0040 - accuracy: 0.9971 - val_loss: 0.5133 - val_accuracy: 0.9572
Epoch 113/250
17/17 [=====] - 0s 6ms/step - loss: 0.0023 - accuracy: 0.9990 - val_loss: 0.5123 - val_accuracy: 0.9611
Epoch 114/250
17/17 [=====] - 0s 6ms/step - loss: 0.0052 - accuracy: 0.9971 - val_loss: 0.5027 - val_accuracy: 0.9650
Epoch 115/250
17/17 [=====] - 0s 5ms/step - loss: 0.0034 - accuracy: 0.9981 - val_loss: 0.5562 - val_accuracy: 0.9650
Epoch 116/250
17/17 [=====] - 0s 4ms/step - loss: 0.0103 - accuracy: 0.9971 - val_loss: 0.4678 - val_accuracy: 0.9689
Epoch 117/250
17/17 [=====] - 0s 5ms/step - loss: 0.0070 - accuracy: 0.9981 - val_loss: 0.4145 - val_accuracy: 0.9689
Epoch 118/250
17/17 [=====] - 0s 6ms/step - loss: 0.0085 - accuracy: 0.9971 - val_loss: 0.5148 - val_accuracy: 0.9650
Epoch 119/250
17/17 [=====] - 0s 6ms/step - loss: 0.0039 - accuracy: 0.9981 - val_loss: 0.5768 - val_accuracy: 0.9650
Epoch 120/250
17/17 [=====] - 0s 6ms/step - loss: 7.1219e-04 - accuracy: 1.0000 - val_loss: 0.5576 - val_accuracy: 0.9650
Epoch 121/250
17/17 [=====] - 0s 6ms/step - loss: 0.0033 - accuracy: 0.9990 - val_loss: 0.5557 - val_accuracy: 0.9650
Epoch 122/250
17/17 [=====] - 0s 6ms/step - loss: 0.0014 - accuracy: 1.0000 - val_loss: 0.5618 - val_accuracy: 0.9650
Epoch 123/250
17/17 [=====] - 0s 6ms/step - loss: 0.0019 - accuracy: 0.9990 - val_loss: 0.5482 - val_accuracy: 0.9689
Epoch 124/250
17/17 [=====] - 0s 6ms/step - loss: 0.0100 - accuracy: 0.9951 - val_loss: 0.5308 - val_accuracy: 0.9689
Epoch 125/250
17/17 [=====] - 0s 6ms/step - loss: 0.0071 - accuracy: 0.9961 - val_loss: 0.4934 - val_accuracy: 0.9689
Epoch 126/250
17/17 [=====] - 0s 5ms/step - loss: 0.0015 - accuracy: 1.0000 - val_loss: 0.4856 - val_accuracy: 0.9689
Epoch 127/250

17/17 [=====] - 0s 7ms/step - loss: 0.0024 - accuracy: 0.9990 - val_loss: 0.5035 - val_accuracy: 0.9650
Epoch 128/250

17/17 [=====] - 0s 5ms/step - loss: 0.0015 - accuracy: 0.9990 - val_loss: 0.5199 - val_accuracy: 0.9611
Epoch 129/250

17/17 [=====] - 0s 6ms/step - loss: 0.0052 - accuracy: 0.9981 - val_loss: 0.4970 - val_accuracy: 0.9650
Epoch 130/250

17/17 [=====] - 0s 5ms/step - loss: 0.0014 - accuracy: 1.0000 - val_loss: 0.5107 - val_accuracy: 0.9611
Epoch 131/250

17/17 [=====] - 0s 6ms/step - loss: 0.0016 - accuracy: 1.0000 - val_loss: 0.5435 - val_accuracy: 0.9572
Epoch 132/250

17/17 [=====] - 0s 6ms/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.5311 - val_accuracy: 0.9650
Epoch 133/250

17/17 [=====] - 0s 6ms/step - loss: 0.0014 - accuracy: 1.0000 - val_loss: 0.5546 - val_accuracy: 0.9572
Epoch 134/250

17/17 [=====] - 0s 6ms/step - loss: 0.0054 - accuracy: 0.9981 - val_loss: 0.5967 - val_accuracy: 0.9494
Epoch 135/250

17/17 [=====] - 0s 5ms/step - loss: 0.0028 - accuracy: 0.9990 - val_loss: 0.5142 - val_accuracy: 0.9611
Epoch 136/250

17/17 [=====] - 0s 6ms/step - loss: 0.0027 - accuracy: 0.9981 - val_loss: 0.5212 - val_accuracy: 0.9650
Epoch 137/250

17/17 [=====] - 0s 6ms/step - loss: 0.0028 - accuracy: 0.9990 - val_loss: 0.5220 - val_accuracy: 0.9689
Epoch 138/250

17/17 [=====] - 0s 6ms/step - loss: 0.0125 - accuracy: 0.9971 - val_loss: 0.5363 - val_accuracy: 0.9611
Epoch 139/250

17/17 [=====] - 0s 6ms/step - loss: 0.0031 - accuracy: 0.9990 - val_loss: 0.4794 - val_accuracy: 0.9689
Epoch 140/250

17/17 [=====] - 0s 5ms/step - loss: 0.0033 - accuracy: 0.9981 - val_loss: 0.5276 - val_accuracy: 0.9650
Epoch 141/250

17/17 [=====] - 0s 6ms/step - loss: 0.0040 - accuracy: 0.9981 - val_loss: 0.5164 - val_accuracy: 0.9650
Epoch 142/250

17/17 [=====] - 0s 6ms/step - loss: 0.0021 - accuracy: 0.9990 - val_loss: 0.5175 - val_accuracy: 0.9650
Epoch 143/250

17/17 [=====] - 0s 6ms/step - loss: 0.0046 - accuracy: 0.9981 - val_loss: 0.5117 - val_accuracy: 0.9650
Epoch 144/250

17/17 [=====] - 0s 5ms/step - loss: 0.0026 - accuracy: 0.9990 - val_loss: 0.5103 - val_accuracy: 0.9650
Epoch 145/250

17/17 [=====] - 0s 5ms/step - loss: 0.0010 - accuracy: 1.0000 - val_loss: 0.5193 - val_accuracy: 0.9650
Epoch 146/250

17/17 [=====] - 0s 6ms/step - loss: 0.0077 - accuracy: 0.9981 - val_loss: 0.5364 - val_accuracy: 0.9650
Epoch 147/250

17/17 [=====] - 0s 5ms/step - loss: 0.0128 - accuracy: 0.9951 - val_loss: 0.4921 - val_accuracy: 0.9650
Epoch 148/250

17/17 [=====] - 0s 6ms/step - loss: 0.0049 - accuracy: 0.9990 - val_loss: 0.4895 - val_accuracy: 0.9650
Epoch 149/250

17/17 [=====] - 0s 6ms/step - loss: 0.0057 - accuracy: 0.9981 - val_loss: 0.5034 - val_accuracy: 0.9650
Epoch 150/250

17/17 [=====] - 0s 7ms/step - loss: 0.0022 - accuracy: 0.9990 - val_loss: 0.5292 - val_accuracy: 0.9572
Epoch 151/250

17/17 [=====] - 0s 7ms/step - loss: 0.0032 - accuracy: 0.9971 - val_loss: 0.4945 - val_accuracy: 0.9650
Epoch 152/250

17/17 [=====] - 0s 6ms/step - loss: 0.0021 - accuracy: 0.9990 - val_loss: 0.5055 - val_accuracy: 0.9650
Epoch 153/250

17/17 [=====] - 0s 6ms/step - loss: 0.0031 - accuracy: 0.9981 - val_loss: 0.4586 - val_accuracy: 0.9689
Epoch 154/250

17/17 [=====] - 0s 5ms/step - loss: 0.0031 - accuracy: 0.9981 - val_loss: 0.4746 - val_accuracy: 0.9572
Epoch 155/250

17/17 [=====] - 0s 6ms/step - loss: 0.0013 - accuracy: 1.0000 - val_loss: 0.5087 - val_accuracy: 0.9650
Epoch 156/250

17/17 [=====] - 0s 6ms/step - loss: 0.0029 - accuracy: 0.9981 - val_loss: 0.4987 - val_accuracy: 0.9650
Epoch 157/250

17/17 [=====] - 0s 6ms/step - loss: 0.0032 - accuracy: 0.9990 - val_loss: 0.5077 - val_accuracy: 0.9689
Epoch 158/250
17/17 [=====] - 0s 6ms/step - loss: 0.0039 - accuracy: 0.9981 - val_loss: 0.5283 - val_accuracy: 0.9650
Epoch 159/250
17/17 [=====] - 0s 6ms/step - loss: 0.0069 - accuracy: 0.9981 - val_loss: 0.4933 - val_accuracy: 0.9650
Epoch 160/250
17/17 [=====] - 0s 6ms/step - loss: 0.0031 - accuracy: 0.9981 - val_loss: 0.4952 - val_accuracy: 0.9650
Epoch 161/250
17/17 [=====] - 0s 6ms/step - loss: 0.0057 - accuracy: 0.9981 - val_loss: 0.5274 - val_accuracy: 0.9611
Epoch 162/250
17/17 [=====] - 0s 6ms/step - loss: 0.0021 - accuracy: 0.9990 - val_loss: 0.5060 - val_accuracy: 0.9650
Epoch 163/250
17/17 [=====] - 0s 6ms/step - loss: 0.0019 - accuracy: 0.9990 - val_loss: 0.5221 - val_accuracy: 0.9650
Epoch 164/250
17/17 [=====] - 0s 6ms/step - loss: 0.0016 - accuracy: 0.9990 - val_loss: 0.5292 - val_accuracy: 0.9650
Epoch 165/250
17/17 [=====] - 0s 6ms/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.5278 - val_accuracy: 0.9650
Epoch 166/250
17/17 [=====] - 0s 6ms/step - loss: 0.0019 - accuracy: 0.9990 - val_loss: 0.5329 - val_accuracy: 0.9650
Epoch 167/250
17/17 [=====] - 0s 7ms/step - loss: 0.0096 - accuracy: 0.9990 - val_loss: 0.5160 - val_accuracy: 0.9650
Epoch 168/250
17/17 [=====] - 0s 6ms/step - loss: 0.0035 - accuracy: 0.9990 - val_loss: 0.5197 - val_accuracy: 0.9650
Epoch 169/250
17/17 [=====] - 0s 4ms/step - loss: 0.0053 - accuracy: 0.9990 - val_loss: 0.4516 - val_accuracy: 0.9689
Epoch 170/250
17/17 [=====] - 0s 5ms/step - loss: 0.0021 - accuracy: 1.0000 - val_loss: 0.4802 - val_accuracy: 0.9689
Epoch 171/250
17/17 [=====] - 0s 5ms/step - loss: 0.0025 - accuracy: 0.9990 - val_loss: 0.5039 - val_accuracy: 0.9650
Epoch 172/250

17/17 [=====] - 0s 6ms/step - loss: 0.0069 - accuracy: 0.9981 - val_loss: 0.5028 - val_accuracy: 0.9611
Epoch 173/250
17/17 [=====] - 0s 6ms/step - loss: 0.0045 - accuracy: 0.9990 - val_loss: 0.4795 - val_accuracy: 0.9611
Epoch 174/250
17/17 [=====] - 0s 6ms/step - loss: 0.0023 - accuracy: 0.9990 - val_loss: 0.4820 - val_accuracy: 0.9611
Epoch 175/250
17/17 [=====] - 0s 6ms/step - loss: 0.0022 - accuracy: 0.9990 - val_loss: 0.4849 - val_accuracy: 0.9611
Epoch 176/250
17/17 [=====] - 0s 7ms/step - loss: 0.0024 - accuracy: 0.9981 - val_loss: 0.4890 - val_accuracy: 0.9611
Epoch 177/250
17/17 [=====] - 0s 6ms/step - loss: 0.0016 - accuracy: 0.9990 - val_loss: 0.4880 - val_accuracy: 0.9611
Epoch 178/250
17/17 [=====] - 0s 6ms/step - loss: 0.0016 - accuracy: 0.9990 - val_loss: 0.4899 - val_accuracy: 0.9611
Epoch 179/250
17/17 [=====] - 0s 5ms/step - loss: 0.0015 - accuracy: 0.9990 - val_loss: 0.5433 - val_accuracy: 0.9611
Epoch 180/250
17/17 [=====] - 0s 7ms/step - loss: 4.4802e-04 - accuracy: 1.0000 - val_loss: 0.5720 - val_accuracy: 0.9611
Epoch 181/250
17/17 [=====] - 0s 5ms/step - loss: 0.0010 - accuracy: 1.0000 - val_loss: 0.5640 - val_accuracy: 0.9611
Epoch 182/250
17/17 [=====] - 0s 6ms/step - loss: 0.0026 - accuracy: 0.9990 - val_loss: 0.5951 - val_accuracy: 0.9611
Epoch 183/250
17/17 [=====] - 0s 6ms/step - loss: 0.0010 - accuracy: 1.0000 - val_loss: 0.5689 - val_accuracy: 0.9611
Epoch 184/250
17/17 [=====] - 0s 6ms/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.5536 - val_accuracy: 0.9611
Epoch 185/250
17/17 [=====] - 0s 6ms/step - loss: 0.0012 - accuracy: 0.9990 - val_loss: 0.5522 - val_accuracy: 0.9611
Epoch 186/250
17/17 [=====] - 0s 5ms/step - loss: 0.0011 - accuracy: 0.9990 - val_loss: 0.5539 - val_accuracy: 0.9611
Epoch 187/250

```
17/17 [=====] - 0s 6ms/step - loss: 1.8859e-04 - accuracy: 1.0000 - val_loss: 0.5569 - val_accuracy: 0.9611
Epoch 188/250
17/17 [=====] - 0s 6ms/step - loss: 0.0018 - accuracy: 0.9990 - val_loss: 0.5619 - val_accuracy: 0.9611
Epoch 189/250
17/17 [=====] - 0s 6ms/step - loss: 0.0015 - accuracy: 1.0000 - val_loss: 0.5441 - val_accuracy: 0.9650
Epoch 190/250
17/17 [=====] - 0s 6ms/step - loss: 8.6467e-04 - accuracy: 1.0000 - val_loss: 0.5504 - val_accuracy: 0.9650
Epoch 191/250
17/17 [=====] - 0s 6ms/step - loss: 2.1028e-04 - accuracy: 1.0000 - val_loss: 0.5689 - val_accuracy: 0.9611
Epoch 192/250
17/17 [=====] - 0s 6ms/step - loss: 0.0013 - accuracy: 0.9990 - val_loss: 0.5311 - val_accuracy: 0.9650
Epoch 193/250
17/17 [=====] - 0s 6ms/step - loss: 3.6715e-04 - accuracy: 1.0000 - val_loss: 0.5122 - val_accuracy: 0.9689
Epoch 194/250
17/17 [=====] - 0s 6ms/step - loss: 6.2790e-04 - accuracy: 1.0000 - val_loss: 0.5420 - val_accuracy: 0.9650
Epoch 195/250
17/17 [=====] - 0s 6ms/step - loss: 3.8848e-04 - accuracy: 1.0000 - val_loss: 0.5497 - val_accuracy: 0.9650
Epoch 196/250
17/17 [=====] - 0s 6ms/step - loss: 2.7500e-04 - accuracy: 1.0000 - val_loss: 0.5476 - val_accuracy: 0.9650
Epoch 197/250
17/17 [=====] - 0s 6ms/step - loss: 5.5528e-04 - accuracy: 1.0000 - val_loss: 0.5502 - val_accuracy: 0.9650
Epoch 198/250
17/17 [=====] - 0s 5ms/step - loss: 0.0032 - accuracy: 0.9990 - val_loss: 0.7018 - val_accuracy: 0.9650
Epoch 199/250
17/17 [=====] - 0s 5ms/step - loss: 0.0081 - accuracy: 0.9981 - val_loss: 0.6617 - val_accuracy: 0.9689
Epoch 200/250
17/17 [=====] - 0s 5ms/step - loss: 0.0036 - accuracy: 0.9981 - val_loss: 0.5875 - val_accuracy: 0.9689
Epoch 201/250
17/17 [=====] - 0s 5ms/step - loss: 0.0020 - accuracy: 0.9990 - val_loss: 0.6954 - val_accuracy: 0.9533
Epoch 202/250
```

17/17 [=====] - 0s 6ms/step - loss: 0.0028 - accuracy: 0.9990 - val_loss: 0.6171 - val_accuracy: 0.9572
Epoch 203/250

17/17 [=====] - 0s 5ms/step - loss: 0.0046 - accuracy: 0.9981 - val_loss: 0.6629 - val_accuracy: 0.9572
Epoch 204/250

17/17 [=====] - 0s 5ms/step - loss: 0.0017 - accuracy: 0.9990 - val_loss: 0.6259 - val_accuracy: 0.9572
Epoch 205/250

17/17 [=====] - 0s 6ms/step - loss: 0.0010 - accuracy: 1.0000 - val_loss: 0.5568 - val_accuracy: 0.9611
Epoch 206/250

17/17 [=====] - 0s 6ms/step - loss: 4.4824e-04 - accuracy: 1.0000 - val_loss: 0.5581 - val_accuracy: 0.9611
Epoch 207/250

17/17 [=====] - 0s 6ms/step - loss: 0.0012 - accuracy: 1.0000 - val_loss: 0.6319 - val_accuracy: 0.9572
Epoch 208/250

17/17 [=====] - 0s 6ms/step - loss: 0.0012 - accuracy: 0.9990 - val_loss: 0.6705 - val_accuracy: 0.9533
Epoch 209/250

17/17 [=====] - 0s 6ms/step - loss: 0.0011 - accuracy: 0.9990 - val_loss: 0.6800 - val_accuracy: 0.9572
Epoch 210/250

17/17 [=====] - 0s 5ms/step - loss: 7.1603e-04 - accuracy: 1.0000 - val_loss: 0.6825 - val_accuracy: 0.9572
Epoch 211/250

17/17 [=====] - 0s 5ms/step - loss: 4.0668e-04 - accuracy: 1.0000 - val_loss: 0.6714 - val_accuracy: 0.9572
Epoch 212/250

17/17 [=====] - 0s 6ms/step - loss: 0.0040 - accuracy: 0.9990 - val_loss: 0.6308 - val_accuracy: 0.9572
Epoch 213/250

17/17 [=====] - 0s 6ms/step - loss: 0.0022 - accuracy: 0.9990 - val_loss: 0.6621 - val_accuracy: 0.9572
Epoch 214/250

17/17 [=====] - 0s 6ms/step - loss: 0.0028 - accuracy: 0.9981 - val_loss: 0.6639 - val_accuracy: 0.9572
Epoch 215/250

17/17 [=====] - 0s 6ms/step - loss: 0.0050 - accuracy: 0.9981 - val_loss: 0.6609 - val_accuracy: 0.9572
Epoch 216/250

17/17 [=====] - 0s 5ms/step - loss: 0.0013 - accuracy: 0.9990 - val_loss: 0.5936 - val_accuracy: 0.9689
Epoch 217/250

17/17 [=====] - 0s 5ms/step - loss: 0.0077 - accuracy: 0.9990 - val_loss: 0.5642 - val_accuracy: 0.9689
Epoch 218/250
17/17 [=====] - 0s 6ms/step - loss: 0.0028 - accuracy: 0.9981 - val_loss: 0.5763 - val_accuracy: 0.9611
Epoch 219/250
17/17 [=====] - 0s 7ms/step - loss: 0.0048 - accuracy: 0.9981 - val_loss: 0.6355 - val_accuracy: 0.9572
Epoch 220/250
17/17 [=====] - 0s 6ms/step - loss: 0.0215 - accuracy: 0.9981 - val_loss: 0.6800 - val_accuracy: 0.9572
Epoch 221/250
17/17 [=====] - 0s 6ms/step - loss: 0.0046 - accuracy: 0.9990 - val_loss: 0.6543 - val_accuracy: 0.9611
Epoch 222/250
17/17 [=====] - 0s 5ms/step - loss: 0.0073 - accuracy: 0.9981 - val_loss: 0.6273 - val_accuracy: 0.9611
Epoch 223/250
17/17 [=====] - 0s 5ms/step - loss: 0.0033 - accuracy: 0.9990 - val_loss: 0.6263 - val_accuracy: 0.9611
Epoch 224/250
17/17 [=====] - 0s 4ms/step - loss: 0.0036 - accuracy: 0.9981 - val_loss: 0.6177 - val_accuracy: 0.9611
Epoch 225/250
17/17 [=====] - 0s 4ms/step - loss: 0.0030 - accuracy: 0.9990 - val_loss: 0.6232 - val_accuracy: 0.9611
Epoch 226/250
17/17 [=====] - 0s 4ms/step - loss: 0.0013 - accuracy: 1.0000 - val_loss: 0.6308 - val_accuracy: 0.9611
Epoch 227/250
17/17 [=====] - 0s 5ms/step - loss: 0.0053 - accuracy: 0.9990 - val_loss: 0.5839 - val_accuracy: 0.9650
Epoch 228/250
17/17 [=====] - 0s 5ms/step - loss: 0.0094 - accuracy: 0.9981 - val_loss: 0.5740 - val_accuracy: 0.9689
Epoch 229/250
17/17 [=====] - 0s 5ms/step - loss: 0.0052 - accuracy: 0.9981 - val_loss: 0.6337 - val_accuracy: 0.9611
Epoch 230/250
17/17 [=====] - 0s 4ms/step - loss: 0.0039 - accuracy: 0.9990 - val_loss: 0.6816 - val_accuracy: 0.9572
Epoch 231/250
17/17 [=====] - 0s 6ms/step - loss: 0.0051 - accuracy: 0.9981 - val_loss: 0.6585 - val_accuracy: 0.9572
Epoch 232/250

17/17 [=====] - 0s 6ms/step - loss: 0.0061 - accuracy: 0.9981 - val_loss: 0.6182 - val_accuracy: 0.9611
Epoch 233/250
17/17 [=====] - 0s 6ms/step - loss: 0.0021 - accuracy: 0.9990 - val_loss: 0.6239 - val_accuracy: 0.9611
Epoch 234/250
17/17 [=====] - 0s 6ms/step - loss: 0.0021 - accuracy: 0.9990 - val_loss: 0.6150 - val_accuracy: 0.9611
Epoch 235/250
17/17 [=====] - 0s 6ms/step - loss: 0.0014 - accuracy: 1.0000 - val_loss: 0.5667 - val_accuracy: 0.9611
Epoch 236/250
17/17 [=====] - 0s 6ms/step - loss: 0.0041 - accuracy: 0.9990 - val_loss: 0.5869 - val_accuracy: 0.9611
Epoch 237/250
17/17 [=====] - 0s 6ms/step - loss: 6.3611e-04 - accuracy: 1.0000 - val_loss: 0.5847 - val_accuracy: 0.9650
Epoch 238/250
17/17 [=====] - 0s 5ms/step - loss: 5.3910e-04 - accuracy: 1.0000 - val_loss: 0.5876 - val_accuracy: 0.9650
Epoch 239/250
17/17 [=====] - 0s 6ms/step - loss: 5.6139e-04 - accuracy: 1.0000 - val_loss: 0.5945 - val_accuracy: 0.9650
Epoch 240/250
17/17 [=====] - 0s 6ms/step - loss: 0.0023 - accuracy: 0.9990 - val_loss: 0.6178 - val_accuracy: 0.9611
Epoch 241/250
17/17 [=====] - 0s 6ms/step - loss: 7.3668e-04 - accuracy: 1.0000 - val_loss: 0.6328 - val_accuracy: 0.9572
Epoch 242/250
17/17 [=====] - 0s 6ms/step - loss: 0.0010 - accuracy: 1.0000 - val_loss: 0.6597 - val_accuracy: 0.9572
Epoch 243/250
17/17 [=====] - 0s 6ms/step - loss: 9.8472e-04 - accuracy: 1.0000 - val_loss: 0.6663 - val_accuracy: 0.9572
Epoch 244/250
17/17 [=====] - 0s 5ms/step - loss: 0.0013 - accuracy: 0.9990 - val_loss: 0.6608 - val_accuracy: 0.9611
Epoch 245/250
17/17 [=====] - 0s 6ms/step - loss: 0.0011 - accuracy: 0.9990 - val_loss: 0.6491 - val_accuracy: 0.9611
Epoch 246/250
17/17 [=====] - 0s 6ms/step - loss: 4.1839e-04 - accuracy: 1.0000 - val_loss: 0.6265 - val_accuracy: 0.9650
Epoch 247/250

```

17/17 [=====] - 0s 5ms/step - loss: 4.2240e-04 - accuracy: 1.0000 - val_loss: 0.6315 - val_accuracy: 0.9650
Epoch 248/250
17/17 [=====] - 0s 5ms/step - loss: 0.0031 - accuracy: 0.9981 - val_loss: 0.6438 - val_accuracy: 0.9611
Epoch 249/250
17/17 [=====] - 0s 5ms/step - loss: 0.0011 - accuracy: 0.9990 - val_loss: 0.6612 - val_accuracy: 0.9611
Epoch 250/250
17/17 [=====] - 0s 6ms/step - loss: 0.0055 - accuracy: 0.9981 - val_loss: 0.6611 - val_accuracy: 0.9611
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])

```

The output provides a summary of the architecture of the created Sequential model. Here's an interpretation of the output:

- The model consists of three layers: an input layer, two hidden layers, and an output layer.
- The input layer (dense) has 500 units/neurons. It takes an input with shape (None, 33), where 33 is the number of input features.
- A dropout layer (dropout) follows with a rate of 0.5. This helps prevent overfitting by randomly deactivating 50% of the units during each epoch.
- The first hidden layer (dense_1) has 200 units/neurons.
- Another dropout layer (dropout_1) follows with the same 0.5 rate.
- The output layer (dense_2) has 1 unit/neuron, suitable for binary classification (sigmoid activation function).
- The summary displays the total number of trainable parameters (117,401) in the model, which are learned during training.

This output provides insights into the structure of the ANN model, including the number of layers, units in each layer, and the total number of parameters. It helps you understand the architecture and the potential complexity of the model.

Accuracy and Loss

Step 1 Plot accuracy and loss versus epoch:

```

1 #Plots accuracy and loss
2 acc = history.history['accuracy']
3 val_acc = history.history['val_accuracy']
4 loss = history.history['loss']
5 val_loss = history.history['val_loss']
6 epochs = range(1, len(acc) + 1)

```

```

7
8 #accuracy
9 fig, ax = plt.subplots(figsize=(25, 15))
10 plt.plot(epochs, acc, 'r', label='Training accuracy',
11 lw=10)
12 plt.plot(epochs, val_acc, 'b--', label='Validation
13 accuracy', lw=10)
14 plt.title('Training and validation accuracy',
15 fontsize=35)
16 plt.legend(fontsize=25)
17 ax.set_xlabel("Epoch", fontsize=30)
18 ax.tick_params(labelsize=30)
19 plt.show()
20
21 #loss
22 fig, ax = plt.subplots(figsize=(25, 15))
23 plt.plot(epochs, loss, 'r', label="Training loss", lw=10)
24 plt.plot(epochs, val_loss, 'b--', label='Validation loss',
25 lw=10)
26 plt.title('Training and validation loss', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

```

The results are shown in Figure 113 and 114.

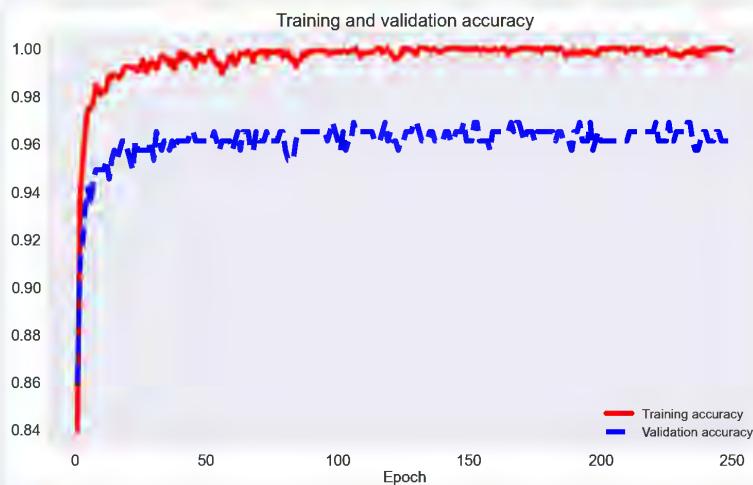


Figure 113 Training and validation accuracy

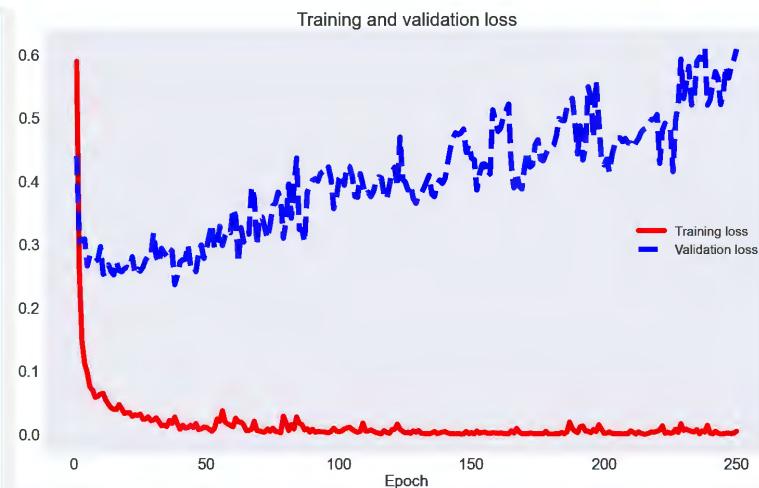


Figure 114 Training and validation loss

The code is responsible for visualizing the training and validation accuracy as well as the training and validation loss of a neural network model across multiple epochs.

The script extracts the accuracy and loss values from the history object, which contains the recorded metrics during training. It prepares the x-axis values (epochs) for plotting based on the number of training epochs.

Then it plots and visualizes the training and validation accuracy. The plt.subplots() function creates a figure and axes for the plot. The accuracy values for both training and validation sets are plotted with red lines for training accuracy and blue dashed lines for validation accuracy. The plt.title, plt.legend, and ax.set_xlabel functions set labels and titles for the plot, while ax.tick_params adjusts the tick label sizes. This plot provides insights into how well the model is learning and generalizing based on accuracy.

Similarly, the third paragraph visualizes the training and validation loss. It follows a similar structure to the accuracy plot, but this time the loss values are plotted. The loss measures how well the model's predictions match the true values and how well it's learning from the data.

Both sets of visualizations serve as critical tools to assess the performance and behavior of the neural network model during training. Patterns in accuracy and loss can help identify potential overfitting, underfitting, or convergence issues, assisting in making informed decisions about model adjustments.

Predicting Result Using Test Data

Step Predict result using test data:

1

```
1 #Sets the threshold for the predictions.  
2 #In this case, the threshold is 0.5 (this value can be  
3 modified).  
4 #prediction on test set  
5 y_pred = ann.predict(X_test)  
6 y_pred = [int(p>=0.5) for p in y_pred]  
print(y_pred)
```

Output:

```
[0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1,  
0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1,  
1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0,  
1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0,  
1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0,  
1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1,  
1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1,  
1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0,  
0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0,  
1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1,  
0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1,  
1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1,  
1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0,  
0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0,  
1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0,  
0, 0, 1, 0, 1, 0, 0, 0, 0]
```

In the code, a threshold of 0.5 is set for making predictions using the trained neural network model. The script performs predictions on the test set and adjusts the predictions based on whether the predicted values are greater than or equal to the specified threshold.

The `ann.predict(X_test)` line calculates the predicted values for the given test data `X_test` using the trained neural network model `ann`.

The subsequent line, `y_pred = [int(p>=0.5) for p in y_pred]`, iterates through the predicted values (`y_pred`) and transforms each prediction into an integer. If a predicted value is greater than or equal to 0.5, it is converted to 1; otherwise, it is

converted to 0. This process effectively classifies the predictions into binary classes based on the chosen threshold.

Finally, the script prints the modified predictions `y_pred`, which now represent the binary classification outcomes for the test set based on the specified threshold. This threshold adjustment is a common technique to convert continuous prediction probabilities into discrete class labels, allowing for easier interpretation and comparison with actual target labels.

Classification Report

Step Print accuracy and classification report:

1

```
1 #Performance Evaluation - Accuracy and
2 Classification Report
3 #Accuracy Score
4 print ('Accuracy Score :', accuracy_score(y_pred,
5 y_test, \
6 normalize=True), '\n')
7
8 #precision, recall report
9 print ('Classification Report :\n\n', \
classification_report(y_pred, y_test))
```

Output:

Accuracy Score : 0.9751552795031055

Classification Report :

	precision	recall	f1-score	support
0	0.95	1.00	0.97	153
1	1.00	0.95	0.98	169
accuracy			0.98	322
macro avg	0.98	0.98	0.98	322
weighted avg	0.98	0.98	0.98	322

In the code, the performance of the classification model is evaluated using accuracy and a classification report.

The `accuracy_score()` function from the `sklearn.metrics` module is used to calculate the accuracy of the model's predictions. It takes the predicted values (`y_pred`) and the actual target values (`y_test`) as arguments. The

`normalize=True` parameter ensures that the accuracy score is normalized, giving the fraction of correctly predicted instances out of the total number of instances in the test set. The printed output displays the accuracy score, indicating the overall correctness of the model's predictions.

Additionally, the `classification_report` function from the same module is used to generate a comprehensive classification report. This report provides various metrics for each class, including precision, recall, F1-score, and support. It is computed based on the comparison between the predicted values (`y_pred`) and the actual target values (`y_test`). The classification report offers insights into the model's performance for different classes and can be useful in assessing its strengths and weaknesses in classifying instances.

These evaluation steps help in understanding the model's effectiveness in making accurate predictions and provide detailed information about its performance on each class. This information can be crucial for determining the model's suitability for the specific classification task and for making informed decisions about model improvements or adjustments.

The output displays the results of evaluating the performance of a classification model using the provided code:

1. Accuracy Score: The accuracy score is calculated to be approximately 0.975, which indicates that around 97.5% of the instances in the test set were correctly classified by the model.
2. Classification Report: This report provides detailed metrics for each class (0 and 1), as well as overall metrics. Here's an explanation of the key metrics:
 - Precision: Precision measures the proportion of true positive predictions among all instances predicted as positive. For class 0, precision is 0.95, indicating that 95% of the instances predicted as class 0 were actually class 0. For class 1, precision is 1.00,

meaning that all instances predicted as class 1 were indeed class 1.

- Recall: Recall, also known as sensitivity or true positive rate, measures the proportion of actual positive instances that were correctly predicted by the model. For class 0, recall is 1.00, indicating that all actual class 0 instances were correctly identified. For class 1, recall is 0.95, meaning that 95% of the actual class 1 instances were predicted correctly.
- F1-Score: The F1-score is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, considering both false positives and false negatives. Both classes have high F1-scores, indicating balanced performance between precision and recall.
- Support: Support represents the number of actual instances of each class in the test set.
- Accuracy: The accuracy reported in the macro and weighted averages is approximately 0.98, which matches the previously calculated accuracy score. This value indicates the overall correctness of the model across all classes.

Overall, the high accuracy and balanced F1-scores suggest that the model is performing well on the given classification task. The classification report provides valuable insights into the model's performance for each class, allowing for a more thorough understanding of its strengths and weaknesses.

Confusion Matrix

Step Plot confusion matrix:

```

1 #Confusion matrix:
2 conf_mat = confusion_matrix(y_true=y_test, y_pred
3 = y_pred)
4 class_list = ['Biopsy = 0', 'Biopsy = 1']
5 fig, ax = plt.subplots(figsize=(25, 15))
6 ax= plt.subplot()
7 sns.heatmap(conf_mat, annot=True, ax = ax,
8 cmap='YlOrBr', fmt='g')
9 ax.set_xlabel('Predicted labels')
10 ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')
ax.xaxis.set_ticklabels(class_list),
ax.yaxis.set_ticklabels(class_list)

```

The result is shown in Figure 115.

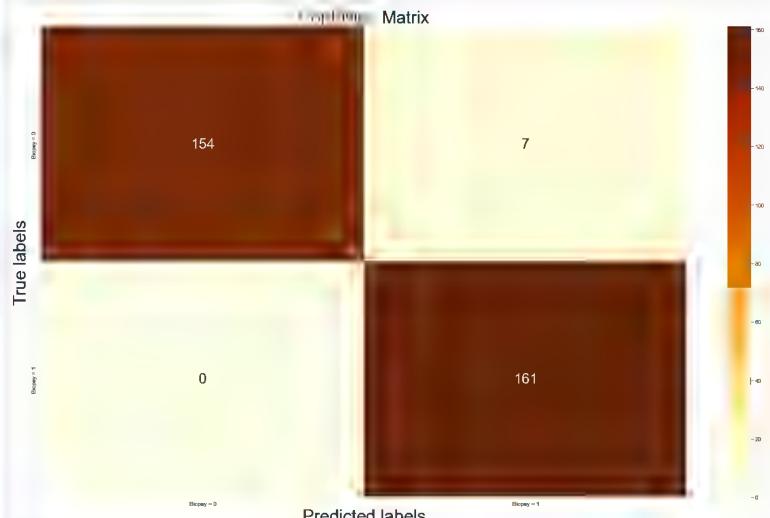


Figure 115 The confusion matrix

The code calculates and visualizes a confusion matrix, which is a powerful tool for understanding the performance of a classification model:

1. Confusion Matrix Calculation: The `confusion_matrix` function is used to calculate the confusion matrix based on the true labels (`y_test`) and the predicted labels (`y_pred`) from the model's predictions. The resulting matrix shows the number of true positive, true negative, false positive, and false negative instances for each class (`Biopsy = 0` and `Biopsy = 1`).

2. Heatmap Visualization: The confusion matrix is visualized using a heatmap. Each cell in the heatmap corresponds to a combination of true and predicted labels and is annotated with the corresponding count. The color scale (cmap) 'YlOrBr' is used to indicate the intensity of the counts. The diagonal cells represent the correct predictions (true positive and true negative), while off-diagonal cells represent errors (false positive and false negative).
3. Axis Labels and Title: The x-axis and y-axis labels are set to indicate the predicted and true labels, respectively. The title of the heatmap is set as "Confusion Matrix."
4. Tick Labels: The tick labels on both axes are set to match the class labels ('Biopsy = 0' and 'Biopsy = 1') for better interpretation.

Overall, the confusion matrix and the heatmap provide a clear visual representation of the model's performance in terms of correct and incorrect predictions for each class. It allows for easy identification of any patterns or areas where the model may be making more errors, helping to guide further analysis and improvement.

True Values versus Predicted Values Diagram

Step 1 Plot true values versus predicted values:

```

1 def plot_real_pred_val(Y_test, ypred, name):
2     plt.figure(figsize=(25,15))
3     acc=accuracy_score(Y_test,ypred)
4
5     plt.scatter(range(len(ypred)),ypred,color="blue",lw=5,\n6         label="Predicted")
7     plt.scatter(range(len(Y_test)),\n8         Y_test,color="red",label="Actual")
9     plt.title("Predicted Values vs True Values of " +
10    name, \
11    fontsize=10)
12

```

```

13     plt.xlabel("Accuracy: " + str(round((acc*100),3)) +
14     "%")
15     plt.legend()
16     plt.grid(True, alpha=0.75, lw=1, ls='-.')
17     plt.show()

plot_real_pred_val(y_test, y_pred, 'ANN')

```

The result is shown in Figure 116.



Figure 116 The true values versus predicted values

The code defines a function `plot_real_pred_val()` to visualize the predicted values versus the true values for a given model's predictions:

1. Function Definition: The function is defined with three parameters: `Y_test` (true values), `ypred` (predicted values), and `name` (model name).
2. Scatter Plot: The function generates a scatter plot with the predicted values (in blue) and the true values (in red). The x-axis represents the index of the data points, while the y-axis represents the values themselves.
3. Title and Accuracy Display: The title of the plot includes the model name and indicates that it displays predicted values versus true values. Additionally, the accuracy of the predictions is displayed in the x-axis label. The accuracy is calculated using the

accuracy_score function and is rounded to three decimal places.

4. Legend and Grid: A legend is included in the plot to differentiate between the predicted and true values. The grid lines are added to improve readability.
5. Visualization: The plt.show() function is used to display the plot.
6. Function Call: The function is called with the parameters y_test (true values), y_pred (predicted values), and the string 'ANN' as the model name.

The resulting scatter plot provides a visual comparison between the model's predicted values and the actual true values, enabling easy identification of any discrepancies or patterns in the predictions. The accuracy percentage is included in the x-axis label for reference.

Following is the full version of **cervical_ann.py**:

```
#cervical_ann.py
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O
import os
import cv2
import pandas as pd
import seaborn as sns
sns.set_style('darkgrid')
from matplotlib import pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from sklearn.metrics import confusion_matrix, classification_report,
accuracy_score
from imblearn.over_sampling import SMOTE

#Reads dataset
curr_path = os.getcwd()
df = pd.read_csv(curr_path+"/kag_risk_factors_cervical_cancer.csv")

#Checks null values
df = df.replace('?', np.NaN)
```

```

print(df.isnull().sum())
print('Total number of null values: ', df.isnull().sum().sum())

#Drops two columns of STDs
df.drop(['STDs: Time since first diagnosis','STDs: Time since last
diagnosis'],inplace=True,axis=1)

numerical_df = ['Age', 'Number of sexual partners', 'First sexual
intercourse','Num of pregnancies', 'Smokes (years)',
'Smokes (packs/year)','Hormonal Contraceptives (years)','IUD
(years)','STDs (number)']

categorical_df = ['Smokes','Hormonal
Contraceptives','IUD','STDs','STDs:condylomatosis','STDs:cervical
condylomatosis',
'STDs:vaginal condylomatosis','STDs:vulvo-perineal condylomatosis',
'STDs:syphilis',
'STDs:pelvic inflammatory disease', 'STDs:genital
herpes','STDs:molluscum contagiosum', 'STDs:AIDS',
'STDs:HIV','STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of
diagnosis','Dx:Cancer', 'Dx:CIN',
'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller','Citology', 'Biopsy']

#Fills the missing values of numeric data columns with mean of the
column data.
for feature in numerical_df:
    print(feature,",",df[feature].apply(pd.to_numeric,
errors='coerce').mean())
    feature_mean = round(df[feature].apply(pd.to_numeric,
errors='coerce').mean(),1)
    df[feature] = df[feature].fillna(feature_mean)

for feature in categorical_df:
    df[feature] = df[feature].apply(pd.to_numeric,
errors='coerce').fillna(1.0)

X = df.drop('Biopsy', axis =1).apply(pd.to_numeric,
errors='coerce').astype('float64')
y = df["Biopsy"]
sm = SMOTE(random_state=42)
X,y = sm.fit_resample(X, y.ravel())

#Splits the data into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 2021, stratify=y)

```

```
#Standar scaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

#Imports Tensorflow and create a Sequential Model to add layer for the
ANN
ann = tf.keras.models.Sequential()

#Input layer
ann.add(tf.keras.layers.Dense(units=500,
    input_dim=33,
    kernel_initializer='uniform',
    activation='relu'))
ann.add(tf.keras.layers.Dropout(0.5))

#Hidden layer 1
ann.add(tf.keras.layers.Dense(units=200,
    kernel_initializer='uniform',
    activation='relu'))
ann.add(tf.keras.layers.Dropout(0.5))

#Output layer
ann.add(tf.keras.layers.Dense(units=1,
    kernel_initializer='uniform',
    activation='sigmoid'))

print(ann.summary()) #for showing the structure and parameters

#Compiles the ANN using ADAM optimizer.
ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
['accuracy'])

#Trains the ANN with 100 epochs.
history = ann.fit(X_train, y_train, batch_size = 64, validation_split=0.20,
epochs = 250, shuffle=True)

#Saves model
ann.save('cervical_model.h5')

#Saves history into npy file
np.save('cervical_history.npy', history.history)

print (history.history.keys())

#Plots accuracy and loss
```

```

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)

#accuracy
fig, ax = plt.subplots(figsize=(25, 15))
plt.plot(epochs, acc, 'r', label='Training accuracy', lw=10)
plt.plot(epochs, val_acc, 'b--', label='Validation accuracy', lw=10)
plt.title('Training and validation accuracy', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

#loss
fig, ax = plt.subplots(figsize=(25, 15))
plt.plot(epochs, loss, 'r', label='Training loss', lw=10)
plt.plot(epochs, val_loss, 'b--', label='Validation loss', lw=10)
plt.title('Training and validation loss', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

#Sets the threshold for the predictions. In this case, the threshold is 0.5
#(this value can be modified).
#prediction on test set
y_pred = ann.predict(X_test)
y_pred = [int(p>=0.5) for p in y_pred]
print(y_pred)

#Performance Evaluation - Accuracy and Classification Report
#Accuracy Score
print ('Accuracy Score : ', accuracy_score(y_pred, y_test,
normalize=True), '\n')

#precision, recall report
print ('Classification Report :\n\n', classification_report(y_pred, y_test))

#Confusion matrix:
conf_mat = confusion_matrix(y_true=y_test, y_pred = y_pred)
class_list = ['Biopsy = 0', 'Biopsy = 1']
fig, ax = plt.subplots(figsize=(25, 15))

```

```

sns.heatmap(conf_mat, annot=True, ax = ax, cmap='YlOrBr', fmt='g',
annot_kws={"size": 25})
ax.set_xlabel('Predicted labels', fontsize=30)
ax.set_ylabel('True labels', fontsize=30)
ax.set_title('Confusion Matrix', fontsize=30)
ax.xaxis.set_ticklabels(class_list), ax.yaxis.set_ticklabels(class_list)

def plot_real_pred_val(Y_test, ypred, name):
    plt.figure(figsize=(25,15))
    acc=accuracy_score(Y_test,ypred)

    plt.scatter(range(len(ypred)),ypred,color="blue",lw=5,label="Predicted")
    plt.scatter(range(len(Y_test)), Y_test,color="red",label="Actual")
    plt.title("Predicted Values vs True Values of " + name, fontsize=10)
    plt.xlabel("Accuracy: " + str(round((acc*100),3)) + "%")
    plt.legend(fontsize=25)
    plt.grid(True, alpha=0.75, lw=1, ls='-.')
    plt.show()

plot_real_pred_val(y_test, y_pred, 'ANN')

```

Cervical Cancer Using LSTM Model

Step 1 Open a new Python script and save it as **cervical_lstm.py**.

2

Step 2 Read dataset and preprocess input:

```

1 #cervical_lstm
2 import numpy as np
3 import pandas as pd
4 import os
5 import seaborn as sns
6 sns.set_style('darkgrid')
7 from matplotlib import pyplot as plt
8 from sklearn.model_selection import train_test_split
9 from sklearn.preprocessing import StandardScaler
10 import tensorflow as tf
11 from sklearn.metrics import confusion_matrix,
12 classification_report, accuracy_score
13 from imblearn.over_sampling import SMOTE
14
15 # Reads dataset
16 curr_path = os.getcwd()
17 df = pd.read_csv(curr_path +
18 "/kag_risk_factors_cervical_cancer.csv")

```

```

19 #Checks null values
20 df = df.replace('?', np.NaN)
21 print(df.isnull().sum())
22 print('Total number of null values: ', df.isnull().sum().sum())
23
24 #Drops two columns of STDs
25 df.drop(['STDs: Time since first diagnosis','STDs: Time since last
diagnosis'],inplace=True,axis=1)
26
27 numerical_df = ['Age', 'Number of sexual partners', 'First sexual
intercourse','Num of pregnancies', 'Smokes (years)', 'Smokes (packs/year)', 'Hormonal Contraceptives (years)', 'IUD (years)', 'STDs (number)']
28
29 categorical_df = ['Smokes','Hormonal
Contraceptives','IUD','STDs','STDs:condylomatosis','STDs:cervical
condylomatosis',
30 'STDs:vaginal condylomatosis','STDs:vulvo-perineal
condylomatosis', 'STDs:syphilis',
31 'STDs:pelvic inflammatory disease', 'STDs:genital
herpes','STDs:molluscum contagiosum', 'STDs:AIDS',
32 'STDs:HIV','STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of
diagnosis','Dx:Cancer', 'Dx:CIN',
33 'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller','Citology', 'Biopsy']
34
35 #Fills the missing values of numeric data columns with mean of the
36 column data.
37 for feature in numerical_df:
38     print(feature,",df[feature].apply(pd.to_numeric,
39 errors='coerce').mean()")
39         feature_mean = round(df[feature].apply(pd.to_numeric,
40 errors='coerce').mean(),1)
40         df[feature] = df[feature].fillna(feature_mean)
41
42 for feature in categorical_df:
43     df[feature] = df[feature].apply(pd.to_numeric,
44 errors='coerce').fillna(1.0)
45
46 X = df.drop('Biopsy', axis =1).apply(pd.to_numeric,
47 errors='coerce').astype('float64')
48 y = df["Biopsy"]
49 sm = SMOTE(random_state=42)
50 X,y = sm.fit_resample(X, y.ravel())
51
52 #Splits the data into training and testing

```

```
65 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
66 0.2, random_state = 2021, stratify=y)
67
68 #Standar scaler
69 sc = StandardScaler()
70 X_train = sc.fit_transform(X_train)
71 X_test = sc.transform(X_test)
72
73
```

Step 3 Reshape input, compile and train LSTM model:

```
1 # Reshape the data for LSTM input
2 X_train = X_train.reshape(X_train.shape[0], 1,
3 X_train.shape[1])
4 X_test = X_test.reshape(X_test.shape[0], 1,
5 X_test.shape[1])
6
7 # Create an LSTM model
8 lstm_model = tf.keras.models.Sequential([
9     tf.keras.layers.LSTM(units=64, input_shape=
10    (X_train.shape[1], X_train.shape[2])),
11    tf.keras.layers.Dropout(0.5),
12    tf.keras.layers.Dense(units=1,
13 activation='sigmoid')
14])
15
16 print(lstm_model.summary()) # Display the model
17 summary
18
19 # Compile the LSTM model
20 lstm_model.compile(optimizer='adam',
21 loss='binary_crossentropy', metrics=['accuracy'])
22
23 # Train the LSTM model
24 history = lstm_model.fit(X_train, y_train,
batch_size=64, validation_split=0.20, epochs=250,
shuffle=True)

# Save the LSTM model
lstm_model.save('cervical_lstm_model.h5')
```

The code segment outlines the process of creating, training, and saving a Long Short-Term Memory (LSTM) model for predicting cervical cancer risk based on the dataset. Here's an

explanation of each step:

1. Data Reshaping: In this step, the input data for the LSTM model is reshaped to match the required format. The X_train and X_test datasets are reshaped to have dimensions (batch_size, time_steps, features). The reshaping is done to prepare the data for the sequential nature of LSTM, where each instance is represented as a sequence of time steps with associated features.
2. Creating the LSTM Model: The code segment defines an LSTM model using TensorFlow's Keras API. The model is a sequential one, meaning layers are added sequentially. The architecture consists of an LSTM layer with 64 units, which are designed to capture temporal patterns and relationships in the sequential data. A dropout layer is added to mitigate overfitting, followed by a dense layer with a single unit and a sigmoid activation function to produce a binary classification output (0 or 1). The input_shape parameter of the LSTM layer is set to match the reshaped input data's dimensions.
3. Model Compilation: The LSTM model is compiled in this step. The compile function configures the model for training. The chosen optimizer is 'adam', a popular optimization algorithm. The loss function is set to 'binary_crossentropy', which is suitable for binary classification tasks. Additionally, the 'accuracy' metric is specified to monitor the model's performance during training.
4. Training and Saving the Model: The LSTM model is trained using the fit function. It takes the reshaped training data X_train and associated labels y_train, along with other parameters like batch size, validation split, and number of epochs. During training, the model learns to capture patterns and relationships in the data. The training progress is stored in the history variable. After training, the model is saved as an HDF5 file named 'cervical_lstm_model.h5'. This saved model can be loaded and used for making predictions without retraining.

Overall, this code showcases the complete process of creating an LSTM model, training it on the reshaped data, and saving the trained model for future use in predicting cervical cancer risk.

Output:

Model: "sequential_1"

Layer (type)	Output Shape	Param #
<hr/>		
lstm_1 (LSTM)	(None, 64)	25088
dropout_1 (Dropout)	(None, 64)	0

dense_1 (Dense)	(None, 1)	65
-----------------	-----------	----

Total params: 25,153
 Trainable params: 25,153
 Non-trainable params: 0

None

The output is the summary of the LSTM model architecture. Here's an explanation of each part:

1. Model Architecture: The model is a sequential neural network, meaning layers are added sequentially. It starts with an LSTM (Long Short-Term Memory) layer, followed by a dropout layer to mitigate overfitting, and finally, a dense layer.
2. LSTM Layer: The LSTM layer has 64 units. This layer is designed to capture long-range dependencies and temporal patterns in sequential data. The (None, 64) output shape indicates that the model will output a sequence of 64-dimensional vectors for each input sequence.
3. Dropout Layer: The dropout layer helps prevent overfitting by randomly setting a fraction of input units to zero during each update. The (None, 64) output shape remains the same as the input.
4. Dense Layer: The dense layer has a single unit with a sigmoid activation function. This layer is responsible for producing the final classification output. The (None, 1) output shape indicates that the model will produce a single scalar value for each input sequence, representing the predicted risk of cervical cancer.
5. Total Parameters: The total number of parameters in the model is 25,153. These parameters are learned during training and contribute to the model's ability to make accurate predictions.
6. Trainable Parameters: All 25,153 parameters are trainable, meaning they will be updated during the training process to minimize the chosen loss function.
7. Non-trainable Parameters: There are no non-trainable parameters in this model.

Overall, the model's architecture consists of an LSTM layer followed by a dropout layer to prevent overfitting, and a dense layer for final classification. The summary provides insight into the model's structure, the number of parameters, and the shapes of input and output for each layer.

Step 4 Plot accuracy, loss, confusion matrix, and true values versus predicted values diagram:

1 #Plots accuracy and loss

```

2 acc = history.history['accuracy']
3 val_acc = history.history['val_accuracy']
4 loss = history.history['loss']
5 val_loss = history.history['val_loss']
6 epochs = range(1, len(acc) + 1)
7
8 #accuracy
9 fig, ax = plt.subplots(figsize=(25, 15))
10 plt.plot(epochs, acc, 'r', label='Training accuracy', lw=10)
11 plt.plot(epochs, val_acc, 'b--', label='Validation accuracy', lw=10)
12 plt.title('Training and validation accuracy', fontsize=35)
13 plt.legend(fontsize=25)
14 ax.set_xlabel("Epoch", fontsize=30)
15 ax.tick_params(labelsize=30)
16 plt.show()
17
18 #loss
19 fig, ax = plt.subplots(figsize=(25, 15))
20 plt.plot(epochs, loss, 'r', label='Training loss', lw=10)
21 plt.plot(epochs, val_loss, 'b--', label='Validation loss', lw=10)
22 plt.title('Training and validation loss', fontsize=35)
23 plt.legend(fontsize=25)
24 ax.set_xlabel("Epoch", fontsize=30)
25 ax.tick_params(labelsize=30)
26 plt.show()
27
28 #Sets the threshold for the predictions. In this case, the threshold is 0.5
29 (this value can be modified).
30 #prediction on test set
31 y_pred = lstm_model.predict(X_test)
32 y_pred = [int(p>=0.5) for p in y_pred]
33 print(y_pred)
34
35 #Performance Evaluation - Accuracy and Classification Report
36 #Accuracy Score
37 print ('Accuracy Score : ', accuracy_score(y_pred, y_test,
38 normalize=True), '\n')
39
40 #precision, recall report
41 print ('Classification Report :\n\n',classification_report(y_pred, y_test))
42
43 #Confusion matrix:
44 conf_mat = confusion_matrix(y_true=y_test, y_pred = y_pred)
45 class_list = ['Biopsy = 0', 'Biopsy = 1']
46 fig, ax = plt.subplots(figsize=(25, 15))
47

```

```

48 sns.heatmap(conf_mat, annot=True, ax = ax, cmap='YlOrBr', fmt='g',
49 annot_kws={"size": 25})
50 ax.set_xlabel('Predicted labels', fontsize=30)
51 ax.set_ylabel('True labels', fontsize=30)
52 ax.set_title('Confusion Matrix', fontsize=30)
53 ax.xaxis.set_ticklabels(class_list), ax.yaxis.set_ticklabels(class_list)
54
55 def plot_real_pred_val(Y_test, ypred, name):
56     plt.figure(figsize=(25,15))
57     acc=accuracy_score(Y_test,ypred)
58
59     plt.scatter(range(len(ypred)),ypred,color="blue",lw=5,label="Predicted")
60     plt.scatter(range(len(Y_test)), Y_test,color="red",label="Actual")
61     plt.title("Predicted Values vs True Values of " + name, fontsize=10)
62     plt.xlabel("Accuracy: " + str(round((acc*100),3)) + "%")
63     plt.legend(fontsize=25)
64     plt.grid(True, alpha=0.75, lw=1, ls='-.')
65     plt.show()
66
67 plot_real_pred_val(y_test, y_pred, 'LSTM')

```

Cervical Cancer Using Self Organizing Maps (SOMs) Model

Step 1 This is the full source code to implement cervical cancer using Self Organizing Maps (SOMs) model:

```

#cervical_RBFNs
import numpy as np
import pandas as pd
import os
from minisom import MiniSom
import seaborn as sns
sns.set_style('darkgrid')
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report,
accuracy_score
import tensorflow as tf
from sklearn.cluster import KMeans
from sklearn.metrics.pairwise import euclidean_distances

# Reads dataset

```

```

curr_path = os.getcwd()
df = pd.read_csv(curr_path + "/kag_risk_factors_cervical_cancer.csv")

# Checks null values
df = df.replace('?', np.NaN)
print(df.isnull().sum())
print('Total number of null values: ', df.isnull().sum().sum())

# Drops two columns of STDs
df.drop(['STDs: Time since first diagnosis','STDs: Time since last
diagnosis'], inplace=True, axis=1)

numerical_df = ['Age', 'Number of sexual partners', 'First sexual
intercourse','Num of pregnancies', 'Smokes (years)',
'Smokes (packs/year)', 'Hormonal Contraceptives (years)', 'IUD
(years)', 'STDs (number)']

categorical_df = ['Smokes', 'Hormonal
Contraceptives', 'IUD', 'STDs', 'STDs:condylomatosis', 'STDs:cervical
condylomatosis',
'STDs:vaginal condylomatosis', 'STDs:vulvo-perineal condylomatosis',
'STDs:syphilis',
'STDs:pelvic inflammatory disease', 'STDs:genital
herpes', 'STDs:molluscum contagiosum', 'STDs:AIDS',
'STDs:HIV', 'STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of
diagnosis', 'Dx:Cancer', 'Dx:CIN',
'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller', 'Citology', 'Biopsy']

# Fills the missing values of numeric data columns with mean of the
column data.
for feature in numerical_df:
    print(feature, ",df[feature].apply(pd.to_numeric,
errors='coerce').mean()")
    feature_mean = round(df[feature].apply(pd.to_numeric,
errors='coerce').mean(), 1)
    df[feature] = df[feature].fillna(feature_mean)

for feature in categorical_df:
    df[feature] = df[feature].apply(pd.to_numeric,
errors='coerce').fillna(1.0)

X = df.drop('Biopsy', axis=1).apply(pd.to_numeric,
errors='coerce').astype('float64')
y = df["Biopsy"]

# Standard scaler

```

```

sc = StandardScaler()
X = sc.fit_transform(X)

# Apply KMeans for SOM initialization
n_centers = 100
kmeans = KMeans(n_clusters=n_centers)
kmeans.fit(X)
centers = kmeans.cluster_centers_

# Train a Self Organizing Map (SOM)
som = MiniSom(x=10, y=10, input_len=X.shape[1], sigma=1.0,
learning_rate=0.5)
som.random_weights_init(X)
som.train_random(data=X, num_iteration=100)

# Transform input data into the SOM feature space
X_som = np.array([som.winner(x) for x in X])
X_som = X_som.reshape(X_som.shape[0], -1)

# Split the data into training and testing
X_train, X_test, y_train, y_test = train_test_split(X_som, y,
test_size=0.2, random_state=2021, stratify=y)

# Reshape the data for LSTM input
X_train = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
X_test = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])

# Create an LSTM model
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(units=64, input_shape=(X_train.shape[1],
X_train.shape[2])),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])
print(lstm_model.summary()) # Display the model summary

# Compile the LSTM model
lstm_model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train the LSTM model
history = lstm_model.fit(X_train, y_train, batch_size=64,
validation_split=0.2, epochs=250, shuffle=True)

# Save the LSTM model

```

```

lstm_model.save('cervical_lstm_model_with_som.h5')

#Plots accuracy and loss
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)

#accuracy
fig, ax = plt.subplots(figsize=(25, 15))
plt.plot(epochs, acc, 'r', label='Training accuracy', lw=10)
plt.plot(epochs, val_acc, 'b--', label='Validation accuracy', lw=10)
plt.title('Training and validation accuracy', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

#loss
fig, ax = plt.subplots(figsize=(25, 15))
plt.plot(epochs, loss, 'r', label='Training loss', lw=10)
plt.plot(epochs, val_loss, 'b--', label='Validation loss', lw=10)
plt.title('Training and validation loss', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

#Sets the threshold for the predictions. In this case, the threshold is 0.5
#(this value can be modified).
#prediction on test set
y_pred = lstm_model.predict(X_test)
y_pred = [int(p>=0.5) for p in y_pred]
print(y_pred)

#Performance Evaluation - Accuracy and Classification Report
#Accuracy Score
print ('Accuracy Score : ', accuracy_score(y_pred, y_test,
normalize=True), '\n')

#precision, recall report
print ('Classification Report :\n\n', classification_report(y_pred, y_test))

#Confusion matrix:
conf_mat = confusion_matrix(y_true=y_test, y_pred = y_pred)

```

```
class_list = ['Biopsy = 0', 'Biopsy = 1']
fig, ax = plt.subplots(figsize=(25, 15))
sns.heatmap(conf_mat, annot=True, ax = ax, cmap='YlOrBr', fmt='g',
annot_kws={"size": 25})
ax.set_xlabel('Predicted labels', fontsize=30)
ax.set_ylabel('True labels', fontsize=30)
ax.set_title('Confusion Matrix', fontsize=30)
ax.xaxis.set_ticklabels(class_list), ax.yaxis.set_ticklabels(class_list)

def plot_real_pred_val(Y_test, ypred, name):
    plt.figure(figsize=(25,15))
    acc=accuracy_score(Y_test,ypred)

    plt.scatter(range(len(ypred)),ypred,color="blue",lw=5,label="Predicted")
    plt.scatter(range(len(Y_test)), Y_test,color="red",label="Actual")
    plt.title("Predicted Values vs True Values of " + name, fontsize=10)
    plt.xlabel("Accuracy: " + str(round((acc*100),3)) + "%")
```

```
plt.legend(fontsize=25)
plt.grid(True, alpha=0.75, lw=1, ls='-.')
plt.show()

plot_real_pred_val(y_test, y_pred, 'SOM_LSTM')
```

This code implements a workflow using Self Organizing Maps (SOMs) and Long Short-Term Memory (LSTM) neural networks for a classification task on the cervical cancer risk factors dataset. Here's a detailed explanation of each section of the code:

1. Importing Libraries:

- numpy, pandas, os: Basic data manipulation libraries.
- minisom: Library for creating and training Self Organizing Maps (SOMs).
- seaborn, matplotlib.pyplot: Libraries for data visualization.
- sklearn: Library for machine learning utilities.
- tensorflow: Deep learning framework.

2. Reading and Preprocessing the Dataset:

- Reads the cervical cancer risk factors dataset and handles missing values.
- Drops two columns related to STDs.
- Separates features into numerical and categorical data.
- Fills missing values in numerical features with the mean.
- Converts categorical features into numeric and fills missing values with 1.0.
- Scales the feature data using StandardScaler.

3. Applying KMeans for SOM Initialization:

- Performs KMeans clustering to initialize the centers of the Self Organizing Map (SOM).
- n_centers determines the number of clusters.

- The centers are used as initial weights for the SOM.

4. Training Self Organizing Map (SOM):

- Creates a Self Organizing Map (SOM) with a 10x10 grid and input length corresponding to the number of features.
- Initializes the SOM's weights with the KMeans centers.
- Trains the SOM with input data using random training order for 100 iterations.
- Transforms input data into the SOM feature space by finding the winning neuron for each input.

5. Data Splitting and Reshaping:

- Splits the transformed SOM data into training and testing sets.
- Reshapes the data to be suitable for LSTM input (3D shape).

6. Creating and Training LSTM Model:

- Constructs an LSTM model using TensorFlow's Keras API.
- The LSTM layer has 64 units followed by dropout and a sigmoid activation output layer.
- Compiles the model with 'adam' optimizer and binary cross-entropy loss.
- Trains the model using the training data, validates on a portion of it, and records training history.

7. Plotting Accuracy and Loss:

- Plots training and validation accuracy over epochs.
- Plots training and validation loss over epochs.

8. Prediction and Evaluation:

- Makes predictions on the test set using the trained LSTM model.
- Converts probabilities to binary class predictions based on a threshold (0.5).
- Calculates and prints accuracy score and classification report.

9. Confusion Matrix Visualization:

- Computes and visualizes the confusion matrix using a heatmap.

10. Real vs. Predicted Value Plot:

- Compares the predicted values against the actual values in a scatter plot.
- Displays the accuracy of predictions on the plot.

11. Output Visualization:

Calls the `plot_real_pred_val()` function to visualize real vs. predicted values.

This code effectively combines Self Organizing Maps (SOMs) with a Long Short-Term Memory (LSTM) neural network for a classification task, showcasing how SOMs can be used for feature transformation before feeding the data into an LSTM model. The final output includes visualization of accuracy, loss, confusion matrix, and a comparison of actual vs. predicted values.

Cervical Cancer Using Deep Belief Networks (DBNs) Model

Step 1 This is the full source code to implement cervical cancer using Deep Belief Networks (DBNs) model:

```
#cervical_DBNs
import numpy as np
import pandas as pd
import os
import seaborn as sns
```

```
sns.set_style('darkgrid')
from matplotlib import pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from sklearn.metrics import confusion_matrix,
classification_report, accuracy_score
from imblearn.over_sampling import SMOTE

# Reads dataset
curr_path = os.getcwd()
df = pd.read_csv(curr_path +
"/kag_risk_factors_cervical_cancer.csv")

# Checks null values
df = df.replace('?', np.NaN)
print(df.isnull().sum())
print('Total number of null values: ', df.isnull().sum().sum())

# Drops two columns of STDs
df.drop(['STDs: Time since first diagnosis','STDs: Time since last
diagnosis'], inplace=True, axis=1)

numerical_df = ['Age', 'Number of sexual partners', 'First sexual
intercourse','Num of pregnancies', 'Smokes (years)',
'Smokes (packs/year)', 'Hormonal Contraceptives (years)', 'IUD
(years)', 'STDs (number)']

categorical_df = ['Smokes', 'Hormonal
Contraceptives', 'IUD', 'STDs', 'STDs:condylomatosis', 'STDs:cervical
condylomatosis',
'STDs:vaginal condylomatosis', 'STDs:vulvo-perineal
condylomatosis', 'STDs:syphilis',
'STDs:pelvic inflammatory disease', 'STDs:genital
herpes', 'STDs:molluscum contagiosum', 'STDs:AIDS',
'STDs:HIV', 'STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of
diagnosis', 'Dx:Cancer', 'Dx:CIN',
'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller', 'Cytology', 'Biopsy']
```

```
# Fills the missing values of numeric data columns with mean of  
the column data.  
for feature in numerical_df:  
    print(feature,",df[feature].apply(pd.to_numeric,  
errors='coerce').mean())  
    feature_mean = round(df[feature].apply(pd.to_numeric,  
errors='coerce').mean(), 1)  
    df[feature] = df[feature].fillna(feature_mean)  
  
for feature in categorical_df:  
    df[feature] = df[feature].apply(pd.to_numeric,  
errors='coerce').fillna(1.0)  
  
X = df.drop('Biopsy', axis=1).apply(pd.to_numeric,  
errors='coerce').astype('float64')  
y = df["Biopsy"]  
sm = SMOTE(random_state=42)  
X,y = sm.fit_resample(X, y.ravel())  
  
# Splits the data into training and testing  
X_train, X_test, y_train, y_test = train_test_split(X, y,  
test_size=0.2, random_state=2021, stratify=y)  
  
# Standard scaler  
sc = StandardScaler()  
X_train = sc.fit_transform(X_train)  
X_test = sc.transform(X_test)  
  
# Create a Deep Belief Network (DBN)  
dbn_model = tf.keras.models.Sequential([  
    tf.keras.layers.Dense(units=500, input_dim=33,  
    kernel_initializer='uniform', activation='relu'),  
    tf.keras.layers.Dropout(0.5),  
    tf.keras.layers.Dense(units=200, kernel_initializer='uniform',  
    activation='relu'),  
    tf.keras.layers.Dropout(0.5),  
    tf.keras.layers.Dense(units=1, kernel_initializer='uniform',  
    activation='sigmoid')
```

1)

```
print(dbn_model.summary()) # Display the model summary

# Compile the DBN model
dbn_model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train the DBN model
history = dbn_model.fit(X_train, y_train, batch_size=64,
validation_split=0.20, epochs=250, shuffle=True)

# Save the DBN model
dbn_model.save('cervical_dbn_model.h5')

# Save history into npy file
np.save('cervical_dbn_history.npy', history.history)

print(history.history.keys())

# Plots accuracy and loss
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)

# Accuracy
fig, ax = plt.subplots(figsize=(25, 15))
plt.plot(epochs, acc, 'r', label='Training accuracy', lw=10)
plt.plot(epochs, val_acc, 'b--', label='Validation accuracy', lw=10)
plt.title('Training and validation accuracy', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

# Loss
fig, ax = plt.subplots(figsize=(25, 15))
```

```

plt.plot(epochs, loss, 'r', label='Training loss', lw=10)
plt.plot(epochs, val_loss, 'b--', label='Validation loss', lw=10)
plt.title('Training and validation loss', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

# Sets the threshold for the predictions. In this case, the threshold is
# 0.5 (this value can be modified).
# Prediction on test set
y_pred = dbn_model.predict(X_test)
y_pred = [int(p>=0.5) for p in y_pred]
print(y_pred)

# Performance Evaluation - Accuracy and Classification Report
# Accuracy Score
print('Accuracy Score : ', accuracy_score(y_pred, y_test,
normalize=True), '\n')

# Precision, recall report
print('Classification Report :\n\n', classification_report(y_pred,
y_test))

# Confusion matrix
conf_mat = confusion_matrix(y_true=y_test, y_pred=y_pred)
class_list = ['Biopsy = 0', 'Biopsy = 1']
fig, ax = plt.subplots(figsize=(25, 15))
sns.heatmap(conf_mat, annot=True, ax=ax, cmap='YlOrBr',
fmt='g', annot_kws={"size": 25})
ax.set_xlabel('Predicted labels', fontsize=30)
ax.set_ylabel('True labels', fontsize=30)
ax.set_title('Confusion Matrix', fontsize=30)
ax.xaxis.set_ticklabels(class_list),
ax.yaxis.set_ticklabels(class_list)

def plot_real_pred_val(Y_test, ypred, name):
    plt.figure(figsize=(25, 15))
    acc = accuracy_score(Y_test, ypred)

```

```

plt.scatter(range(len(ypred)), ypred, color="blue", lw=5,
label="Predicted")
plt.scatter(range(len(Y_test)), Y_test, color="red",
label="Actual")
plt.title("Predicted Values vs True Values of " + name,
fontsize=10)
plt.xlabel("Accuracy: " + str(round((acc*100), 3)) + "%")
plt.legend(fontsize=25)
plt.grid(True, alpha=0.75, lw=1, ls='-.')
plt.show()

plot_real_pred_val(y_test, y_pred, 'DBN')

```

The code step by step is explained as follows:

1. Importing Libraries:

The code begins by importing necessary libraries such as NumPy, Pandas, OS, seaborn, matplotlib, scikit-learn, TensorFlow, and imbalanced-learn (for SMOTE).

2. Reading and Preprocessing Data:

- The current working directory path is obtained using os.getcwd().
- The dataset is read using Pandas from a CSV file.
- Missing values represented as '?' are replaced with NaN (Not a Number) using df.replace('?', np.NaN).
- The code checks for the count of null values in each column using df.isnull().sum(). The sum of all null values across the dataset is also displayed.
- Two columns related to STDs are dropped using df.drop(), as they are not needed.
- Lists numerical_df and categorical_df are created containing names of numerical and categorical features respectively.

- Missing values in numerical features are filled with the mean value of the column using `df[feature].fillna(feature_mean)`.
- Categorical features are converted to numeric and missing values are filled with 1.0 using `df[feature].apply(pd.to_numeric, errors='coerce').fillna(1.0)`.
- The input features X are extracted by dropping the target variable 'Biopsy', and the target variable y is extracted.

3. Handling Imbalance:

- SMOTE (Synthetic Minority Over-sampling Technique) is used to balance the classes by generating synthetic samples of the minority class.
- `sm = SMOTE(random_state=42)` initializes the SMOTE object, then `X,y = sm.fit_resample(X, y.ravel())` applies SMOTE to create balanced data.

4. Data Splitting and Standardization:

- The data is split into training and testing sets using `train_test_split()`. The `stratify` parameter ensures class balance in both sets.
- Standardization is performed using `StandardScaler`. `sc.fit_transform()` scales the training data and `sc.transform()` scales the testing data.

5. Creating the Deep Belief Network (DBN) Model:

- A Sequential model `dbn_model` is created using Keras with three Dense layers and Dropout layers in between.
- Each Dense layer has a specific number of units, specified activation function (ReLU for

hidden layers, sigmoid for output), and dropout to prevent overfitting.

6. Compiling and Training the DBN Model:

- The model is compiled using the Adam optimizer and binary cross-entropy loss for binary classification.
- The model is trained using the training data (`X_train`, `y_train`) and validated using a subset (20%) of the training data.
- The training history (`history`) is recorded to analyze the training process later.

7. Saving Model and History:

- The trained model is saved to a file named '`cervical_dbn_model.h5`' using `dbn_model.save()`.
- The training history is saved as a NumPy file '`cervical_dbn_history.npy`' using `np.save()`.

8. Plotting Training Metrics:

- The code extracts accuracy and loss values from the training history.
- Matplotlib is used to create two plots: one showing accuracy (training and validation) and another showing loss (training and validation).

9. Making Predictions and Evaluation:

- The trained DBN model is used to predict outcomes on the test data.
- A threshold of 0.5 is used to classify the predictions into binary values.
- Accuracy, classification report, and confusion matrix are computed and printed to evaluate the model's performance.

10. Visualization:

- A heatmap of the confusion matrix is plotted using Seaborn to visualize true positive, true negative, false positive, and false negative predictions.
- The function `plot_real_pred_val()` is defined to visualize predicted values against true values using scatter plots.

In summary, this code performs data preprocessing, creates and trains a Deep Belief Network (DBN) model for cervical cancer diagnosis, saves the trained model and training history, plots training metrics, makes predictions, and evaluates the model's performance using accuracy, classification report, and confusion matrix. Visualizations are used to understand the model's behavior and predictions.

Cervical Cancer Using Restricted Boltzmann Machines (RBMs) Model

Step 1 This is the full source code to implement cervical cancer using Restricted Boltzmann Machines (RBMs) model:

```
#cervical_rbm.py
import numpy as np
import pandas as pd
import os
import seaborn as sns
sns.set_style('darkgrid')
from matplotlib import pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from sklearn.metrics import confusion_matrix, classification_report,
accuracy_score
from imblearn.over_sampling import SMOTE
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
```

```
from sklearn.linear_model import LogisticRegression

#Reads dataset
curr_path = os.getcwd()
df = pd.read_csv(curr_path+"/kag_risk_factors_cervical_cancer.csv")

#Checks null values
df = df.replace('?', np.NaN)
print(df.isnull().sum())
print('Total number of null values: ', df.isnull().sum().sum())

#Drops two columns of STDs
df.drop(['STDs: Time since first diagnosis','STDs: Time since last diagnosis'],inplace=True,axis=1)

numerical_df = ['Age', 'Number of sexual partners', 'First sexual intercourse','Num of pregnancies', 'Smokes (years)', 'Smokes (packs/year)', 'Hormonal Contraceptives (years)', 'IUD (years)', 'STDs (number)']

categorical_df = ['Smokes','Hormonal Contraceptives','IUD','STDs','STDs:condylomatosis','STDs:cervical condylomatosis', 'STDs:vaginal condylomatosis','STDs:vulvo-perineal condylomatosis', 'STDs:syphilis', 'STDs:pelvic inflammatory disease', 'STDs:genital herpes', 'STDs:molluscum contagiosum', 'STDs:AIDS', 'STDs:HIV','STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of diagnosis','Dx:Cancer', 'Dx:CIN', 'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller','Citology', 'Biopsy']

#Fills the missing values of numeric data columns with mean of the column data.
for feature in numerical_df:
    print(feature, "df[feature].apply(pd.to_numeric, errors='coerce').mean()")
    feature_mean = round(df[feature].apply(pd.to_numeric, errors='coerce').mean(),1)
    df[feature] = df[feature].fillna(feature_mean)
```

```
for feature in categorical_df:  
    df[feature] = df[feature].apply(pd.to_numeric,  
    errors='coerce').fillna(1.0)  
  
X = df.drop('Biopsy', axis=1).apply(pd.to_numeric,  
errors='coerce').astype('float64')  
y = df["Biopsy"]  
sm = SMOTE(random_state=42)  
X,y = sm.fit_resample(X, y.ravel())  
  
#Splits the data into training and testing  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,  
random_state = 2021, stratify=y)  
  
#Standard scaler  
sc = StandardScaler()  
X_train = sc.fit_transform(X_train)  
X_test = sc.transform(X_test)  
  
#Creating the RBM and Logistic Regression Classifier pipeline  
rbm = BernoulliRBM(n_components=256, n_iter=10,  
learning_rate=0.01, verbose=True)  
logreg = LogisticRegression(max_iter=100, random_state=42)  
  
rbm_features_classifier = Pipeline(  
    steps=[('rbm', rbm), ('logreg', logreg)]  
)  
  
#Training the RBM and Logistic Regression Classifier  
rbm_features_classifier.fit(X_train, y_train)  
  
#Predicting using RBM-transformed features  
y_pred = rbm_features_classifier.predict(X_test)  
print(y_pred)  
  
#Performance Evaluation - Accuracy and Classification Report  
#Accuracy Score
```

```

print ('Accuracy Score : ', accuracy_score(y_pred, y_test,
normalize=True), '\n')

#Precision, recall report
print ('Classification Report :\n\n',classification_report(y_pred, y_test))

#Confusion matrix
conf_mat = confusion_matrix(y_true=y_test, y_pred = y_pred)
class_list = ['Biopsy = 0', 'Biopsy = 1']
fig, ax = plt.subplots(figsize=(25, 15))
sns.heatmap(conf_mat, annot=True, ax = ax, cmap='YlOrBr', fmt='g',
annot_kws={"size": 25})
ax.set_xlabel('Predicted labels', fontsize=30)
ax.set_ylabel('True labels', fontsize=30)
ax.set_title('Confusion Matrix', fontsize=30)
ax.xaxis.set_ticklabels(class_list), ax.yaxis.set_ticklabels(class_list)

def plot_real_pred_val(Y_test, ypred, name):
    plt.figure(figsize=(25,15))
    acc=accuracy_score(Y_test,ypred)

    plt.scatter(range(len(ypred)),ypred,color="blue",lw=5,label="Predicted")
    plt.scatter(range(len(Y_test)), Y_test,color="red",label="Actual")
    plt.title("Predicted Values vs True Values of " + name, fontsize=10)
    plt.xlabel("Accuracy: " + str(round((acc*100),3)) + "%")
    plt.legend(fontsize=25)
    plt.grid(True, alpha=0.75, lw=1, ls='-.')
    plt.show()

plot_real_pred_val(y_test, y_pred, 'RBM')

```

The code step by step is explained as follows:

1. Imports and Setup: Importing necessary libraries and setting up the visualization style for plots using seaborn.
2. Reads and Preprocesses Dataset: The code reads a dataset from a CSV file and handles missing values. It replaces any '?' in the dataset with NaN (Not a Number) for consistent handling.

3. Drop Columns: Two columns related to STDs are dropped since they are not needed for the analysis.
4. Numerical and Categorical Features: Separate lists are created for numerical and categorical features. These lists help organize the features based on their types.
5. Fill Missing Values: The missing values in the dataset are filled. For numerical columns, the missing values are replaced with the mean of the respective column. For categorical columns, they are converted to numeric values and missing values are filled with 1.
6. Prepare Data and Oversampling: The data is prepared for training. It is transformed into numerical format, and then the Synthetic Minority Over-sampling Technique (SMOTE) is applied to handle class imbalance by generating synthetic samples for the minority class.
7. Train-Test Split and Scaling: The data is split into training and testing sets. The features are standardized using the StandardScaler, which helps bring all features to a similar scale, which can improve the training of the models.
8. Creating RBM and Logistic Regression Pipeline: A pipeline is created that consists of two steps: first, applying the Bernoulli Restricted Boltzmann Machine (RBM) transformation, and second, applying Logistic Regression for classification. The RBM is used to learn useful features from the input data before feeding it into the classifier.
9. Training the Classifier: The pipeline is trained on the training data. This involves fitting the RBM and then using the transformed features to train the Logistic Regression classifier.
10. Predictions and Performance Evaluation:
 - The trained pipeline is used to predict on the test data.

- The accuracy score is calculated, which measures the proportion of correct predictions.
 - The classification report is generated, including metrics such as precision, recall, F1-score, and support for each class. This report provides a more detailed understanding of the model's performance.
11. Confusion Matrix Visualization: The confusion matrix is computed and visualized using a heatmap. It shows how well the predicted labels match the true labels for different classes.
12. Plot Real vs. Predicted Values: A function is defined to create a scatter plot comparing the true values with the predicted values. This provides a visual representation of how well the model's predictions match the actual values.
13. Call the Plotting Function: The function defined in the previous step is called to generate a scatter plot that compares the true and predicted values.

In essence, this code performs data preprocessing, transforms the features using an RBM, trains a Logistic Regression classifier on the transformed features, and evaluates the classifier's performance. The focus is on utilizing an RBM to extract meaningful features from the data, which are then used for classification. The code concludes with visualizations to help understand the quality of the predictions and the overall performance of the model.

Cervical Cancer Using Autoencoders Model

Step 1 This is the full source code to implement cervical cancer using Autoencoders model:

```
#cervical_autoencoders.py
import numpy as np
import pandas as pd
```

```
import os
import seaborn as sns
sns.set_style('darkgrid')
from matplotlib import pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from sklearn.metrics import confusion_matrix, classification_report,
accuracy_score
from imblearn.over_sampling import SMOTE

# Reads dataset
curr_path = os.getcwd()
df = pd.read_csv(curr_path + "/kag_risk_factors_cervical_cancer.csv")

# Checks null values
df = df.replace('?', np.NaN)
print(df.isnull().sum())
print('Total number of null values: ', df.isnull().sum().sum())

# Drops two columns of STDs
df.drop(['STDs: Time since first diagnosis', 'STDs: Time since last
diagnosis'], inplace=True, axis=1)

numerical_df = ['Age', 'Number of sexual partners', 'First sexual
intercourse', 'Num of pregnancies', 'Smokes (years)',
'Smokes (packs/year)', 'Hormonal Contraceptives (years)', 'IUD (years)',
'STDs (number)']

categorical_df = ['Smokes', 'Hormonal Contraceptives', 'IUD', 'STDs',
'STDs:condylomatosis', 'STDs:cervical condylomatosis',
'STDs:vaginal condylomatosis', 'STDs:vulvo-perineal condylomatosis',
'STDs:syphilis',
'STDs:pelvic inflammatory disease', 'STDs:genital herpes',
'STDs:molluscum contagiosum', 'STDs:AIDS',
'STDs:HIV', 'STDs:Hepatitis B', 'STDs:HPV', 'STDs: Number of
diagnosis', 'Dx:Cancer', 'Dx:CIN',
'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller', 'Cytology', 'Biopsy']
```

```
# Fills the missing values of numeric data columns with the mean of the
# column data.
for feature in numerical_df:
    print(feature, ", df[feature].apply(pd.to_numeric,
errors='coerce').mean())
    feature_mean = round(df[feature].apply(pd.to_numeric,
errors='coerce').mean(), 1)
    df[feature] = df[feature].fillna(feature_mean)

for feature in categorical_df:
    df[feature] = df[feature].apply(pd.to_numeric,
errors='coerce').fillna(1.0)

X = df.drop('Biopsy', axis=1).apply(pd.to_numeric,
errors='coerce').astype('float64')
y = df["Biopsy"]
sm = SMOTE(random_state=42)
X, y = sm.fit_resample(X, y.ravel())

# Splits the data into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=2021, stratify=y)

# Standard scaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Autoencoder Model
input_layer = tf.keras.layers.Input(shape=(X_train.shape[1],))
encoder = tf.keras.layers.Dense(units=128, activation='relu')(input_layer)
encoder = tf.keras.layers.Dense(units=64, activation='relu')(encoder)
encoder = tf.keras.layers.Dense(units=32, activation='relu')(encoder)
decoder = tf.keras.layers.Dense(units=64, activation='relu')(encoder)
decoder = tf.keras.layers.Dense(units=128, activation='relu')(decoder)
decoder = tf.keras.layers.Dense(units=X_train.shape[1],
activation='linear')(decoder)
```

```
autoencoder = tf.keras.models.Model(inputs=input_layer,  
outputs=decoder)  
  
autoencoder.compile(optimizer='adam', loss='mean_squared_error')  
  
autoencoder.fit(X_train, X_train, batch_size=64, epochs=100,  
validation_split=0.2, shuffle=True)  
  
# Transform features using the Autoencoder  
encoded_features = autoencoder.predict(X_train)  
encoded_features_test = autoencoder.predict(X_test)  
  
# Imports Tensorflow and create a Sequential Model to add a layer for  
the Logistic Regression  
logreg = tf.keras.models.Sequential()  
  
# Input layer  
logreg.add(tf.keras.layers.Input(shape=(encoded_features.shape[1],)))  
  
# Output layer  
logreg.add(tf.keras.layers.Dense(units=1, kernel_initializer='uniform',  
activation='sigmoid'))  
  
print(logreg.summary()) # for showing the structure and parameters  
  
# Compiles the Logistic Regression using ADAM optimizer.  
logreg.compile(optimizer='adam', loss='binary_crossentropy', metrics=[  
'accuracy'])  
  
# Trains the Logistic Regression with 100 epochs.  
logreg_history = logreg.fit(encoded_features, y_train, batch_size=64,  
validation_split=0.2, epochs=250, shuffle=True)  
  
# Predict using Logistic Regression  
y_pred = logreg.predict(encoded_features_test)  
y_pred = [int(p >= 0.5) for p in y_pred]  
print(y_pred)
```

```

# Performance Evaluation - Accuracy and Classification Report
# Accuracy Score
print('Accuracy Score : ', accuracy_score(y_pred, y_test,
normalize=True), '\n')

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

# Plots accuracy and loss
acc = logreg_history.history['accuracy']
val_acc = logreg_history.history['val_accuracy']
loss = logreg_history.history['loss']
val_loss = logreg_history.history['val_loss']
epochs = range(1, len(acc) + 1)

# Accuracy
fig, ax = plt.subplots(figsize=(25, 15))
plt.plot(epochs, acc, 'r', label='Training accuracy', lw=10)
plt.plot(epochs, val_acc, 'b--', label='Validation accuracy', lw=10)
plt.title('Training and validation accuracy', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

# Loss
fig, ax = plt.subplots(figsize=(25, 15))
plt.plot(epochs, loss, 'r', label='Training loss', lw=10)
plt.plot(epochs, val_loss, 'b--', label='Validation loss', lw=10)
plt.title('Training and validation loss', fontsize=35)
plt.legend(fontsize=25)
ax.set_xlabel("Epoch", fontsize=30)
ax.tick_params(labelsize=30)
plt.show()

# Confusion matrix:
conf_mat = confusion_matrix(y_true=y_test, y_pred=y_pred)
class_list = ['Biopsy = 0', 'Biopsy = 1']
fig, ax = plt.subplots(figsize=(25, 15))

```

```

sns.heatmap(conf_mat, annot=True, ax=ax, cmap='YlOrBr', fmt='g',
annot_kws={"size": 25})
ax.set_xlabel('Predicted labels', fontsize=30)
ax.set_ylabel('True labels', fontsize=30)
ax.set_title('Confusion Matrix', fontsize=30)
ax.xaxis.set_ticklabels(class_list), ax.yaxis.set_ticklabels(class_list)
plt.show()

def plot_real_pred_val(Y_test, ypred, name):
    plt.figure(figsize=(25,15))
    acc=accuracy_score(Y_test,ypred)

    plt.scatter(range(len(ypred)),ypred,color="blue",lw=5,label="Predicted")
    plt.scatter(range(len(Y_test)), Y_test,color="red",label="Actual")
    plt.title("Predicted Values vs True Values of " + name, fontsize=10)
    plt.xlabel("Accuracy: " + str(round((acc*100),3)) + "%")
    plt.legend(fontsize=25)
    plt.grid(True, alpha=0.75, lw=1, ls='-.')
    plt.show()

plot_real_pred_val(y_test, y_pred, 'Autoencoders')

```

The code step by step is explained as follows:

1. Imports and Setup: The initial section imports necessary libraries for data manipulation, visualization, preprocessing, machine learning, and evaluation. It sets up the visualization style using Seaborn and identifies the current working directory. The dataset is loaded into a Pandas DataFrame.
2. Data Preprocessing: This section handles missing values in the dataset. It replaces question mark values ('?') with NaN and then drops two columns related to STDs. The numerical and categorical columns are defined to categorize features. Missing values in numerical columns are filled with the mean of each column, and categorical columns are converted to a numeric format

by applying a function to convert the values to numeric, and missing values are filled with 1.0.

3. Data Splitting and Scaling: The dataset is split into training and testing sets using the `train_test_split` function. Features are standardized using the `StandardScaler` to ensure that they have zero mean and unit variance.
4. Autoencoder Model: An autoencoder model is created using the Keras functional API. The autoencoder aims to learn a compressed representation of the input data. It consists of an input layer followed by three encoder layers and three decoder layers. Each encoder layer reduces the dimensionality of the data, capturing its essential features. The decoder layers aim to reconstruct the original data from the encoded representation.
5. Autoencoder Training: The autoencoder is compiled with the '`adam`' optimizer and '`mean_squared_error`' loss function. It is trained using the training data (input and output are the same), with a batch size of 64 and over 100 epochs. During training, the autoencoder learns to minimize the difference between the input and the reconstructed output.
6. Feature Transformation: The trained autoencoder is used to transform the features. It encodes the original features into a lower-dimensional representation, which captures essential information about the data.
7. Logistic Regression Model: A logistic regression model is defined using a Sequential Keras model. It has a single input layer with the same shape as the encoded features and an output layer with a sigmoid activation function. This simple model will use the encoded features for prediction.
8. Logistic Regression Training: The logistic regression model is compiled with the '`adam`' optimizer and

'binary_crossentropy' loss function. It is trained using the encoded features obtained from the autoencoder and the corresponding training labels. The model aims to learn the relationship between the encoded features and the target labels.

9. Predictions and Performance Evaluation: The trained logistic regression model is used to make predictions on the encoded features of the testing data. The predictions are thresholded at 0.5 to convert them into binary predictions (0 or 1). Accuracy, classification report, and confusion matrix are computed and displayed to evaluate the model's performance.
10. Accuracy and Loss Plots: The code generates line plots to visualize the training and validation accuracy as well as the training and validation loss of the logistic regression model. These plots help in understanding how well the model is learning over the epochs.
11. Confusion Matrix Plot: A heatmap of the confusion matrix is plotted using Seaborn to visualize the model's performance in terms of true positive, true negative, false positive, and false negative predictions.
12. Real vs. Predicted Values Plot: The function `plot_real_pred_val` is defined to create a scatter plot comparing the actual values and predicted values of the test set. The points in the scatter plot represent individual instances, with blue points indicating predicted values and red points indicating actual values. This plot visually assesses the model's performance on the test set.

In summary, the code preprocesses the dataset, constructs an autoencoder to learn essential features, uses the encoded features to train a logistic regression model, evaluates the model's performance, and generates visualizations to aid in understanding the model's behavior and effectiveness.



IMPLEMENTING GRAPHICAL USER INTERFACE USING PYQT

IMPLEMENTING GRAPHICAL USER INTERFACE USING PYQT

Designing GUI

- Step 1 Now, you will create a GUI to implement how to analyze and predict cervical cancer using some machine learning algorithms and ANN. Open **Qt Designer** and choose **Main Window** template. Save the form as **gui_cervical.ui**.
- Step 2 Put three **Push Button** widgets onto form. Set their **text** property as **LOAD DATA**, **TRAIN ML MODEL**, and **TRAIN DL MODEL**. Set their **objectName** property as **pbLoad**, **pbTrainML**, dan **pbTrainDL**.
- Step 3 Put two **Table Widget** onto form. Set their **objectName** properties as **twData1** and **twData2**.
- Step 4 Add two **Label** widgets onto form. Set their **text** properties as **Label 1** and **Label 2** and set their **objectName** properties as **label1** and **label2**.
- Step 5 Put three **Widget** from **Containers** panel onto form and set their **ObjectName** property as **widgetPlot1**, **widgetPlot2**, and **widgetPlot3**.

Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
cbClassifier	QComboBox
cbData	QComboBox
cbPrediction	QComboBox
label	QLabel
label1	QLabel
label2	QLabel
label_2	QLabel
label_3	QLabel
pbLoad	QPushButton
pbTrain	QPushButton
twData1	QTableWidget
twData2	QTableWidget
widgetPlot1	plot_class
widgetPlot2	plot_class
widgetPlot3	plot_class
menubar	QMenuBar
statusbar	QStatusBar

Figure 117 The **widgetPlot1**, **widgetPlot2**, and **widgetPlot3** are now an object of **plot_class**

Step 6 Right click on the three **Widgets** and choose **Promote to** Set **Promoted class name** as **plot_class**. Click **Add** and **Promote** button. In **Object Inspector** window, you can see that **widgetPlot1**, **widgetPlot2**, and **widgetPlot3** are now an object of **plot_class** as shown in Figure 117.

Step 7 Write the definition of **plot_class** and save it as **plot_class.py** as follows:

```

1 #plot_class.py
2 from PyQt5.QtWidgets import*
3 from matplotlib.backends.backend_qt5agg
4 import FigureCanvas
5 from matplotlib.figure import Figure
6
7 class plot_class(QWidget):
8     def __init__(self, parent = None):
9         QWidget.__init__(self, parent)
10        self.canvas = FigureCanvas(Figure())
11

```

```
12     vertical_layout = QVBoxLayout()
13     vertical_layout.addWidget(self.canvas)
14
15     self.canvas.axis1 =
16     self.canvas.figure.add_subplot(111)
17     self.canvas.axis1 =
18     self.canvas.figure.subplots_adjust(
19             top=0.936,
20             bottom=0.104,
21             left=0.047,
22             right=0.981,
23             hspace=0.2,
24             wspace=0.2
25         )
26
27     self.canvas.figure.set_facecolor("xkcd:light blue")
28     self.setLayout(vertical_layout)
```

The code snippet introduces a specialized PyQt5 widget, `plot_class`, designed to seamlessly incorporate interactive matplotlib plots into PyQt5 applications. This widget encapsulates the functionality required for creating and displaying graphical plots within a user interface.

To begin, the `plot_class` inherits from the PyQt5 `QWidget` class, establishing a foundation for GUI component creation. This inheritance facilitates the integration of the plot widget into larger PyQt5 applications effortlessly.

Inside the `__init__()` method of the `plot_class`, the constructor is meticulously crafted to initialize the widget correctly. The parent `QWidget` class is instantiated, ensuring proper inheritance and compatibility within the PyQt5 framework. A pivotal element, the `FigureCanvas`, is created, serving as a designated area to host the matplotlib plot.

For optimal layout and organization, a vertical box layout (`QVBoxLayout`) is established. This layout aligns and

manages the placement of UI elements. The previously created FigureCanvas is added to this layout using the addWidget method. This ensures that the canvas, which will house the graphical plot, becomes an integral part of the overall widget layout.

The heart of the widget's functionality lies in configuring the main subplot of the matplotlib plot. Referred to as axis1, this subplot is positioned within the canvas using the subplots_adjust method. The method's parameters meticulously define various aspects of layout, such as top, bottom, left, and right margins, as well as horizontal and vertical spacing. This precision enables fine-tuning of the subplot's placement within the canvas.

Moreover, a distinctive visual touch is added by altering the background color of the entire plot. This is achieved by invoking the set_facecolor method on the figure, which encapsulates the entire graphical plot. The chosen color, a soothing "xkcd:light blue," enhances the aesthetics of the widget and harmonizes its appearance with the broader UI.

In conclusion, the plot_class widget's intricate design and meticulous configuration underscore its purpose as a versatile tool for seamlessly embedding interactive matplotlib plots into PyQt5-based applications. Its architecture is geared towards offering developers an intuitive and reusable means of visualizing data and enhancing the user experience within the PyQt5 framework.

- Step 8 Add a **Combo Box** widget and set its **objectName** property as **cbData**. Let it empty. You will populate it from the code.
- Step 9 Add another **Combo Box** widget and set its **objectName** property as **cbClassifier**. Populate this widget with fourteen items as shown in Figure 118.

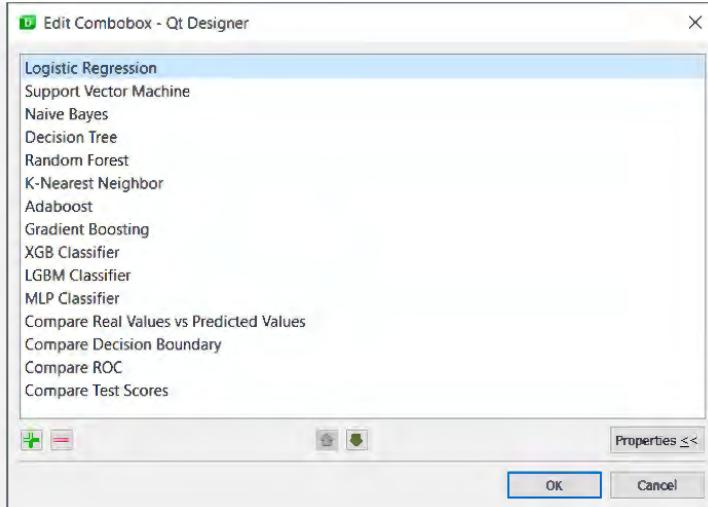


Figure 118 Populating **cbClassifier** widget with fourteen items

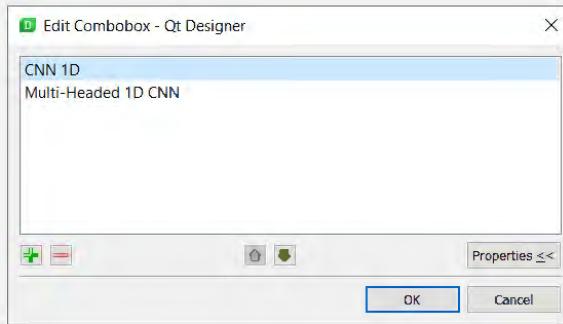


Figure 119 Populating **cbPredictionDL** widget with two items

- | | |
|------------|--|
| Step
10 | Add three radio buttons and set their text properties as Raw , Norm , and Stand . Then, set their objectName as rbRaw , rbNorm , and rbStand . |
| Step
11 | Add the last Combo Box widget and set its objectName property as cbPredictionDL . Populate this widget with two items as shown in Figure 119. |

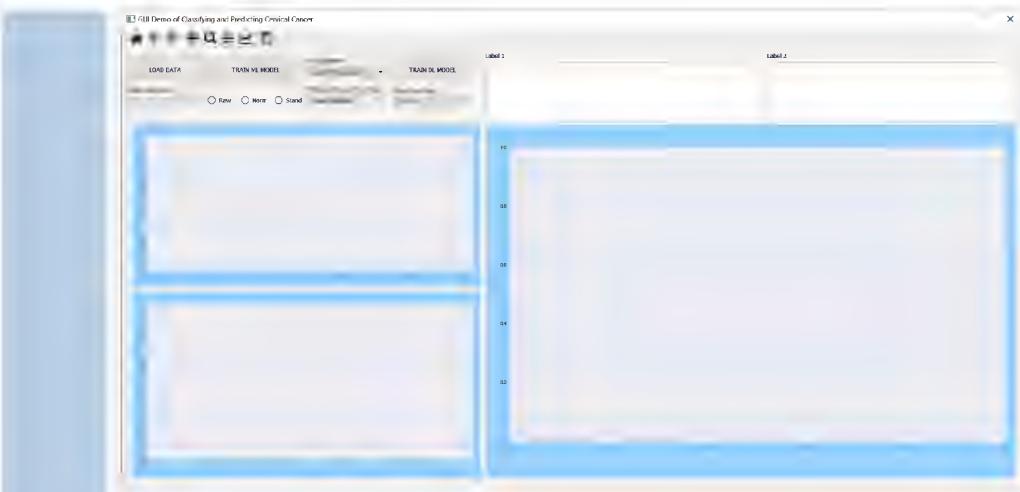


Figure 120 The form when it first runs

Step
12

Write this Python script and save it as **gui_cervical.py**:

```
1 #gui_cervical.py
2 from PyQt5.QtWidgets import *
3 from PyQt5.uic import loadUi
4 from matplotlib.backends.backend_qt5agg
5 import (NavigationToolbar2QT as
6 NavigationToolbar)
7 from matplotlib.colors import ListedColormap
8
9 class DemoGUI_Cervical(QMainWindow):
10     def __init__(self):
11         QMainWindow.__init__(self)
12         loadUi("gui_cervical.ui",self)
13         self.setWindowTitle(\n            "GUI Demo of Classifying and Predicting Cervical\n            Cancer")
14         self.addToolBar(NavigationToolbar(\n            self.widgetPlot1.canvas, self))
15
16
17
18
19 if __name__ == '__main__':
20     import sys
21     app = QApplication(sys.argv)
22     ex = DemoGUI_Cervical()
23     ex.show()
```

```
    sys.exit(app.exec_())
```

The code snippet introduces a PyQt5-based graphical user interface (GUI) application for demonstrating the classification and prediction of cervical cancer. The application leverages both PyQt5 and matplotlib for creating a user-friendly interface with interactive plotting capabilities.

The DemoGUI_Cervical class is the main component of the application. It inherits from the QMainWindow class, which forms the foundation for creating a top-level application window. The `__init__()` method of this class is responsible for setting up the GUI components and initializing the interface.

Upon instantiation, the `loadUi()` function is used to load the design of the graphical user interface from the "gui_cervical.ui" file. This file likely contains the layout and design elements of the interface, including buttons, labels, and potentially input fields, which are not present in the provided code snippet.

The title of the application window is set using the `setWindowTitle` method to indicate the purpose of the GUI, which is focused on classifying and predicting cervical cancer.

An important addition is the integration of a matplotlib navigation toolbar. This toolbar enhances the plotting capabilities of the application by providing interactive features such as zooming and panning. It is added to the GUI using the `addToolBar` method, and the NavigationToolbar is linked to a specific canvas (`self.widgetPlot1.canvas`), presumably defined within the loaded UI design.

The `if __name__ == '__main__':` block ensures that the code is executed only if the script is run directly (not imported as a module). It initializes the PyQt5 application and creates an

instance of the DemoGUI_Cervical class. The GUI window is then displayed using the show method, and the app.exec_() call initiates the event loop, allowing the application to respond to user interactions. The program exits gracefully when the event loop is terminated.

In essence, the code establishes a foundation for a PyQt5-based GUI application focused on cervical cancer classification and prediction, with the added functionality of interactive plotting through matplotlib. It serves as a starting point for developers to build a comprehensive interface for cervical cancer analysis and visualization.

- Step 13 Run **gui_cervical.py** and click **LOAD DATA** button. You will see form's layout as shown in Figure 120.

Reading Dataset and Preprocessing

- Step 1 Import all necessary modules:

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 sns.set_style('darkgrid')
6 import warnings
7 import mglearn
8 warnings.filterwarnings('ignore')
9 import os
10 import joblib
11 from numpy import save
12 from numpy import load
13 from os import path
14 from sklearn.metrics import
15 roc_auc_score,roc_curve
16 from sklearn.model_selection import
17 train_test_split, RandomizedSearchCV,
```

```
18 GridSearchCV,StratifiedKFold
19 from sklearn.preprocessing import StandardScaler,
20 MinMaxScaler
21 from sklearn.linear_model import
22 LogisticRegression
23 from sklearn.naive_bayes import GaussianNB
24 from sklearn.tree import DecisionTreeClassifier
25 from sklearn.svm import SVC
26 from sklearn.ensemble import
27 RandomForestClassifier, ExtraTreesClassifier
28 from sklearn.neighbors import
29 KNeighborsClassifier
30 from sklearn.ensemble import AdaBoostClassifier,
31 GradientBoostingClassifier
32 from xgboost import XGBClassifier
33 from sklearn.neural_network import
34 MLPClassifier
35 from sklearn.linear_model import SGDClassifier
36 from sklearn.preprocessing import StandardScaler,
37 LabelEncoder, OneHotEncoder
38 from sklearn.metrics import confusion_matrix,
39 accuracy_score, recall_score, precision_score
40 from sklearn.metrics import classification_report,
41 f1_score, plot_confusion_matrix
42 from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import
learning_curve
from mlxtend.plotting import
plot_decision_regions
import tensorflow as tf
from sklearn.base import clone
from sklearn.decomposition import PCA
from tensorflow.keras.models import Sequential,
Model, load_model
```

Step 2 Define **write_df_to_qtable()** and **populate_table()** methods to populate any table widget with some data:

```
1 # Takes a df and writes it to a qtable provided. df
2 headers become qtable headers
3 @staticmethod
4 def write_df_to_qtable(df,table):
5     headers = list(df)
6     table.setRowCount(df.shape[0])
7     table.setColumnCount(df.shape[1])
8     table.setHorizontalHeaderLabels(headers)
9
10    # getting data from df is computationally costly so
11    convert it to array first
12    df_array = df.values
13    for row in range(df.shape[0]):
14        for col in range(df.shape[1]):
15            table.setItem(row, col, \
16                           QTableWidgetItem(str(df_array[row,col])))
17
18 def populate_table(self,data, table):
19     #Populates two tables
20     self.write_df_to_qtable(data,table)
21
22     table.setAlternatingRowColors(True)
23     table.setStyleSheet(
24         "alternate-background-color: #ffb07c;background-
25             color: #e6daa6;");
```

Let's dive into a more detailed explanation of the methods:

1. `write_df_to_qtable(df, table):`

- This static method is part of a class and serves as a utility for transferring data from a Pandas DataFrame (df) to a Qt Table Widget (table). The primary goal is to efficiently populate the table with

the data from the DataFrame while maintaining the DataFrame's structure and headers.

- The method begins by extracting the column headers (also known as column names) from the DataFrame using the `list(df)` command. These headers will be used to label the columns of the Qt Table Widget.
- It then configures the dimensions of the table to match those of the DataFrame. The number of rows is set to the DataFrame's row count (`.shape[0]`), and the number of columns is set to the DataFrame's column count (`.shape[1]`).
- The column headers obtained earlier are assigned to the Qt Table Widget's horizontal header labels using the `setHorizontalHeaderLabels(headers)` function. This ensures that the table's columns are labeled correctly.
- To optimize efficiency, the DataFrame is converted to a NumPy array (`df_array`). This conversion helps streamline data retrieval during the subsequent loop.
- The method then iterates through each cell of the DataFrame array. For each cell, it creates a QTableWidgetItem containing the corresponding value from the DataFrame array (converted to a string). This QTableWidgetItem is then placed in the appropriate cell position within the Qt Table Widget,

identified by the row and column indices.

2. populate_table(self, data, table):

- This member method is intended to be used within a class and facilitates the process of populating two Qt Table Widgets with data from a Pandas DataFrame (data).
- The method first calls the write_df_to_qtable() method, passing the DataFrame (data) and one of the Qt Table Widgets (table). This results in the DataFrame's data being efficiently written into the specified table.
- The visual appearance of the table is enhanced through two settings. The setAlternatingRowColors(True) call alternates the background color of rows to improve readability and aesthetics. The setStyleSheet(...) call sets a custom stylesheet that defines specific background colors for both alternating and non-alternating rows. This creates a visually appealing and user-friendly display.

In summary, these methods work in tandem to provide an efficient and visually pleasing way to transfer data from a Pandas DataFrame to one or more Qt Table Widgets. The write_df_to_qtable() method handles the actual data transfer, optimizing performance by converting the DataFrame to a NumPy array. The populate_table() method then utilizes this transfer utility and further enhances the table's visual appeal using alternating row colors and a custom stylesheet. When used within a class, these methods streamline the process of

populating tables with data, making it straightforward and visually appealing within a PyQt5-based GUI application.

Step 3 Define **initial_state()** method to disable some widgets when form initially runs:

```
1 def initial_state(self, state):
2     self.pbTrainML.setEnabled(state)
3     self.cbData.setEnabled(state)
4     self.cbClassifier.setEnabled(state)
5     self.cbPredictionML.setEnabled(state)
6     self.cbPredictionDL.setEnabled(state)
7     self.pbTrainDL.setEnabled(state)
```

Step 4 Define **read_dataset()** method to read dataset and remove all null values:

```
1 def read_dataset(self, dir):
2     #Loads csv file
3     df = pd.read_csv(dir)
4
5     #Checks null values
6     df = df.replace('?', np.NaN)
7     print(df.isnull().sum())
8     print('Total number of null values: ', \
9           df.isnull().sum().sum())
10
11    #Drops two columns of STDs
12    df.drop(['STDs: Time since first diagnosis', \
13              'STDs: Time since last \
14              diagnosis'], inplace=True, axis=1)
15
16    numerical_df = ['Age', 'Number of sexual partners', \
17                     ]
```

```

18 'First sexual intercourse','Num of pregnancies', \
19 'Smokes (years)', 'Smokes (packs/year)', \
20 'Hormonal Contraceptives (years)','IUD (years)', \
21 'STDs (number)']

22
23     categorical_df = ['Smokes','Hormonal
24     Contraceptives','IUD',\
25     'STDs','STDs:condylomatosis',\
26     'STDs:cervical condylomatosis',\
27     'STDs:vaginal condylomatosis',\
28     'STDs:vulvo-perineal condylomatosis',
29     'STDs:syphilis',\
30     'STDs:pelvic inflammatory disease', \
31     'STDs:genital herpes', 'STDs:molluscum
32     contagiosum', \
33     'STDs:AIDS','STDs:HIV','STDs:Hepatitis B', \
34     'STDs:HPV', 'STDs: Number of
35     diagnosis','Dx:Cancer', \
36     'Dx:CIN', 'Dx:HPV', 'Dx', 'Hinselmann', \
37     'Schiller','Citology', 'Biopsy']

38
39 #Fills the missing values of numeric data columns
40 with mean of the column data.
41 for feature in numerical_df:
42     print(feature,",",df[feature].apply(pd.to_numeric,\n
43             errors='coerce').mean())
44     feature_mean =
45     round(df[feature].apply(pd.to_numeric,\n
46             errors='coerce').mean(),1)
47     df[feature] = df[feature].fillna(feature_mean)

48 for feature in categorical_df:
49     df[feature] = df[feature].apply(pd.to_numeric,\n
50             errors='coerce').fillna(1.0)

51 return df

```

The code defines a method named `read_dataset` that is responsible for reading and preprocessing a dataset from a CSV file. The method applies various data cleaning and handling techniques to ensure the dataset is ready for analysis. Here's a detailed breakdown of the steps:

1. Load CSV File:

The method reads the CSV file located in the specified directory (`dir`) using the Pandas `pd.read_csv` function. The loaded dataset is stored in a DataFrame called `df`.

2. Check and Handle Missing Values:

The dataset is inspected for missing values. All occurrences of the character '?' are replaced with `NaN` (Not a Number) using the `df.replace` method. Then, the method prints the sum of missing values for each column using `df.isnull().sum()`. Additionally, it calculates and prints the total number of missing values in the entire dataset using `df.isnull().sum().sum()`.

3. Drop Columns:

Two columns related to STDs (sexually transmitted diseases) are dropped from the DataFrame using the `df.drop` method. These columns are removed from the dataset to prepare it for further analysis.

4. Numerical and Categorical Features Lists:

Separate lists of numerical and categorical feature names are defined. Numerical features are those that hold numeric values, while categorical features contain categorical data.

5. Fill Missing Values in Numeric Columns:

For each feature in the list of numerical features, the method calculates the mean of the column data after converting it to numeric values using `.apply(pd.to_numeric, errors='coerce')`. It then rounds the mean to one decimal place and fills missing values in that column with the calculated mean using `.fillna(feature_mean)`.

6. Convert and Fill Missing Values in Categorical Columns:

For each feature in the list of categorical features, the method converts the values to numeric using `.apply(pd.to_numeric, errors='coerce')` and fills any missing values with the value 1.0. This effectively handles missing values in categorical columns by replacing them with a default value.

7. Return Processed DataFrame:

The preprocessed DataFrame `df` is returned, now free of missing values and with appropriate handling of both numerical and categorical features.

In summary, the `read_dataset()` method is a comprehensive data preprocessing function that performs tasks such as handling missing values, converting data types, and preparing a dataset for further analysis. This method ensures the dataset is in a clean and usable state, making it suitable for various data science and machine learning tasks.

Step 5 Define **populate_cbData()** to populate **cbData** widget:

```
1 def populate_cbData(self):
2     self.cbData.addItems(["target (Biopsy)"])
3     self.cbData.addItems(self.df.iloc[:,1:-1])
4     self.cbData.addItems(["Features Importance"])
5     self.cbData.addItems(["Correlation Matrix", \
6     "Pairwise Relationship", "Features Correlation"])
```

The code defines a method named `populate_cbData()` within the class. This method is responsible for populating a combo box (`cbData`) with a list of items. These items include options related to data visualization and analysis. Let's break down the method's functionality:

1. Populate with Specific Items:

The method begins by adding a single item, "target (Biopsy)", to the combo box. This item seems to represent the target variable for some analysis or visualization.

2. Populate with DataFrame Columns:

Next, the method adds items to the combo box corresponding to the columns of a DataFrame (`self.df`). It appears that the DataFrame is assumed to be stored as an attribute within the class. The columns included for selection start from the second column (`[:,1:-1]`), indicating that the first column might be excluded, possibly because it contains labels or identifiers. Each column's name is added as an individual item in the combo box.

3. Populate with Additional Items:

The method then adds a set of predefined items to the combo box. These items are relating to specific types of data visualization or analysis:

- "Features Importance": an option to analyze or visualize the importance of different features.
- "Correlation Matrix": an option to display the correlation matrix between features.
- "Pairwise Relationship": an option for exploring pairwise relationships between features.
- "Features Correlation": an option to analyze the correlation between different features.

Overall, the `populate_cbData()` method is designed to conveniently populate a combo box with a set of items that can be selected for data analysis or visualization. The provided items cover a range of potential analyses, including examining feature correlations, pairwise relationships, and features' importance in relation to a target variable. This method is a part of a larger class and could be used within a graphical user interface (GUI) application to provide users with options for exploring and analyzing the dataset.

- Step 6 Define **import_dataset()** method to import dataset for machine learning algorithms (**df**) and populate two table widgets with data and its description:

```
1 def import_dataset(self):
2     curr_path = os.getcwd()
3     dataset_dir = curr_path + \
4         "/kag_risk_factors_cervical_cancer.csv"
5     self.EPOCHS = 250
6     self.BATCH_SIZE = 32
7
8     #Loads csv file
9     self.df = self.read_dataset(dataset_dir)
10
11    #Populates tables with data
12    self.populate_table(self.df, self.twData1)
13    self.label1.setText('Cervical Cancer Data')
14
15    self.populate_table(self.df.iloc[:,1:].describe(), \
16        self.twData2)
17    self.twData2.setVerticalHeaderLabels(['Count', \
18        'Mean', 'Std', 'Min', '25%', '50%', '75%', 'Max'])
19    self.label2.setText('Data Description')
20
21    #Turns on pbTrainML widget
22    self.pbTrainML.setEnabled(True)
23    self.pbTrainDL.setEnabled(True)
24
25    #Turns off pbLoad
26    self.pbLoad.setEnabled(False)
27
28    #Populates cbData
29    self.populate_cbData()
```

The code defines a method named **import_dataset()** within the class. This method is responsible for importing a dataset, populating tables with data, configuring labels and buttons,

and preparing the user interface for further interactions. Let's break down the method's functionality step by step:

1. Set Epochs and Batch Size:

The EPOCHS and BATCH_SIZE class attributes are set to predefined values (250 and 32, respectively). These attributes might be used later for training machine learning or deep learning models.

2. Determine Dataset Path:

The current working directory is obtained using os.getcwd(), and the dataset's full path is constructed by appending the dataset's filename ("kag_risk_factors_cervical_cancer.csv") to the current working directory.

3. Load Dataset:

The dataset is loaded using the read_dataset() method, and the resulting DataFrame is assigned to the class attribute self.df.

4. Populate Table 1:

The method populates the first table (twData1) with the loaded dataset using the populate_table() method. The label above this table is updated to indicate that it contains "Cervical Cancer Data."

5. Populate Table 2:

The second table (twData2) is populated with summary statistics (describe) of the dataset's columns (excluding the first column). The table's vertical header labels are set to represent count, mean, standard deviation, minimum, 25th percentile, median (50th percentile), 75th percentile, and maximum.

6. Update Label 2:

The label above the second table is updated to indicate that it contains "Data Description."

7. Enable Training Buttons:

The "Train ML Model" and "Train DL Model" buttons (pbTrainML and pbTrainDL) are enabled, allowing the user to proceed with training machine learning or deep learning models.

8. Disable Load Button:

The "Load" button (pbLoad) is disabled to prevent redundant data loading.

9. Populate Combo Box:

The populate_cbData() method is called to populate the combo box (cbData) with a set of items for data analysis or visualization.

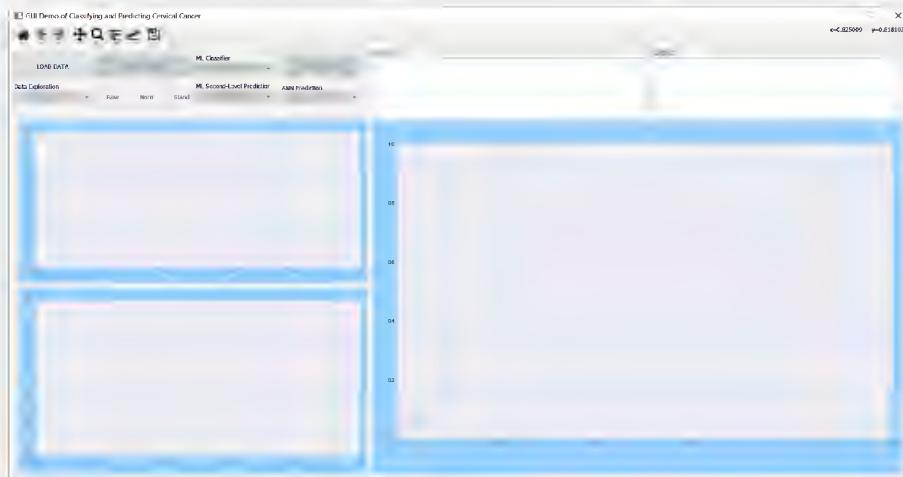


Figure 121 The initial state of form

Overall, the import_dataset() method is responsible for initializing the user interface after importing the dataset. It loads the dataset, populates tables with data and summary statistics, configures labels, enables training buttons, and prepares the combo box for further interactions. This method effectively sets the stage for subsequent data analysis and model training activities within the application.

Step 7 Connect **clicked()** event of **pbLoad** widget with **import_dataset()** and put it inside **__init__()** method as shown in line 8 and invoke **initial_state()** methode in line 9:

```
1 def __init__(self):
2     QMainWindow.__init__(self)
3     loadUi("gui_cervical.ui",self)
4     self.setWindowTitle(
5         "GUI Demo of Classifying and Predicting Cervical
6         Cancer")
```

```

7 self.addToolBar(NavigationToolbar(
8     self.widgetPlot1.canvas, self))
9 self.pbLoad.clicked.connect(self.import_dataset)
self.initial_state(False)

```

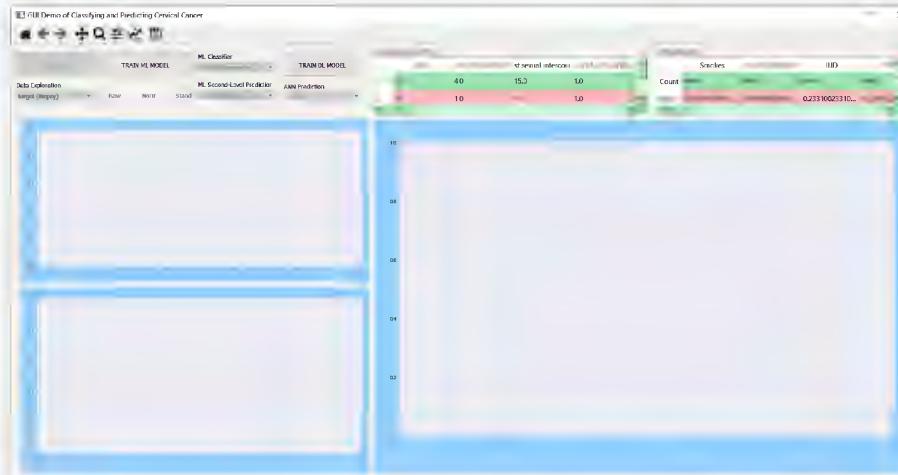


Figure 122 When **LOAD DATA** button is clicked, the two tables will be populated

- Step 8 Run **gui_cervical.py** and you will see the other widgets are initially disabled as shown in Figure 121. Then click **LOAD DATA** button. The two tables will be populated as shown in Figure 122.

Resampling and Splitting Data

- Step 1 Define **fit_dataset()** method to resample data using SMOTE:

```

1 def fit_dataset(self, df):
2     X = df.drop('Biopsy', axis
3 =1).apply(pd.to_numeric, \
4         errors='coerce').astype('float64')
5     y = df["Biopsy"]
6     sm = SMOTE(random_state=42)
7     X,y = sm.fit_resample(X, y.ravel())

```

```
return X, y
```

The code defines a method named `fit_dataset()` within the class. This method is intended for preprocessing and transforming a dataset to prepare it for machine learning model training, specifically addressing class imbalance using the Synthetic Minority Over-sampling Technique (SMOTE). Let's go through the steps in detail:

1. Input:

The method takes a `DataFrame df` as an input parameter.

2. Feature and Target Extraction:

The method first prepares the feature matrix `X` and the target vector `y` for model training. It drops the "Biopsy" column (target variable) from the `DataFrame` using `df.drop('Biopsy', axis=1)`. The `apply` function with `pd.to_numeric` is applied to convert any non-numeric values to `NaN`, and then `.astype('float64')` converts the entire `DataFrame` to a numeric data type (`float64`).

3. Resampling using SMOTE:

The Synthetic Minority Over-sampling Technique (SMOTE) is employed to address class imbalance. SMOTE generates synthetic samples for the minority class (in this case, positive cases of "Biopsy") by interpolating between existing samples. The SMOTE class is initialized with a random state of 42 (`random_state=42`), which ensures reproducibility.

4. The `fit_resample()` method of the SMOTE instance is called, passing `X` and `y` as arguments. This resampling process balances the class distribution by oversampling the minority class ("Biopsy" positive cases) to achieve a balanced ratio with the majority class.

5. Output:

The method returns the resampled feature matrix X and the corresponding target vector y.

In summary, the fit_dataset() method is designed to preprocess and transform a dataset by dropping the target variable, converting non-numeric values to NaN, and then applying SMOTE to address class imbalance. The method is part of a broader data preprocessing and model training pipeline, preparing the data for further analysis using machine learning algorithms.

- Step 2 Define **train_test()** to split dataset into train and test data with raw, normalized, and standardized feature scaling:

```
1 def train_test(self):
2     X, y = self.fit_dataset(self.df)
3
4     #Splits the data into training and testing
5     X_train, X_test, y_train, y_test =
6     train_test_split(X, y,\n7         test_size = 0.2, random_state = 2021,
8         stratify=y)
9     self.X_train_raw = X_train.copy()
10    self.X_test_raw = X_test.copy()
11    self.y_train_raw = y_train.copy()
12    self.y_test_raw = y_test.copy()
13
14    #Saves into npy files
15    save('X_train_raw.npy', self.X_train_raw)
16    save('y_train_raw.npy', self.y_train_raw)
17    save('X_test_raw.npy', self.X_test_raw)
18    save('y_test_raw.npy', self.y_test_raw)
19
20    self.X_train_norm = X_train.copy()
21    self.X_test_norm = X_test.copy()
22    self.y_train_norm = y_train.copy()
23    self.y_test_norm = y_test.copy()
24    norm = MinMaxScaler()
```

```

25     self.X_train_norm[inf_cols] = \
26         norm.fit_transform(self.X_train_norm)
27     self.X_test_norm[inf_cols] = \
28         norm.transform(self.X_test_norm)
29
30     #Saves into npy files
31     save('X_train_norm.npy', self.X_train_norm)
32     save('y_train_norm.npy', self.y_train_norm)
33     save('X_test_norm.npy', self.X_test_norm)
34     save('y_test_norm.npy', self.y_test_norm)
35
36     self.X_train_stand = X_train.copy()
37     self.X_test_stand = X_test.copy()
38     self.y_train_stand = y_train.copy()
39     self.y_test_stand = y_test.copy()
40     scaler = StandardScaler()
41     self.X_train_stand[inf_cols] = \
42         scaler.fit_transform(self.X_train_stand)
43     self.X_test_stand[inf_cols] = \
44         scaler.transform(self.X_test_stand)
45
46     #Saves into npy files
47     save('X_train_stand.npy', self.X_train_stand)
48     save('y_train_stand.npy', self.y_train_stand)

```

The code defines a method named `train_test()` within the class. This method is responsible for preprocessing the dataset, splitting it into training and testing sets, and saving the resulting data in various formats for future use. It performs three types of preprocessing: raw data, normalized data, and standardized data. Let's break down the method step by step:

1. Preprocessing and Splitting:

The method begins by calling the `fit_dataset()` method to preprocess the dataset. The resulting feature matrix `X` and target vector `y` are then split into training and

testing sets using the `train_test_split` function from `sklearn.model_selection`. The `test_size` parameter specifies the proportion of data allocated to the test set (20% in this case), and the `stratify` parameter ensures that the class distribution is preserved in the splits.

2. Saving Raw Data:

The raw training and testing data (both features and targets) are saved as NumPy arrays using the `save` function from the `numpy` library. Separate files are created for raw data with `X_train_raw.npy`, `y_train_raw.npy`, `X_test_raw.npy`, and `y_test_raw.npy`.

3. Normalization:

A copy of the training and testing data is created for normalization. The `MinMaxScaler` from `sklearn.preprocessing` is used to perform feature-wise normalization. The `inf_cols` variable seems to contain a list of column indices that need to be normalized. These columns are selected from the copy of the training and testing data, and their values are normalized using the scaler's `fit_transform` and `transform` methods.

4. Saving Normalized Data:

Similar to the raw data, the normalized training and testing data are saved as NumPy arrays with `X_train_norm.npy`, `y_train_norm.npy`, `X_test_norm.npy`, and `y_test_norm.npy`.

5. Standardization:

A copy of the training and testing data is created again for standardization. The `StandardScaler` from `sklearn.preprocessing` is used to perform feature-wise standardization. The columns listed in `inf_cols` are selected, and their values are standardized using the scaler's `fit_transform` and `transform` methods.

6. Saving Standardized Data:

Similar to previous steps, the standardized training and testing data are saved as NumPy arrays with `X_train_stand.npy`, `y_train_stand.npy`, `X_test_stand.npy`, and `y_test_stand.npy`.

In summary, the `train_test()` method performs preprocessing on the dataset, including splitting it into training and testing sets and applying raw data, normalization, and standardization transformations. It then saves the resulting data in multiple formats for future use in machine learning model training and evaluation. The method effectively prepares the data for various experiments and analyses within the broader context of a machine learning application.

Step 3 Define `split_data_ML()` method execute splitting dataset into train and test data:

```
1  def split_data_ML(self):
2      if path.isfile('X_train_raw.npy'):
3          #Loads npy files
4          self.X_train_raw = \
5
6          np.load('X_train_raw.npy',allow_pickle=True)
7          self.y_train_raw = \
8              np.load('y_train_raw.npy',allow_pickle=True)
9          self.X_test_raw = \
10             np.load('X_test_raw.npy',allow_pickle=True)
11          self.y_test_raw = \
12              np.load('y_test_raw.npy',allow_pickle=True)
13
14          self.X_train_norm = \
15
16          np.load('X_train_norm.npy',allow_pickle=True)
17          self.y_train_norm = \
18
19          np.load('y_train_norm.npy',allow_pickle=True)
20          self.X_test_norm = \
21
22          np.load('X_test_norm.npy',allow_pickle=True)
23          self.y_test_norm = \
24
25          np.load('y_test_norm.npy',allow_pickle=True)
26
```

```
27 self.X_train_stand = \
28
29 np.load('X_train_stand.npy',allow_pickle=True)
30 self.y_train_stand = \
31
32 np.load('y_train_stand.npy',allow_pickle=True)
33 self.X_test_stand = \
34
35 np.load('X_test_stand.npy',allow_pickle=True)
36 self.y_test_stand = \
37
38 np.load('y_test_stand.npy',allow_pickle=True)
39
40 else:
41 self.train_test()
42
43 #Prints each shape
44 print('X train raw shape: ', self.X_train_raw.shape)
45 print('Y train raw shape: ', self.y_train_raw.shape)
46 print('X test raw shape: ', self.X_test_raw.shape)
47 print('Y test raw shape: ', self.y_test_raw.shape)
48
49 #Prints each shape
50 print('X train norm shape: ',
self.X_train_norm.shape)
print('Y train norm shape: ',
self.y_train_norm.shape)
print('X test norm shape: ', self.X_test_norm.shape)
print('Y test norm shape: ', self.y_test_norm.shape)
51
52 #Prints each shape
53 print('X train stand shape: ',
self.X_train_stand.shape)
print('Y train stand shape: ',
self.y_train_stand.shape)
print('X test stand shape: ', self.X_test_stand.shape)
print('Y test stand shape: ', self.y_test_stand.shape)
```

The code defines a method named `split_data_ML()` within the class. This method handles the loading of preprocessed data from NumPy files, performs a check for existing files, and prints the shapes of the loaded data arrays. The method seems to be part of a data preparation workflow for machine learning. Let's break down the method step by step:

1. Check for Existing Files:

The method first checks if the NumPy files for various data splits (raw, normalized, standardized) exist in the current directory using the `path.isfile` function from the `os.path` module.

2. Load Data from NumPy Files:

If the files exist, the method loads the data from the NumPy files using the `np.load` function. The data arrays are loaded into respective class attributes such as `self.X_train_raw`, `self.y_train_raw`, etc.

3. Data Preparation if Files Don't Exist:

If the files do not exist, the method calls the `train_test` method (not shown) to preprocess and split the dataset. This ensures that the required data files are generated before attempting to load them.

4. Print Data Shapes:

After loading or generating the data, the method prints the shapes of each data split to provide insight into the sizes of the various arrays. The shapes are printed for raw, normalized, and standardized data arrays for both training and testing sets.

In summary, the `split_data_ML()` method is responsible for managing the loading and preparation of preprocessed data arrays from NumPy files. It checks for the existence of the files, loads the data if available, generates the data if files are missing, and then prints the shapes of the loaded/generated data arrays. This method streamlines the process of handling and accessing the prepared data for machine learning tasks, contributing to a more organized and efficient workflow.

Step Define **train_model_ML()** method to invoke

4 **split_data_ML()** method:

```
1  def train_model_ML(self):
2      self.split_data_ML()
3
4      #Turns on three widgets
5      self.cbData.setEnabled(True)
6      self.cbClassifier.setEnabled(True)
7      self.cbPredictionML.setEnabled(True)
8
9      #Turns off pbTrainML
10     self.pbTrainML.setEnabled(False)
```

The code defines a method named `train_model_ML()` within the class. This method is responsible for preparing the data, enabling specific widgets, and disabling a button as part of the process of training a machine learning model. Let's break down the method's functionality:

1. Data Splitting:

The method begins by calling the `split_data_ML()` method, which is responsible for loading or generating the preprocessed data arrays for machine learning. This ensures that the required data is available before training the model.

2. Enable Widgets:

The method proceeds to enable three specific widgets: `cbData`, `cbClassifier`, and `cbPredictionML`. Enabling these widgets makes them interactive and available for user selection or input.

3. Disable Training Button:

The "Train ML" button (`pbTrainML`) is disabled using the `setEnabled` method. This button is turned off to prevent users from initiating model training multiple times or unnecessarily.

In summary, the `train_model_ML()` method is designed to facilitate the preparation of data for machine learning, enable relevant widgets for user interaction, and disable the "Train

ML" button to prevent redundant training. This method is part of a larger workflow within a graphical user interface (GUI) application, guiding users through the process of data preparation and model training in a controlled manner.

- Step 5 Connect `clicked()` event of `pbTrainML` widget with `train_model_ML()` and put it inside `__init__()` method as shown in line 10:

```
1 def __init__(self):
2     QMainWindow.__init__(self)
3     loadUi("gui_cervical.ui",self)
4     self.setWindowTitle(
5         "GUI Demo of Classifying and Predicting Cervical
6         Cancer")
7     self.addToolBar(NavigationToolbar(
8         self.widgetPlot1.canvas, self))
9     self.pbLoad.clicked.connect(self.import_dataset)
10    self.initial_state(False)
11    self.pbTrainML.clicked.connect(self.train_model_ML
12 )
```

- Step 6 Run `gui_cervical.py` and you will see the other widgets are initially disabled. Click **LOAD DATA** button. The two tables are populated, **LOAD DATA** button is disabled, and **TRAIN ML MODEL** and **TRAIN DL MODEL** buttons are enabled. Then, click on **TRAIN ML MODEL** button. You will see that **cbData**, **cbClassifier**, and **cbPredictionML** are enabled and **pbTrainML** is disabled as shown in Figure 123. You also will find four training dan test npy files for machine learning in your working directory.



Figure 123 The **cbData**, **cbClassifier**, and **cbPredictionML** widgets are enabled user clicks **TRAIN ML MODEL** button

Distribution of Features

Step 1 Define **dist_target()** and **dist_target_versus_features()** methods to plot distribution of target variables versus other features:

```

1 def dist_target(self, df_target, var_target, labels,
2     widget):
3     df_target.value_counts().plot.pie(\n        ax =
5     widget.canvas.axis1, labels=labels, startangle=40,\n        explode=[0,0.15], shadow=True,\n        colors=['#ff6666', '#F5C7B8FF'], autopct =
8     '%1.1f%%',\n        textprops={'fontsize': 14})\n    widget.canvas.axis1.set_title(\n11     'The distribution of target variable ('+ var_target+')', \
12         fontweight ="bold", fontsize=14)\n13     widget.canvas.figure.tight_layout()\n14     widget.canvas.draw()
15
16 def
17     dist_target_versus_features(self, df_target, var_target,
```

```
18 labels):
19 #Plots distribution of target variable in pie chart
20 self.widgetPlot1.canvas.figure.clf()
21 self.widgetPlot1.canvas.axis1 = \
22 self.widgetPlot1.canvas.figure.add_subplot(111, \
23 facecolor = '#fbe7dd')
24 self.dist_target(df_target,var_target, labels, \
25 self.widgetPlot1)
26
27 #Plots distribution: Hormonal Contraceptives (years)
28 versus var_target
29 self.widgetPlot2.canvas.figure.clf()
30 self.widgetPlot2.canvas.axis1 = \
31 self.widgetPlot2.canvas.figure.add_subplot(211, \
32 facecolor = '#fbe7dd')
33 self.widgetPlot2.canvas.axis1.set_title(\n
34 'Distribution: Hormonal Contraceptives (years) versus\n'
35 '\n
36 + var_target, fontweight ="bold",fontsize=16)
37 sns.histplot(data=self.df, \
38 x=self.df["Hormonal Contraceptives\n
39 (years)"].apply(\n
40 pd.to_numeric, errors='coerce').astype('float64'), \
41 ax= self.widgetPlot2.canvas.axis1,zorder=2,\n
42 kde=False,hue=df_target,multiple="stack", \
43 shrink=.8,linewidth=0.3,alpha=1)
44 self.widgetPlot2.canvas.axis1.set_xlabel(\n
45 "Hormonal Contraceptives (years)",fontweight
46 ="bold",\n
47 fontsize=16)
48
49 self.widgetPlot2.canvas.axis1 = \
50 self.widgetPlot2.canvas.figure.add_subplot(212, \
51 facecolor = '#fbe7dd')
52 self.widgetPlot2.canvas.axis1.set_title(\n
53 'Distribution: IUD (years) versus ' + var_target, \
54 fontweight ="bold",fontsize=16)
55 sns.histplot(data=self.df, x=self.df["IUD\n
56 (years)"].apply(\n
```

```
57     pd.to_numeric, errors='coerce').astype('float64'),\
58     ax=
59     self.widgetPlot2.canvas.axis1,zorder=2,kde=False,\n
60         hue=df_target,multiple="stack",shrink=.8,\n
61             linewidth=0.3,alpha=1)
62     self.widgetPlot2.canvas.axis1.set_xlabel("IUD\n(years"),\
63         fontweight ="bold",fontsize=16)
64     self.widgetPlot2.canvas.figure.tight_layout()
65     self.widgetPlot2.canvas.draw()
66
67
68 #Plots distribution: Age versus var_target
69 self.widgetPlot3.canvas.figure.clf()
70 self.widgetPlot3.canvas.axis1 =
71 self.widgetPlot3.canvas.figure.add_subplot(221,\n
72     facecolor = '#fbe7dd')
73 self.widgetPlot3.canvas.axis1.set_title(\n
74 'Distribution: Age versus ' + var_target, \
75     fontweight ="bold",fontsize=16)
76 sns.histplot(data=self.df, x=self.df.Age,\n
77     ax= self.widgetPlot3.canvas.axis1,zorder=2,\n
78         kde=False,hue=df_target,multiple="stack", \
79             shrink=.8,linewidth=0.3,alpha=1)
80 self.widgetPlot3.canvas.axis1.set_xlabel("Age",\
81     fontweight ="bold",fontsize=16)
82
83 #Plots distribution: First sexual intercourse versus
84 var_target
85 self.widgetPlot3.canvas.axis1 =
86 self.widgetPlot3.canvas.figure.add_subplot(222,\n
87     facecolor = '#fbe7dd')
88 self.widgetPlot3.canvas.axis1.set_title(\n
89 'Distribution: First sexual intercourse versus ' + \
90     var_target, fontweight ="bold",fontsize=16)
91 sns.histplot(data=self.df, \
92     x=self.df["First sexual intercourse"].apply(\n
93         pd.to_numeric, errors='coerce').astype('float64'),\n
94         ax= self.widgetPlot3.canvas.axis1,zorder=2,\n
95             kde=False,hue=df_target,multiple="stack", \
```

```
96     shrink=.8,linewidth=0.3,alpha=1)
97 self.widgetPlot3.canvas.axis1.set_xlabel(
98 "First sexual intercourse",fontweight
99 ="bold",fontsize=16)
100
101 #Plots distribution: Num of pregnancies versus
102 var_target
103 self.widgetPlot3.canvas.axis1 = \
104 self.widgetPlot3.canvas.figure.add_subplot(223, \
105      facecolor = '#fbe7dd')
106 self.widgetPlot3.canvas.axis1.set_title(\
107 'Distribution: Num of pregnancies versus ' + \
108      var_target, fontweight ="bold",fontsize=16)
109 sns.histplot(data=self.df, \
110      x=self.df["Num of pregnancies"].apply(\
111      pd.to_numeric, errors='coerce').astype('float64'), \
112      ax= self.widgetPlot3.canvas.axis1,zorder=2, \
113      kde=False,hue=df_target,multiple="stack", \
114      shrink=.8,linewidth=0.3,alpha=1)
115 self.widgetPlot3.canvas.axis1.set_xlabel(\
116 "Num of pregnancies",fontweight
117 ="bold",fontsize=16)
118
119 #Plots distribution: Hormonal Contraceptives (years)
120 versus var_target
121 self.widgetPlot3.canvas.axis1 = \
122 self.widgetPlot3.canvas.figure.add_subplot(224, \
123      facecolor = '#fbe7dd')
124 self.widgetPlot3.canvas.axis1.set_title(\
125 'Distribution: Hormonal Contraceptives (years) versus
126 ' \
127      + var_target, fontweight ="bold",fontsize=16)
128 sns.histplot(data=self.df, \
129      x=self.df["Hormonal Contraceptives
130 (years)"].apply(\
131      pd.to_numeric, errors='coerce').astype('float64'), \
132      ax= self.widgetPlot3.canvas.axis1,zorder=2, \
133      kde=False,hue=df_target,multiple="stack", \
134      shrink=.8,linewidth=0.3,alpha=1)
```

```
self.widgetPlot3.canvas.axis1.set_xlabel(\n    "Hormonal Contraceptives (years)",\n    fontweight ="bold",fontsize=16)\n\n    self.widgetPlot3.canvas.figure.tight_layout()\n    self.widgetPlot3.canvas.draw()
```

The code defines two methods within the class that are involved in visualizing the distribution of a target variable along with selected features. The methods create pie charts and histograms to illustrate these distributions using the seaborn and matplotlib libraries. Let's break down the functionality of each method:

1. dist_target() Method:

This method is responsible for plotting a pie chart that visualizes the distribution of a target variable and displaying it in a designated widget (a FigureCanvas). The method accepts the following parameters:

- df_target: A pandas Series representing the target variable.
- var_target: The name of the target variable.
- labels: Labels for the pie chart segments.
- widget: The widget where the pie chart will be displayed.

The method uses the value_counts() function to calculate the distribution of the target variable and then plots a pie chart with the specified labels, start angle, colors, and other settings. The title is set for the pie chart, and the figure is drawn on the canvas.

2. dist_target_versus_features() Method:

This method is responsible for creating a set of histograms that illustrate the distribution of selected features against the target variable. It seems to display these histograms in a three-by-two grid layout using

three different subplots. Here's a breakdown of the steps within this method:

- The method first clears the existing figure and sets up subplot axes for plotting.
- It then calls the `dist_target()` method to plot the distribution of the target variable as a pie chart in the first subplot.
- Following that, the method creates histograms for three different features (`Age`, `First sexual intercourse`, and `Num of pregnancies`) against the target variable. It uses the `sns.histplot` function to create the histograms, specifying various parameters such as `data`, `x-axis`, `axes`, and visualization options. The histograms are stacked based on `hue` (the target variable) to compare distributions for different target classes.
- The final subplot in the grid displays the distribution of Hormonal Contraceptives (years) against the target variable, similar to the previous feature distributions.
- Finally, the layout is adjusted, and the figure is drawn on the canvas.

In summary, these methods are designed to create informative visualizations that show the distribution of a target variable and the relationship between selected features and the target variable. The visualizations are generated using pie charts and histograms, and the methods are intended to enhance data exploration and analysis within a graphical user interface (GUI) application.

- Step 2 Define **choose_plot()** to read **currentText** property of **cbData** widget and act accordingly:

```
1 def choose_plot(self):
2     strCB = self.cbData.currentText()
3
4     if strCB == 'target (Biopsy)':
5         self.dist_target_versus_features(self.df.Biopsy, \
6             "Biopsy", ['Biopsy=0','Biopsy=1'])
7
8     if strCB == 'Hinselmann':
9         self.dist_target_versus_features(self.df.Hinselmann, \
10
11
12
13
14
15
16
17
18
```

```
"Hinselmann", ['Hinselmann=0','Hinselmann=1'])
```

```
if strCB == 'Schiller':  
    self.dist_target_versus_features(self.df.Schiller, \  
    "Schiller", ['Schiller=0','Schiller=1'])  
  
if strCB == 'Citology':  
    self.dist_target_versus_features(self.df.Citology, \  
    "Citology", ['Citology=0','Citology=1'])
```

The code defines a method named `choose_plot()` within the class. This method is designed to generate specific visualizations based on user-selected options from a combo box (`cbData`). The method determines the selected option and calls the `dist_target_versus_features()` method to create corresponding visualizations. Let's break down the functionality of the `choose_plot()` method:

1. Get Current Combo Box Selection:

The method begins by retrieving the currently selected text from the combo box (`cbData`) using the `currentText` method. The selected text is stored in the variable `strCB`.

2. Visualization Based on Selection:

The method uses a series of conditional statements (`if` statements) to determine the selected option and generate a corresponding visualization:

- If the selected option is 'target (Biopsy)', the method calls `dist_target_versus_features` with the target variable `self.df.Biopsy`, the label "Biopsy", and labels for the pie chart segments `['Biopsy=0', 'Biopsy=1']`.
- If the selected option is 'Hinselmann', 'Schiller', or 'Citology', similar to the previous step, the method calls `dist_target_versus_features()` with the respective target variable and appropriate labels.

3. Visualization Generation:

For each selected option, the corresponding visualization is generated by calling the `dist_target_versus_features()` method with the appropriate arguments.

In summary, the `choose_plot()` method is responsible for generating specific visualizations based on user-selected options from a combo box. Depending on the selection, it triggers the creation of visualizations that display the distribution of target variables or specific features against the target variable. This method is part of a graphical user interface (GUI) application and aims to provide users with

insights into data distributions and relationships through interactive visualizations.

Step 3

Connect **currentIndexChanged()** event of **cbData** **choose_plot()** method and put it inside **__init__** shown in line 11:

```
1 def __init__(self):
2     QMainWindow.__init__(self)
3     loadUi("gui_cervical.ui",self)
4     self.setWindowTitle(
5         "GUI Demo of Classifying and Predicting Ce
6         Cancer")
7     self.addToolBar(NavigationToolbar(
8         self.widgetPlot1.canvas, self))
9     self.pbLoad.clicked.connect(self.import_datal
10    s)
11    self.initial_state(False)
12    self.pbTrainML.clicked.connect(self.train_mo
13    self.cbData.currentIndexChanged.connect(self.
14    )
```

Step 4

Run **gui_cervical.py** and click **LOAD DATA** and **MODEL** buttons. Then, choose **target (Biopsy)** in **cbData** widget. You will see the result as shown in Figure 124.



Figure 124 The distribution of **Biopsy** variable vs other features

Then, choose **Hinselmann** item from **cbData** widget and see the result as shown in Figure 125.



Figure 125 The distribution of **Hinselmann** variable features

Then, choose **Schiller** item from **cbData** widget. You will get the result as shown in Figure 126.

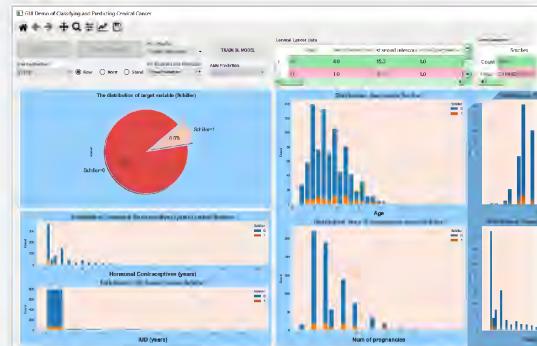


Figure 126 The distribution of **Schiller** variable features

Then, choose **Citology** item from **cbData** widget. You will get the result as shown in Figure 127.



Figure 127 The distribution of **Citology** variable features

Histogram and Density

Step 1 Define **hist_feat_versus_other()**, **prob_feat_versus_other()**, and **dist_feat_versus_others()** methods to plot distribution and

density of other features:

```
1 def
2 hist_feat_versus_other(self,df,feat,another,legend,ax0):
3     background_color = "#fbe7dd"
4     sns.set_palette(['#ff355d','#ffd514'])
5     for s in ["right", "top"]:
6         ax0.spines[s].set_visible(False)
7
8     ax0.set_facecolor(background_color)
9     ax0_sns = sns.histplot(data=df, x=feat,ax=ax0,\n10         zorder=2,kde=False,hue=another,multiple="stack", \
11         shrink=.8,linewidth=0.3,alpha=1)
12
13     ax0_sns.set_xlabel("",fontsize=10, weight='bold')
14     ax0_sns.set_ylabel("",fontsize=10, weight='bold')
15
16     ax0_sns.grid(which='major', axis='x', zorder=0, \
17         color="#EEEEEE", linewidth=0.4)
18     ax0_sns.grid(which='major', axis='y', zorder=0, \
19         color="#EEEEEE", linewidth=0.4)
20
21     ax0_sns.tick_params(labelsize=8, width=0.5,
22 length=1.5)
23     ax0_sns.legend(legend, ncol=2,
24 facecolor="#D8D8D8",\
25     edgecolor=background_color, fontsize=12, \
26     bbox_to_anchor=(1, 0.989), loc='upper right')
27     ax0.set_facecolor(background_color)
28
29 def
30 prob_feat_versus_other(self,df,feat,another,legend,ax0):
31     background_color = "#fbe7dd"
32     sns.set_palette(['#ff355d','#ffd514'])
33     for s in ["right", "top"]:
34         ax0.spines[s].set_visible(False)
35
36     ax0.set_facecolor(background_color)
37
38     ax0_sns = sns.kdeplot(x=feat,ax=ax0,hue=another,\n39         linewidth=0.3,fill=True,cbar='g',zorder=2,\n40         alpha=1,multiple='stack')
41
42     ax0_sns.set_xlabel("",fontsize=10, weight='bold')
43     ax0_sns.set_ylabel("",fontsize=10, weight='bold')
44
45     ax0_sns.grid(which='major', axis='x', \
46         zorder=0, color="#EEEEEE", linewidth=0.4)
47     ax0_sns.grid(which='major', axis='y', zorder=0, \
48         color="#EEEEEE", linewidth=0.4)
49
50     ax0_sns.tick_params(labelsize=8, width=0.5,
51 length=1.5)
```

```

52     ax0_sns.legend(legend, ncol=2,
53 facecolor='#D8D8D8',\
54     edgecolor=background_color, fontsize=12, \
55     bbox_to_anchor=(1, 0.989), loc='upper right')
56 ax0.set_facecolor(background_color)
57
58 def dist_feat_versus_others(self,df_target,title):
59     self.widgetPlot3.canvas.figure.clf()
60     print(self.df.Smokes.value_counts())
61     self.widgetPlot3.canvas.axis1 = \
62     self.widgetPlot3.canvas.figure.add_subplot(231, \
63         facecolor = '#fbe7dd')
64     self.widgetPlot3.canvas.axis1.set_title('Smokes versus ' \
65     + \
66         title, fontweight ="bold",fontsize=16)
67     self.hist_feat_versus_other(self.df,df_target, \
68     self.df.Smokes,['Smokes','No Smokes'], \
69     self.widgetPlot3.canvas.axis1)
70     self.widgetPlot3.canvas.axis1.set_xlabel(title, \
71         fontweight ="bold",fontsize=16)
72
73     print(self.df["Hormonal
74     Contraceptives"].value_counts())
75     self.widgetPlot3.canvas.axis1 = \
76     self.widgetPlot3.canvas.figure.add_subplot(232, \
77         facecolor = '#fbe7dd')
78     self.widgetPlot3.canvas.axis1.set_title(\
79     'Hormonal Contraceptives versus ' + title, \
80         fontweight ="bold",fontsize=16)
81     self.hist_feat_versus_other(self.df,df_target, \
82     self.df["Hormonal Contraceptives"],['Hormonal'], \
83     'No Hormonal'],self.widgetPlot3.canvas.axis1)
84     self.widgetPlot3.canvas.axis1.set_xlabel(title, \
85         fontweight ="bold",fontsize=16)
86
87     print(self.df.IUD.value_counts())
88     self.widgetPlot3.canvas.axis1 = \
89     self.widgetPlot3.canvas.figure.add_subplot(233, \
90         facecolor = '#fbe7dd')
91     self.widgetPlot3.canvas.axis1.set_title('IUD versus ' + \
92         title, fontweight ="bold",fontsize=16)
93     self.hist_feat_versus_other(self.df,df_target, \
94     self.df.IUD,['IUD','No IUD'], \
95     self.widgetPlot3.canvas.axis1)
96     self.widgetPlot3.canvas.axis1.set_xlabel(title, \
97         fontweight ="bold",fontsize=16)
98
99     print(self.df.STDs.value_counts())
100    self.widgetPlot3.canvas.axis1 = \
101    self.widgetPlot3.canvas.figure.add_subplot(234, \
102        facecolor = '#fbe7dd')
103    self.widgetPlot3.canvas.axis1.set_title('STDs versus ' + \
104        title, fontweight ="bold",fontsize=16)
105    self.hist_feat_versus_other(self.df,df_target, \

```

```

106 self.df.STDs,['STDs','No STDs'],\
107 self.widgetPlot3.canvas.axis1)
108 self.widgetPlot3.canvas.axis1.set_xlabel(title,\n
109     fontweight ="bold",fontsize=16)
110
111 print(self.df.Dx.value_counts())
112 self.widgetPlot3.canvas.axis1 = \
113 self.widgetPlot3.canvas.figure.add_subplot(235,\n
114     facecolor = '#fbe7dd')
115 self.widgetPlot3.canvas.axis1.set_title('Dx versus ' + \
116     title, fontweight ="bold",fontsize=16)
117 self.hist_feat_versus_other(self.df,df_target,\n
118 self.df.Dx,['Dx','No Dx'],self.widgetPlot3.canvas.axis1)
119 self.widgetPlot3.canvas.axis1.set_xlabel(title,\n
120     fontweight ="bold",fontsize=16)
121
122 print(self.df.Hinselmann.value_counts())
123 self.widgetPlot3.canvas.axis1 = \
124 self.widgetPlot3.canvas.figure.add_subplot(236,\n
125     facecolor = '#fbe7dd')
126 self.widgetPlot3.canvas.axis1.set_title(\n
127 'Hinselmann versus ' + title, \
128     fontweight ="bold",fontsize=16)
129 self.hist_feat_versus_other(self.df,df_target,\n
130 self.df.Hinselmann,['Hinselmann=1','Hinselmann=0'],\n
131 self.widgetPlot3.canvas.axis1)
132 self.widgetPlot3.canvas.axis1.set_xlabel(title,\n
133     fontweight ="bold",fontsize=16)
134
135 self.widgetPlot3.canvas.figure.tight_layout()
136 self.widgetPlot3.canvas.draw()
137
138 self.widgetPlot2.canvas.figure.clf()
139 print(self.df.Schiller.value_counts())
140 self.widgetPlot2.canvas.axis1 = \
141 self.widgetPlot2.canvas.figure.add_subplot(211,\n
142     facecolor = '#fbe7dd')
143 self.widgetPlot2.canvas.axis1.set_title(\n
144 'Schiller versus ' + title, \
145     fontweight ="bold",fontsize=16)
146 self.hist_feat_versus_other(self.df,df_target,\n
147 self.df.Schiller,['Schiller=1','Schiller=0'],\n
148 self.widgetPlot2.canvas.axis1)
149 self.widgetPlot2.canvas.axis1.set_xlabel(title,\n
150     fontweight ="bold",fontsize=16)
151
152 print(self.df.Citology.value_counts())
153 self.widgetPlot2.canvas.axis1 = \
154 self.widgetPlot2.canvas.figure.add_subplot(212,\n
155     facecolor = '#fbe7dd')
156 self.widgetPlot2.canvas.axis1.set_title(\n
157 'Citology versus ' + title, fontweight ="bold",\n
158     fontsize=16)
159 self.hist_feat_versus_other(self.df,df_target,\n

```

```

160 self.df.Citology,['Citology=1','Citology=0'],\
161 self.widgetPlot2.canvas.axis1)
162 self.widgetPlot2.canvas.axis1.set_xlabel(title,\
163     fontweight ="bold",fontsize=16)
164
165 self.widgetPlot2.canvas.figure.tight_layout()
166 self.widgetPlot2.canvas.draw()
167
168 self.widgetPlot1.canvas.figure.clf()
169 print(self.df.Biopsy.value_counts())
170 self.widgetPlot1.canvas.axis1 = \
171 self.widgetPlot1.canvas.figure.add_subplot(211,\
172     facecolor = '#fbe7dd')
173 self.widgetPlot1.canvas.axis1.set_title(\
174 'Biopsy versus ' + title, \
175     fontweight ="bold",fontsize=16)
176 self.prob_feat_versus_other(self.df,df_target,\
177 self.df.Biopsy,['Biopsy=1','Biopsy=0'],\
178 self.widgetPlot1.canvas.axis1)
179 self.widgetPlot1.canvas.axis1.set_xlabel(title,\
180     fontweight ="bold",fontsize=16)
181
182 print(self.df["Dx:HPV"].value_counts())
183 self.widgetPlot1.canvas.axis1 = \
184 self.widgetPlot1.canvas.figure.add_subplot(212,\
185     facecolor = '#fbe7dd')
186 self.widgetPlot1.canvas.axis1.set_title(\
'Dx:HPV versus ' + title, fontweight
="bold",fontsize=16)
self.prob_feat_versus_other(self.df,df_target,\
self.df["Dx:HPV"],['Dx:HPV=1','Dx:HPV=0'],\
self.widgetPlot1.canvas.axis1)
self.widgetPlot1.canvas.axis1.set_xlabel(title,\
fontweight ="bold",fontsize=16)
self.widgetPlot1.canvas.figure.tight_layout()
self.widgetPlot1.canvas.draw()

```

The code defines several methods within a class that are responsible for generating different types of visualizations. These methods create histograms and kernel density estimation (KDE) plots to compare the distribution of a selected feature against other categorical features or target variables. Let's break down the functionality of these methods:

1. hist_feat_versus_other() Method:

This method creates histograms to compare the distribution of a selected feature (feat) against another categorical feature (another). It customizes the appearance of the histogram plot, sets axis labels, grid lines, tick parameters, and a legend. The ax0 parameter specifies the axis where the plot is generated.

2. prob_feat_versus_other() Method:

Similar to the previous method, this one generates KDE plots to compare the distribution of the selected feature

(feat) against another categorical feature (another). It sets up the appearance of the KDE plot, including axis labels, grid lines, tick parameters, and a legend. The ax0 parameter specifies the axis where the plot is created.

3. dist_feat_versus_others() Method:

This method orchestrates the generation of multiple visualizations comparing the distribution of a specified target variable (df_target) against various categorical features. It creates a 2x3 grid of subplots and calls the previous two methods (hist_feat_versus_other and prob_feat_versus_other) to populate these subplots. The method customizes the titles, labels, and layout of the subplots.

Each of these methods contributes to the creation of informative visualizations that allow the user to explore relationships and distributions between different categorical features and target variables within the dataset. These visualizations can help in gaining insights and making data-driven decisions in various applications.

Step 2 Add this code to the end of **choose_plot()**:

```
1 if strCB == 'Age':  
2     self.dist_feat_versus_others(self.df.Age, "Age")  
3  
4 if strCB == 'Smokes':  
5     self.dist_feat_versus_others(self.df.Smokes,  
6         "Smokes")  
7  
8 if strCB == 'Smokes (years)':  
9     self.dist_feat_versus_others(\  
10        self.df["Smokes (years)"].apply(\  
11            pd.to_numeric, errors='coerce').astype('float64'),  
12            \  
13            "Smokes (years)")  
14  
15 if strCB == 'First sexual intercourse':  
16     self.dist_feat_versus_others(\  
17        self.df["First sexual intercourse"].apply(\  
18            pd.to_numeric, errors='coerce').astype('float64'),  
19            \  
20            "First sexual intercourse")  
21  
22 if strCB == 'Num of pregnancies':  
23     self.dist_feat_versus_others(\  
24        self.df["Num of pregnancies"].apply(\  
25            pd.to_numeric, errors='coerce').astype('float64'),  
26            \  
27            "Num of pregnancies")  
28  
29 if strCB == 'Hormonal Contraceptives (years)':  
30     self.dist_feat_versus_others(\
```

```

31 self.df["Hormonal Contraceptives (years)"].apply(
32     pd.to_numeric, errors='coerce').astype('float64'),
33 \
34 "Hormonal Contraceptives (years)"
35
36 if strCB == 'IUD (years)':
37     self.dist_feat_versus_others(
38         self.df["IUD (years)"].apply(pd.to_numeric, \
39             errors='coerce').astype('float64'), "IUD (years)")
40
41 if strCB == 'Number of sexual partners':
42     self.dist_feat_versus_others(
43         self.df["Number of sexual partners"].apply(
44             pd.to_numeric, errors='coerce').astype('float64'),
45 \
46     "Number of sexual partners")

```

The code is a conditional block that checks the value of strCB (presumably a selected feature from a dropdown) and calls the dist_feat_versus_others() method with appropriate arguments based on the selected feature. This code generates distributions of the selected feature against various categorical features for visualization. Here's a breakdown of each condition:

1. if strCB == 'Age':

This condition generates distribution plots comparing the feature "Age" against various categorical features.

2. if strCB == 'Smokes':

This condition generates distribution plots comparing the feature "Smokes" against various categorical features.

3. if strCB == 'Smokes (years)':

This condition generates distribution plots comparing the feature "Smokes (years)" against various categorical features.

4. if strCB == 'First sexual intercourse':

This condition generates distribution plots comparing the feature "First sexual intercourse" against various categorical features.

5. if strCB == 'Num of pregnancies':

This condition generates distribution plots comparing the feature "Num of pregnancies" against various categorical features.

6. if strCB == 'Hormonal Contraceptives (years)':

This condition generates distribution plots comparing the feature "Hormonal Contraceptives (years)" against various categorical features.

7. if strCB == 'IUD (years)':

This condition generates distribution plots comparing the feature "IUD (years)" against various categorical features.

8. if strCB == 'Number of sexual partners':

This condition generates distribution plots comparing the feature "Number of sexual partners" against various categorical features.

In each condition, the appropriate columns from the DataFrame are selected and passed to the dist_feat_versus_others() method along with the title for the visualization. This code provides a way to dynamically create and display distribution plots for different selected features and categorical features, allowing users to explore relationships within the data.

- Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Age** item from **cbData** widget. You will see the result as shown in Figure 128.



Figure 128 The histogram and density of **Age** feature versus some other features

- Step 4 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Num of pregnancies** item from **cbData** widget. You will see the result as shown in Figure 129.



Figure 129 The histogram and density of **Num of pregnancies** feature versus some other features

- Step 5 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Smokes (years)** item from **cbData** widget. You will see the result as shown in Figure 130.



Figure 130 The histogram and density of **Smokes (years)** feature versus some other features

- Step 6 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Hormonal Contraceptives (years)** item from **cbData** widget. You will see the result as shown in Figure 131.



Figure 131 The histogram and density of **Hormonal Contraceptives (years)** feature versus some other features

- Step 7 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **First sexual intercourse** item from **cbData** widget. You will see the result as shown in Figure 132.



Figure 132 The histogram and density of **First sexual intercourse** feature versus some other features

Correlation Matrix and Features Importance

Step Define **plot_corr()** method to plot correlation matrix on a widget:

1

```
1 def plot_corr(self, data, widget):
2     corrrdata = data.corr()
3     sns.heatmap(corrrdata, ax = widget.canvas.axis1, \
4                  lw=1, annot=True, cmap="Reds")
5     widget.canvas.axis1.set_title('Correlation Matrix', \
6                                   fontweight ="bold", fontsize=20)
7     widget.canvas.figure.tight_layout()
8     widget.canvas.draw()
```

This method calculates the correlation matrix using the corr() function of the input DataFrame data. It then uses Seaborn's heatmap function to create a colored heatmap of the correlation matrix. The heatmap is annotated with the correlation values, and the color map "Reds" is used for the color scheme. The title is set for the heatmap, and the layout is adjusted to ensure the heatmap fits within the widget. Finally, the heatmap is drawn on the designated widget using widget.canvas.draw().

Step Add this code to the end of **choose_plot()** method:

2

```
1 if strCB == 'Correlation Matrix':
2     self.widgetPlot3.canvas.figure.clf()
3     self.widgetPlot3.canvas.axis1 = \
4     self.widgetPlot3.canvas.figure.add_subplot(111)
5     X,_ = self.fit_dataset(self.df)
6     self.plot_corr(X, self.widgetPlot3)
```

In this code block:

1. The widgetPlot3 canvas figure is cleared using self.widgetPlot3.canvas.figure.clf() to remove any previous content.
2. A new subplot is added to the widgetPlot3 canvas using self.widgetPlot3.canvas.figure.add_subplot(111).
3. The fit_dataset() method is used to prepare the dataset X for generating the correlation matrix. It appears that only the feature attributes are being used here, as the target variable (_) is ignored.
4. Finally, the plot_corr() method is called with the prepared dataset X and the widgetPlot3 widget, which displays the correlation matrix heatmap on the canvas.

This code essentially updates the widgetPlot3 canvas to display a correlation matrix heatmap for the selected feature attributes

when the value of strCB is 'Correlation Matrix'.

- Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Correlation Matrix** item from **cbData** widget. You will see the result as shown in Figure 133.



Figure 133 Correlation matrix

- Step 4 Define **plot_importance()** method to plot features importance on a widget:

```
1 def plot_importance(self, widget):
2     #Compares different feature importances
3     r = ExtraTreesClassifier(random_state=0)
4     X,y = self.fit_dataset(self.df)
5     r.fit(X, y)
6     feature_importance_normalized = \
7         np.std([tree.feature_importances_ for tree in \
8             r.estimators_], axis = 0)
9
10    sns.barplot(feature_importance_normalized, \
11        X.columns, ax = widget.canvas.axis1)
12    widget.canvas.axis1.set_ylabel('Feature Labels',\
13        fontweight ="bold",fontsize=15)
14    widget.canvas.axis1.set_xlabel('Features
15        Importance',\
16        fontweight ="bold",fontsize=15)
17    widget.canvas.axis1.set_title(\'
18        'Comparison of different Features Importance',\
19        fontweight ="bold",fontsize=20)
20    widget.canvas.figure.tight_layout()
21    widget.canvas.draw()
```

In this code:

1. An `ExtraTreesClassifier` model `r` is initialized with a specified random state.
2. The `fit_dataset()` method is used to prepare the dataset `X` and the target variable `y` for calculating feature importance.
3. The `ExtraTreesClassifier` model `r` is fitted to the data using `r.fit(X, y)`.

4. The normalized feature importance values are calculated across all trees in the ensemble.
5. The sns.barplot function is used to create a bar plot of the normalized feature importances, with the feature labels on the y-axis and the importance values on the x-axis. The plot is created on the widget.canvas.axis1.
6. Labels, title, and other formatting settings are applied to the plot.
7. The layout is adjusted and the plot is drawn on the canvas of the specified widget.

This method essentially generates a bar plot to visualize and compare the importance of different features in the dataset using the ExtraTreesClassifier algorithm.

Step 5 Add this code to the end of **choose_plot()** method:

```

1 if strCB == 'Features Importance':
2     self.widgetPlot3.canvas.figure.clf()
3     self.widgetPlot3.canvas.axis1 = \
4         self.widgetPlot3.canvas.figure.add_subplot(111)
5     self.plot_importance(self.widgetPlot3)

```

In this code:

1. The condition if strCB == 'Features Importance' checks if the value of strCB (a combobox) is equal to 'Features Importance'.
2. If the condition is met, the existing plot on widgetPlot3 is cleared using self.widgetPlot3.canvas.figure.clf().
3. A new subplot is added to widgetPlot3 using self.widgetPlot3.canvas.figure.add_subplot(111).
4. The plot_importance() method is called, passing widgetPlot3 as an argument to generate the feature importance plot.

Overall, this code ensures that when the user selects 'Features Importance' from a combobox or similar UI element, the existing plot is cleared, a new subplot is added, and a new feature importance plot is generated and displayed on the widgetPlot3 canvas.

Step 6 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Then, choose **Features Importance** item from **cbData** widget. You will see the result as shown in Figure 134.



Figure 134 The features importance

Real Values versus Predicted Values and Confusion Matrix

Step 1 Define **plot_real_pred_val()** method to plot true values and predicted values and **plot_cm()** method to calculate and plot confusion matrix:

```

1 def plot_real_pred_val(self, Y_pred, Y_test, widget,
2 title):
3     #Calculate Metrics
4     acc=accuracy_score(Y_test,Y_pred)
5
6     #Output plot
7     widget.canvas.figure.clf()
8     widget.canvas.axis1 = \
9         widget.canvas.figure.add_subplot(111, \
10            facecolor='steelblue')
11    widget.canvas.axis1.scatter(range(len(Y_pred)), \
12
13    Y_pred,color="yellow",lw=5,label="Predictions")
14    widget.canvas.axis1.scatter(range(len(Y_test)), \
15        Y_test,color="red",label="Actual")
16    widget.canvas.axis1.set_title(
17        "Prediction Values vs Real Values of " + title, \
18        fontsize=10)
19    widget.canvas.axis1.set_xlabel("Accuracy: " + \
20        str(round((acc*100),3)) + "%")
21    widget.canvas.axis1.legend()
22    widget.canvas.axis1.grid(True, alpha=0.75, lw=1,
23    ls='-.')
24    widget.canvas.draw()
25
26 def plot_cm(self, Y_pred, Y_test, widget, title):
27     cm=confusion_matrix(Y_test,Y_pred)
28     widget.canvas.figure.clf()
29     widget.canvas.axis1 =
30     widget.canvas.figure.add_subplot(111)
31     class_label = ["1", "0"]
32     df_cm = pd.DataFrame(cm, \
33         index=class_label,columns=class_label)

```

```
34     sns.heatmap(df_cm, ax=widget.canvas.axis1, \
35         annot=True,
36         cmap='plasma', linewidths=2, fmt='d')
        widget.canvas.axis1.set_title("Confusion Matrix of
" + \
            title, fontsize=10)
        widget.canvas.axis1.set_xlabel("Predicted")
        widget.canvas.axis1.set_ylabel("True")
        widget.canvas.draw()
```

In the `plot_real_pred_val()` method:

1. The method calculates the accuracy using the `accuracy_score` function from scikit-learn.
2. The existing plot on the specified widget is cleared, and a new subplot is added.
3. Scatter plots are created to visualize the predicted values (`Y_pred`) in yellow and the actual values (`Y_test`) in red.
4. The title and accuracy information are set for the plot.
5. The legend, grid, and other formatting are applied to the plot.
6. Finally, the plot is drawn on the widget canvas.

In the `plot_cm()` method:

1. The method calculates the confusion matrix using the `confusion_matrix` function from scikit-learn.
2. The existing plot on the specified widget is cleared, and a new subplot is added.
3. A heatmap is created using Seaborn to visualize the confusion matrix.
4. The title, x-label, and y-label are set for the plot.
5. The plot is drawn on the widget canvas.

These methods can be used to visualize prediction results and confusion matrices for evaluating the performance of a machine learning model. You can call these methods with appropriate arguments to generate the desired plots.

ROC

Step Define `plot_roc()` method to plot ROC:

1

```
1 def plot_roc(self, clf, title, widget):
2     pred_prob = clf.predict_proba(xtest)
```

```

3 pred_prob = pred_prob[:, 1]
4 fpr, tpr, thresholds = roc_curve(ytest, pred_prob)
5 widget.canvas.axis1.plot(fpr,tpr, label='ANN',\
6     color='crimson', linewidth=3)
7 widget.canvas.axis1.set_xlabel('False Positive')
8 Rate')
9 widget.canvas.axis1.set_ylabel('True Positive')
10 Rate')
11 widget.canvas.axis1.set_title('ROC Curve of ' +
12 title, \
13     fontweight ="bold", fontsize=15)
14 widget.canvas.axis1.grid(True, alpha=0.75, lw=1,
15 ls='-.')
16 widget.canvas.figure.tight_layout()
17 widget.canvas.draw()

```

Here's a step-by-step explanation of the plot_roc() method:

1. Inputs: The method takes three arguments:
 - clf: The classifier (model) for which the ROC curve will be plotted.
 - title: A string indicating the title of the ROC curve plot.
 - widget: The canvas or plot widget where the ROC curve will be displayed.
2. Calculate Predicted Probabilities: The method first uses the predict_proba() method of the classifier (clf) to calculate the predicted probabilities of the positive class for the test set (xtest). These predicted probabilities are stored in the pred_prob variable.
3. Extract True Positive Rate (TPR) and False Positive Rate (FPR): The method then uses the roc_curve function from scikit-learn to calculate the False Positive Rate (FPR), True Positive Rate (TPR), and thresholds for different probability thresholds. These values are calculated based on the true labels (ytest) and the predicted probabilities (pred_prob).
4. Plot the ROC Curve: The method plots the ROC curve using the FPR on the x-axis and the TPR on the y-axis. The curve is drawn with a crimson color and a linewidth of 3. The label 'ANN' is added to the plot legend.
5. Set Labels and Title: The x-axis and y-axis labels are set to 'False Positive Rate' and 'True Positive Rate', respectively. The title of

- the plot is set to 'ROC Curve of ' followed by the provided title.
6. Styling and Grid: The method sets the grid, which is drawn with a dotted line style ('.-'). Other visual styling attributes, such as linewidths and alpha (transparency), are applied to the grid and the plot.
 7. Adjust Layout and Draw: The `tight_layout` function is used to adjust the layout of the plot to ensure that all elements fit properly. Finally, the plot is drawn on the specified widget canvas.

This method is used to plot the Receiver Operating Characteristic (ROC) curve for evaluating the performance of a binary classification model (`clf`). The ROC curve visually illustrates the trade-off between the True Positive Rate (sensitivity) and the False Positive Rate (1 - specificity) across different probability thresholds. It helps assess how well the model distinguishes between the two classes and allows comparison of different models or parameter settings.

Learning Curve

Step 1 Define `plot_learning_curve()` to plot learning curve of any machine learning model:

```

1 def plot_learning_curve(self, estimator, title, X, y,
2     widget, ylim=None, cv=None, n_jobs=None,
3     train_sizes=np.linspace(.1, 1.0, 5)):
4     widget.canvas.axis1.set_title(title)
5     if ylim is not None:
6         widget.canvas.axis1.set_ylimits(*ylim)
7         widget.canvas.axis1.set_xlabel("Training examples")
8         widget.canvas.axis1.set_ylabel("Score")
9
10    train_sizes, train_scores, test_scores, fit_times, _ =
11    \
12    learning_curve(estimator, X, y, cv=cv,
13    n_jobs=n_jobs,
14        train_sizes=train_sizes,
15        return_times=True)
16    train_scores_mean = np.mean(train_scores,
17    axis=1)
18    train_scores_std = np.std(train_scores, axis=1)
19    test_scores_mean = np.mean(test_scores, axis=1)
20    test_scores_std = np.std(test_scores, axis=1)
21
22
23 # Plot learning curve
24    widget.canvas.axis1.grid()
25    widget.canvas.axis1.fill_between(train_sizes, \
26        train_scores_mean - train_scores_std, \

```

```

27     train_scores_mean + train_scores_std,
28     alpha=0.1,color="r")
29     widget.canvas.axis1.fill_between(train_sizes,\n
30         test_scores_mean - test_scores_std,\n
31         test_scores_mean + test_scores_std, alpha=0.1,\n
color="g")
    widget.canvas.axis1.plot(train_sizes,\n
train_scores_mean,\n
'o-', color="r", label="Training score")
    widget.canvas.axis1.plot(train_sizes,\n
test_scores_mean,\n
'o-', color="g", label="Cross-validation score")
    widget.canvas.axis1.legend(loc="best")

```

Here's an explanation of the `plot_learning_curve()` method step by step:

1. Inputs: The method takes several arguments:
`estimator`: The machine learning model or estimator for which the learning curve will be plotted.
 - `title`: A string representing the title of the learning curve plot.
 - `X`: The feature matrix of the dataset.
 - `y`: The target vector of the dataset.
 - `widget`: The canvas or plot widget where the learning curve will be displayed.
 - `ylim`: Tuple specifying the y-axis limits for the plot (optional).
 - `cv`: Cross-validation strategy (optional).
 - `n_jobs`: Number of parallel jobs (optional).
 - `train_sizes`: An array of training set sizes to use for generating the learning curve (default is [0.1, 0.325, 0.55, 0.775, 1.0]).
2. Set Plot Labels and Title: The method sets the title of the plot to the provided title. If `ylim` is provided, it sets the y-axis limits accordingly. It also sets the x-axis and y-axis labels to "Training examples" and "Score," respectively.
3. Calculate Learning Curve Data: The method uses the `learning_curve` function from scikit-learn to calculate the learning curve data. It computes training scores, test scores, and fit times for different training set sizes (`train_sizes`) using cross-validation (`cv`). The

- `return_times=True` argument ensures that fit times are returned along with the scores.
4. Calculate Mean and Standard Deviation: The mean and standard deviation of training and test scores are calculated along the rows (i.e., across different cross-validation folds) to obtain representative values for each training set size.
 5. Plot Learning Curve: The method plots the learning curve by filling the area between the mean training scores minus/plus the standard deviation (`train_scores_mean ± train_scores_std`) with a light red color (`alpha=0.1`) to represent the training score variability. Similarly, the area between the mean test scores minus/plus the standard deviation (`test_scores_mean ± test_scores_std`) is filled with a light green color. The mean training scores and mean test scores are plotted as red and green markers, respectively, connected by lines ('o-').
 6. Add Legend: A legend is added to the plot to distinguish between the training score and cross-validation score.
 7. Draw Grid and Display: The grid is added to the plot, and it's drawn with lines. The plot is then drawn on the specified widget canvas.

The `plot_learning_curve()` method is used to visualize the performance of a machine learning model as the training set size increases. It helps in understanding whether the model is overfitting or underfitting and provides insights into how well the model generalizes to new data. The plot shows the training score and cross-validation score as functions of the training set size, which helps in assessing bias and variance trade-offs.

Scalability and Performance Curves

- Step 1 Define `plot_scalability_curve()` to plot the scalability of the model and `plot_performance_curve()` method to plot performance of the model on a widget:

```

1 def plot_scalability_curve(self, estimator, title, X, y,
2     widget, ylim=None, cv=None, n_jobs=None,
3     train_sizes=np.linspace(.1, 1.0, 5)):
4     widget.canvas.axis1.set_title(title, \
5         fontweight ="bold", fontsize=15)

```

```

6 if ylim is not None:
7     widget.canvas.axis1.set_ylim(*ylim)
8     widget.canvas.axis1.set_xlabel("Training
9 examples")
10    widget.canvas.axis1.set_ylabel("Score")
11
12    train_sizes, train_scores, test_scores, fit_times, _ =
13    \
14        learning_curve(estimator, X, y, cv=cv,
15        n_jobs=n_jobs,
16            train_sizes=train_sizes,
17            return_times=True)
18    fit_times_mean = np.mean(fit_times, axis=1)
19    fit_times_std = np.std(fit_times, axis=1)
20
21 # Plot n_samples vs fit_times
22    widget.canvas.axis1.grid()
23    widget.canvas.axis1.plot(train_sizes,
24    fit_times_mean, 'o-')
25    widget.canvas.axis1.fill_between(train_sizes, \
26        fit_times_mean - fit_times_std,\n
27        fit_times_mean + fit_times_std, alpha=0.1)
28    widget.canvas.axis1.set_xlabel("Training
29 examples")
30    widget.canvas.axis1.set_ylabel("fit_times")
31
32 def plot_performance_curve(self, estimator, title, X,
33 y, \
34     widget, ylim=None, cv=None, n_jobs=None, \
35     train_sizes=np.linspace(.1, 1.0, 5)):
36
37     widget.canvas.axis1.set_title(title, \
38         fontweight ="bold", fontsize=15)
39     if ylim is not None:
40         widget.canvas.axis1.set_ylim(*ylim)
41
42     widget.canvas.axis1.set_xlabel("Training
43 examples")
44     widget.canvas.axis1.set_ylabel("Score")
45
46     train_sizes, train_scores, test_scores, fit_times, _ =
47     \
48        learning_curve(estimator, X, y, cv=cv,
49        n_jobs=n_jobs,
50            train_sizes=train_sizes,
51            return_times=True)
52    test_scores_mean = np.mean(test_scores, axis=1)
53    test_scores_std = np.std(test_scores, axis=1)
54    fit_times_mean = np.mean(fit_times, axis=1)
55
56 # Plot n_samples vs fit_times
57    widget.canvas.axis1.grid()
58    widget.canvas.axis1.plot(fit_times_mean, \
59    test_scores_mean, 'o-')

```

```
    widget.canvas.axis1.fill_between(fit_times_mean,  
    \  
        test_scores_mean - test_scores_std,\  
        test_scores_mean + test_scores_std, alpha=0.1)  
    widget.canvas.axis1.set_xlabel("fit_times")  
    widget.canvas.axis1.set_ylabel("Score")
```

plot_scalability_curve() Methode:

1. Inputs: The method takes similar arguments to the plot_learning_curve method:
 - estimator: The machine learning model or estimator for which the scalability curve will be plotted.
 - title: A string representing the title of the scalability curve plot.
 - X: The feature matrix of the dataset.
 - y: The target vector of the dataset.
 - widget: The canvas or plot widget where the scalability curve will be displayed.
 - ylim: Tuple specifying the y-axis limits for the plot (optional).
 - cv: Cross-validation strategy (optional).
 - n_jobs: Number of parallel jobs (optional).
 - train_sizes: An array of training set sizes to use for generating the scalability curve (default is [0.1, 0.325, 0.55, 0.775, 1.0]).
2. Set Plot Labels and Title: Similar to the learning curve methods, this method sets the title, y-axis label, and x-axis label for the plot.
3. Calculate Scalability Curve Data: The method uses the learning_curve function to calculate scalability curve data. It computes fit times for different training set sizes (train_sizes) using cross-validation (cv) and returns the fit times using return_times=True.
4. Calculate Mean and Standard Deviation: The mean and standard deviation of fit times are calculated along the rows (i.e., across different cross-validation folds) to obtain representative values for each training set size.

5. Plot Scalability Curve: The method plots the scalability curve by plotting the mean fit times against the training set sizes. The area between the mean fit times minus/plus the standard deviation ($\text{fit_times_mean} \pm \text{fit_times_std}$) is filled with a light color ($\text{alpha}=0.1$).
6. Set Labels and Display: The x-axis is labeled with "Training examples," and the y-axis is labeled with "fit_times." The grid is added to the plot, and it's drawn with lines. The plot is then drawn on the specified widget canvas.

`plot_performance_curve()` Method:

1. Inputs: This method has the same inputs as the `plot_scalability_curve` method.
2. Set Plot Labels and Title: Similar to the previous methods, this method sets the title, y-axis label, and x-axis label for the plot.
3. Calculate Performance Curve Data: The method calculates the learning curve data, including training scores, test scores, and fit times.
4. Calculate Mean and Standard Deviation: The mean and standard deviation of test scores are calculated along the rows (i.e., across different cross-validation folds) to obtain representative values for each training set size. Additionally, the mean fit times are calculated.
5. Plot Performance Curve: The method plots the performance curve by plotting the mean test scores against the mean fit times. The area between the mean test scores minus/plus the standard deviation ($\text{test_scores_mean} \pm \text{test_scores_std}$) is filled with a light color ($\text{alpha}=0.1$).
6. Set Labels and Display: The x-axis is labeled with "fit_times," and the y-axis is labeled with "Score." The grid is added to the plot, and it's drawn with lines. The plot is then drawn on the specified widget canvas.

These methods are used to visualize the scalability and performance of a machine learning model with respect to training set sizes and fit times. They provide insights into how well the model's training time scales with the dataset

size and how the model's performance changes with respect to fit times.

Training Model and Predicting Result

Step 1 Define **train_model()** and **predict_model()** methods to train any classifier and calculate the prediction:

```
1  def train_model(self, model, X, y):
2      model.fit(X, y)
3      return model
4
5  def predict_model(self, model, X, proba=False):
6      if ~proba:
7          y_pred = model.predict(X)
8      else:
9          y_pred_proba = model.predict_proba(X)
10         y_pred = np.argmax(y_pred_proba, axis=1)
11
12     return y_pred
```

Here's a breakdown of how these methods work:

train_model():

1. Inputs:
 - model: The machine learning model (classifier or regressor) that you want to train.
 - X: The feature matrix of your training dataset.
 - y: The target vector of your training dataset.
2. Model Training: The method calls the fit function of the provided model and passes in the training data X and y. This trains the model on the provided training dataset.
3. Return: The trained model is returned from the function, now updated with the learned patterns from the training data.

predict_model():

1. Inputs:
 - model: The trained machine learning model.
 - X: The feature matrix of the dataset for which you want to make predictions.
 - proba: A boolean flag indicating whether to return class probabilities (True) or predicted class labels (False). By default, it's set to False.

2. Prediction:

If proba is set to False (default behavior):

The method calls the predict function of the provided model and passes in the feature matrix X. This predicts the class labels for the input samples.

3. If proba is set to True:

a. The method calls the predict_proba function of the provided model and passes in the feature matrix X. This returns the predicted class probabilities for each class.

b. The argmax function is used to determine the class with the highest probability for each sample, resulting in the predicted class labels.

4. Return:

The method returns the predicted class labels (y_pred) or class probabilities (y_pred_proba) based on the value of the proba flag.

These methods provide a streamlined way to train machine learning models and make predictions using them. The train_model() method is used for model training, and the predict_model method is used for making predictions, either returning class labels or class probabilities based on the proba flag.

- Step 2 Define **run_model()** method to calculate accuracy and precision. It also invokes six methods to plot confusion matrix, true values versus predicted values diagram, ROC, learning curve, and decision boundaries:

```
1 def run_model(self, name, scaling, model, X_train,
2 X_test, y_train, y_test, train=True, proba=True):
3     if train == True:
4         model = self.train_model(model, X_train,
5         y_train)
6         y_pred = self.predict_model(model, X_test, proba)
7
8         accuracy = accuracy_score(y_test, y_pred)
9         recall = recall_score(y_test, y_pred)
10        precision = precision_score(y_test, y_pred)
11        f1 = f1_score(y_test, y_pred)
12
13        print('accuracy: ', accuracy)
14        print('recall: ', recall)
15        print('precision: ', precision)
16        print('f1: ', f1)
17        print(classification_report(y_test, y_pred))
18
19        self.widgetPlot1.canvas.figure.clf()
20        self.widgetPlot1.canvas.axis1 = \
```

```

21 self.widgetPlot1.canvas.figure.add_subplot(111,\n22     facecolor ='#fbe7dd')\n23 self.plot_cm(y_pred, y_test, self.widgetPlot1,\n24     name + " -- " + scaling)\n25 self.widgetPlot1.canvas.figure.tight_layout()\n26 self.widgetPlot1.canvas.draw()\n27\n28 self.widgetPlot2.canvas.figure.clf()\n29 self.widgetPlot2.canvas.axis1 = \\\n30 self.widgetPlot2.canvas.figure.add_subplot(111,\n31     facecolor ='#fbe7dd')\n32 self.plot_real_pred_val(y_pred, y_test, \\\n33 self.widgetPlot2, name + " -- " + scaling)\n34 self.widgetPlot2.canvas.figure.tight_layout()\n35 self.widgetPlot2.canvas.draw()\n36\n37 self.widgetPlot3.canvas.figure.clf()\n38 self.widgetPlot3.canvas.axis1 = \\\n39 self.widgetPlot3.canvas.figure.add_subplot(221,\n40     facecolor ='#fbe7dd')\n41 self.plot_roc(model, X_test, y_test, \\\n42     name + " -- " + scaling, self.widgetPlot3)\n43 self.widgetPlot3.canvas.figure.tight_layout()\n44\n45 self.widgetPlot3.canvas.axis1 = \\\n46 self.widgetPlot3.canvas.figure.add_subplot(222,\n47     facecolor ='#fbe7dd')\n48 self.plot_learning_curve(model, \\\n49 'Learning Curve' + " -- " + scaling, X_train, \\\n50     y_train, self.widgetPlot3)\n51 self.widgetPlot3.canvas.figure.tight_layout()\n52\n53 self.widgetPlot3.canvas.axis1 = \\\n54 self.widgetPlot3.canvas.figure.add_subplot(223,\n55     facecolor ='#fbe7dd')\n56 self.plot_scalability_curve(model, 'Scalability of ' + \\\n57     name + " -- " + scaling, X_train, y_train, \\\n58 self.widgetPlot3)\n59 self.widgetPlot3.canvas.figure.tight_layout()\n60\n61 self.widgetPlot3.canvas.axis1 = \\\n62 self.widgetPlot3.canvas.figure.add_subplot(224,\n63     facecolor ='#fbe7dd')\n64 self.plot_performance_curve(model, \\\n65 'Performance of ' + name + " -- " + scaling, \\\n66     X_train, y_train, self.widgetPlot3)\n67 self.widgetPlot3.canvas.figure.tight_layout()\n68\nself.widgetPlot3.canvas.draw()

```

The `run_model()` method seems to be a comprehensive function to perform a series of steps and analyses for a machine learning model. Let's break down how it works:

1. Inputs:

- name: A string representing the name or identifier of the model being run.
- scaling: A string indicating the type of data scaling/normalization applied to the features.
- model: The machine learning model to be run.
- X_train, X_test: Feature matrices for the training and test datasets.
- y_train, y_test: Target vectors for the training and test datasets.
- train: A boolean flag indicating whether to train the model. Default is True.
- proba: A boolean flag indicating whether to return class probabilities (True) or predicted class labels (False). Default is True.

2. Training and Prediction:

- If train is True, the method trains the model using the train_model method and the provided training data.
- The method then uses the trained model to make predictions on the test data using the predict_model() method.

3. Evaluation Metrics and Reports:

- The method calculates various evaluation metrics such as accuracy, recall, precision, and F1-score using accuracy_score, recall_score, precision_score, and f1_score from scikit-learn.
- It also prints a classification report using classification_report from scikit-learn, providing a comprehensive overview of metrics for each class.

4. Confusion Matrix and Real vs. Predicted Value Plot:

- The method initializes and displays two subplots (widgetPlot1 and widgetPlot2):
- widgetPlot1: Displays the confusion matrix using plot_cm.
- widgetPlot2: Compares real vs. predicted values using plot_real_pred_val.

5. ROC Curve and Learning Curve:

- The method initializes the widgetPlot3 subplot (or axis) to display plots related to model performance.
- It creates two subplots within widgetPlot3:
 - The first subplot shows the ROC curve using plot_roc.
 - The second subplot shows the learning curve using plot_learning_curve.

6. Scalability and Performance Curves:

The method adds two more subplots within widgetPlot3:

- The third subplot shows the scalability curve using plot_scalability_curve.
- The fourth subplot shows the performance curve using plot_performance_curve.

7. Display Plots:

The method adjusts the layout and draws each of the subplots in widgetPlot3.

8. Output:

After running the method, various plots and metrics are displayed, providing insights into the model's performance, learning curve, scalability, and more.

Overall, the run_model() method is a comprehensive function that automates the process of training, evaluating, and visualizing the performance of a machine learning model across different aspects. It helps streamline the analysis and presentation of results during the model development and evaluation process.

Logistic Regression Classifier

Step 1 Define **build_train_lr()** method to build and train Logistic Regression (LR) classifier using three feature scaling: Raw, Normalization, and Standardization:

```

1  def build_train_lr(self):
2    if path.isfile('logregRaw.pkl'):
3      #Loads model
4      self.logregRaw = joblib.load('logregRaw.pkl')
5      self.logregNorm = joblib.load('logregNorm.pkl')
6      self.logregStand = joblib.load('logregStand.pkl')
7
8    if self.rbRaw.isChecked():
9      self.run_model('Logistic Regression', 'Raw', \
10      self.logregRaw, self.X_train_raw, \
11      self.X_test_raw, self.y_train_raw, \

```

```

12 self.y_test_raw)
13 if self.rbNorm.isChecked():
14     self.run_model('Logistic Regression', \
15     'Normalization', self.logregNorm, \
16     self.X_train_norm, self.X_test_norm, \
17     self.y_train_norm, self.y_test_norm)
18
19 if self.rbStand.isChecked():
20     self.run_model('Logistic Regression', \
21     'Standardization', self.logregStand, \
22     self.X_train_stand, self.X_test_stand, \
23     self.y_train_stand, self.y_test_stand)
24
25 else:
26     #Builds and trains Logistic Regression
27     self.logregRaw = LogisticRegression(solver='lbfgs',
28     \
29         max_iter=1000, random_state=2021)
30     self.logregNorm =
31     LogisticRegression(solver='lbfgs', \
32         max_iter=1000, random_state=2021)
33     self.logregStand =
34     LogisticRegression(solver='lbfgs', \
35         max_iter=1000, random_state=2021)
36
37 if self.rbRaw.isChecked():
38     self.run_model('Logistic Regression', 'Raw', \
39     self.logregRaw, self.X_train_raw, \
40     self.X_test_raw, self.y_train_raw, \
41     self.y_test_raw)
42 if self.rbNorm.isChecked():
43     self.run_model('Logistic Regression', \
44     'Normalization', self.logregNorm, \
45     self.X_train_norm, self.X_test_norm, \
46     self.y_train_norm, self.y_test_norm)
47
48 if self.rbStand.isChecked():
49     self.run_model('Logistic Regression', \
50     'Standardization', self.logregStand, \
51     self.X_train_stand, self.X_test_stand, \
52     self.y_train_stand, self.y_test_stand)
53
54 #Saves model
55 joblib.dump(self.logregRaw, 'logregRaw.pkl')
56 joblib.dump(self.logregNorm, 'logregNorm.pkl')
57 joblib.dump(self.logregStand, 'logregStand.pkl')

```

The build_train_lr() function is designed to construct and train Logistic Regression models for predicting cervical cancer using different data scaling methods. It first checks if pre-trained models exist in saved files. If they do, it loads these models and conducts evaluations based on the user's chosen scaling options (raw, normalization, or standardization). If pre-trained models are

unavailable, the function initializes new Logistic Regression models, trains them using training data, evaluates their predictive performance, and then saves these models for future use.

When pre-trained models are present (saved as logregRaw.pkl, logregNorm.pkl, and logregStand.pkl), the function loads them. It then determines the selected scaling option (raw, normalization, or standardization) and utilizes the run_model routine to assess and present the models' predictive capabilities on the corresponding scaled datasets.

In cases where pre-trained models are absent, the function initializes three new Logistic Regression models with defined parameters. Similarly, it employs the run_model approach to gauge and showcase the predictive efficacy of these freshly trained models across the chosen scaling options.

Upon completing the evaluation and presentation phase, the function ensures the retention of the trained models by saving them as files (logregRaw.pkl, logregNorm.pkl, logregStand.pkl). This process streamlines future utilization, obviating the need for redundant model training whenever the program is executed.

In essence, the build_train_lr function adeptly oversees the lifecycle of constructing, training, evaluating, and preserving Logistic Regression models for cervical cancer prediction, integrating diverse scaling techniques to facilitate a coherent and systematic approach to model development and analysis.

Step 2 Define **choose_ML_model()** method to read the selected item in **cbClassifier** widget:

```
1 def choose_ML_model(self):
2     strCB = self.cbClassifier.currentText()
3
4     if strCB == 'Logistic Regression':
5         self.build_train_lr()
```

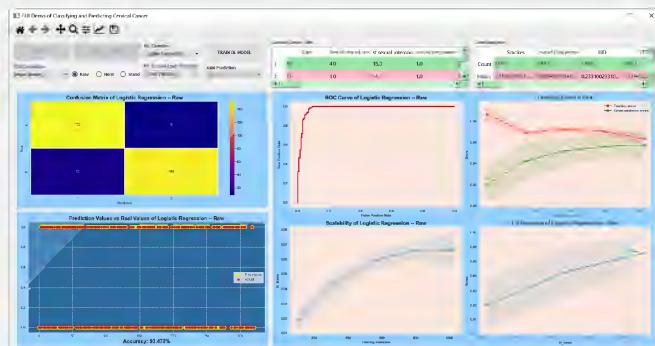


Figure 135 The result using LR model with raw feature scaling

Step 3 Connect **currentIndexChanged()** event of **cbClassifier** widget

- 3 with `choose_ML_model()` method and put it inside `__init__()` method as shown in line 12-13:

```
1 def __init__(self):
2     QMainWindow.__init__(self)
3     loadUi("gui_cervical.ui",self)
4     self.setWindowTitle(
5         "GUI Demo of Classifying and Predicting Cervical
6         Cancer")
7     self.addToolBar(NavigationToolbar(
8         self.widgetPlot1.canvas, self))
9     self.pbLoad.clicked.connect(self.import_dataset)
10    self.initial_state(False)
11    self.pbTrainML.clicked.connect(self.train_model_ML)
12    self.cbData.currentIndexChanged.connect(self.choose_plot
13 )
13     self.cbClassifier.currentIndexChanged.connect(
14         self.choose_ML_model)
```

The line of code you provided is setting up a connection between the `currentIndexChanged` signal of a combo box widget (`self.cbClassifier`) and a method called `choose_ML_model`. This connection enables the `choose_ML_model()` method to be automatically invoked whenever the selected item in the combo box changes.

In other words, when a user selects a different option from the dropdown list in the combo box, the `currentIndexChanged` signal is emitted. This signal triggers the execution of the `choose_ML_model` method, allowing you to respond to the user's selection and perform actions accordingly. This mechanism is commonly used to create interactive interfaces where different functions are triggered based on user interactions with GUI elements.

- Step 4 Run `gui_cervical.py` and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **Logistic Regression** item from `cbClassifier` widget. Then, you will see the result as shown in Figure 135.

Click on **Norm** radio button. Then, choose **Logistic Regression** item from `cbClassifier` widget. Then, you will see the result as shown in Figure 136.

Click on **Stand** radio button. Then, choose **Logistic Regression** item from `cbClassifier` widget. Then, you will see the result as shown in Figure 137.

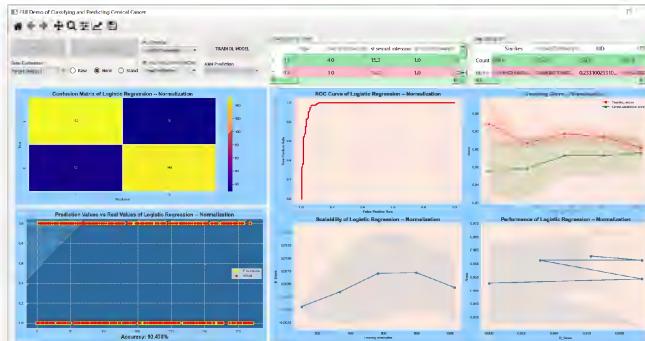


Figure 136 The result using LR model with normalization feature scaling

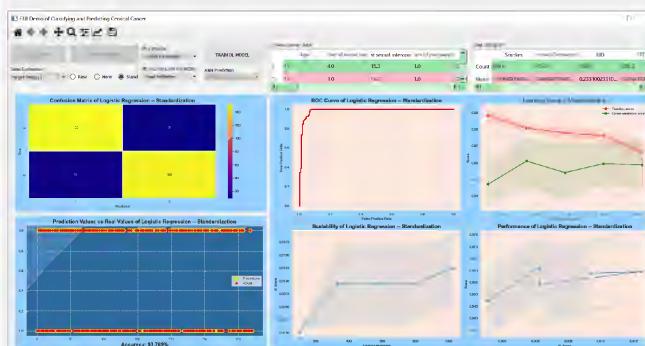


Figure 137 The result using LR model with standardization feature scaling

Support Vector Classifier

Step 1 Define **build_train_svm()** method to build and train Support Vector Machine (SVM) classifier using three feature scaling: Raw, Normalization, and Standardization:

```

1 def build_train_svm(self):
2     if path.isfile('SVMRaw.pkl'):
3         #Loads model
4         self.SVMRaw = joblib.load('SVMRaw.pkl')
5         self.SVMNorm = joblib.load('SVMNorm.pkl')
6         self.SVMStand = joblib.load('SVMStand.pkl')
7
8     if self.rbRaw.isChecked():
9         self.run_model('Support Vector Machine', 'Raw', \
10             self.SVMRaw, self.X_train_raw, self.X_test_raw, \
11             self.y_train_raw, self.y_test_raw)
12     if self.rbNorm.isChecked():
13         self.run_model('Support Vector Machine', \
14             'Normalization', self.SVMNorm, \
15             self.X_train_norm, self.X_test_norm, \
16             self.y_train_norm, self.y_test_norm)
17
18     if self.rbStand.isChecked():

```

```

19 self.run_model('Support Vector Machine', \
20 'Standardization', self.SVMStand, \
21 self.X_train_stand, self.X_test_stand, \
22 self.y_train_stand, self.y_test_stand)
23 else:
24 #Builds and trains Logistic Regression
25 self.SVMRaw =
26 SVC(random_state=2021,probability=True)
27 self.SVMNorm =
28 SVC(random_state=2021,probability=True)
29 self.SVMStand =
30 SVC(random_state=2021,probability=True)
31
32 if self.rbRaw.isChecked():
33 self.run_model('Support Vector Machine', 'Raw', \
34 self.SVMRaw, self.X_train_raw, self.X_test_raw, \
35 self.y_train_raw, self.y_test_raw)
36 if self.rbNorm.isChecked():
37 self.run_model('Support Vector Machine', \
38 'Normalization', self.SVMNorm, \
39 self.X_train_norm, self.X_test_norm, \
40 self.y_train_norm, self.y_test_norm)
41
42 if self.rbStand.isChecked():
43 self.run_model('Support Vector Machine', \
44 'Standardization', self.SVMStand, \
45 self.X_train_stand, self.X_test_stand, \
46 self.y_train_stand, self.y_test_stand)
47
48 #Saves model
49 joblib.dump(self.SVMRaw, 'SVMRaw.pkl')
50 joblib.dump(self.SVMNorm, 'SVMNorm.pkl')
joblib.dump(self.SVMStand, 'SVMStand.pkl')

```

The code block is responsible for building, training, and saving Support Vector Machine (SVM) models for predicting cervical cancer based on different preprocessing methods (raw data, normalization, and standardization). If the saved models are already available, it loads them; otherwise, it creates new SVM models, trains them, and saves them for each preprocessing scenario.

Here's a brief explanation of the code:

1. Load or Create SVM Models: The code checks if pre-trained SVM models are available ('SVMRaw.pkl', 'SVMNorm.pkl', 'SVMStand.pkl'). If they exist, it loads them into the respective variables (self.SVMRaw, self.SVMNorm, self.SVMStand). If not, new SVM models are created for each preprocessing scenario using the SVC class

with random state and probability set to ensure reproducibility.

2. Run Model and Evaluate: For each preprocessing scenario, it uses the run_model function to train and evaluate the SVM model. The run_model function calculates various performance metrics (accuracy, recall, precision, F1-score) and plots confusion matrices, prediction values versus real values, ROC curves, learning curves, scalability curves, and performance curves.
3. Save Models: After training and evaluating the SVM models, it saves the trained models for each preprocessing scenario using the joblib.dump function, creating the files 'SVMRaw.pkl', 'SVMNorm.pkl', and 'SVMStand.pkl'.

Overall, this code block encapsulates the process of building, training, evaluating, and saving SVM models for cervical cancer prediction, allowing for easy comparison of performance under different preprocessing techniques.

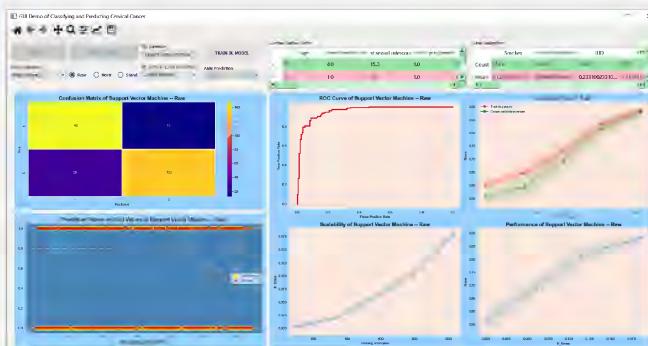


Figure 138 The result using SVM model with raw feature scaling

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'Support Vector Machine':  
2     self.build_train_svm()
```

Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **Support Vector Machine** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 138. Click on **Norm** radio button. Then, choose **Support Vector Machine** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 139.


```

15     self.X_train_norm, self.X_test_norm, \
16     self.y_train_norm, self.y_test_norm)
17
18 if self.rbStand.isChecked():
19     self.run_model('K-Nearest Neighbor', \
20     'Standardization', self.KNNStand, \
21     self.X_train_stand, self.X_test_stand, \
22     self.y_train_stand, self.y_test_stand)
23
24 else:
25     #Builds and trains K-Nearest Neighbor
26     self.KNNRaw = KNeighborsClassifier(n_neighbors
27     = 50)
28     self.KNNNorm =
29     KNeighborsClassifier(n_neighbors = 50)
30     self.KNNStand =
31     KNeighborsClassifier(n_neighbors = 50)
32
33 if self.rbRaw.isChecked():
34     self.run_model('K-Nearest Neighbor', 'Raw', \
35     self.KNNRaw, self.X_train_raw, \
36     self.X_test_raw, self.y_train_raw, \
37     self.y_test_raw)
38 if self.rbNorm.isChecked():
39     self.run_model('K-Nearest Neighbor', \
40     'Normalization', self.KNNNorm, \
41     self.X_train_norm, self.X_test_norm, \
42     self.y_train_norm, self.y_test_norm)
43
44 if self.rbStand.isChecked():
45     self.run_model('K-Nearest Neighbor', \
46     'Standardization', self.KNNStand, \
47     self.X_train_stand, self.X_test_stand, \
48     self.y_train_stand, self.y_test_stand)
49 #Saves model
50     joblib.dump(self.KNNRaw, 'KNNRaw.pkl')
51     joblib.dump(self.KNNNorm, 'KNNNorm.pkl')
52     joblib.dump(self.KNNStand, 'KNNStand.pkl')

```

The code segment focuses on constructing, training, and storing K-Nearest Neighbor (KNN) models for the prediction of cervical cancer, employing various preprocessing strategies such as raw data, normalization, and standardization.

Firstly, the code checks whether pre-existing KNN models are available (e.g., 'KNNRaw.pkl'). If so, it loads these models, otherwise, it creates fresh instances of KNN models for each preprocessing method, setting the number of neighbors to 50.

Next, the script proceeds to execute and assess the KNN models. For each preprocessing approach, the `run_model` function is invoked, facilitating the training and evaluation of

the KNN model. This function gauges performance metrics encompassing accuracy, recall, precision, and F1-score. Additionally, it generates visualizations such as confusion matrices, ROC curves, prediction versus actual value plots, learning curves, scalability curves, and performance curves.

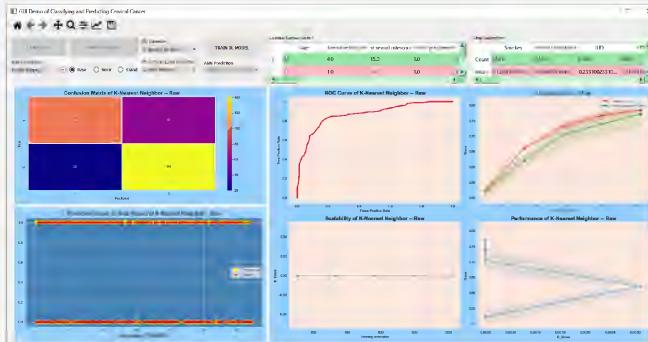


Figure 141 The result using KNN model with raw feature scaling

Upon the completion of model training and evaluation, the code takes a step to store the trained KNN models. It utilizes the joblib.dump function to save the models, resulting in the creation of files like 'KNNRaw.pkl' and 'KNNNorm.pkl'.

In essence, this code section holistically manages the lifecycle of KNN models, encompassing creation, training, evaluation, and storage, and is congruent with the procedural approach followed for other classifier models.

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'K-Nearest Neighbor':
2     self.build_train_knn()
```

Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **K-Nearest Neighbor** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 141.

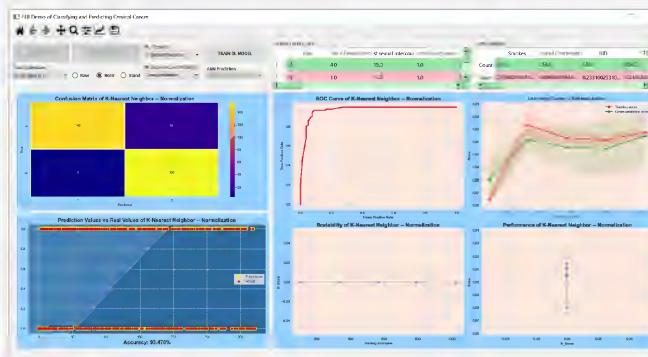


Figure 142 The result using KNN model with normalization feature scaling

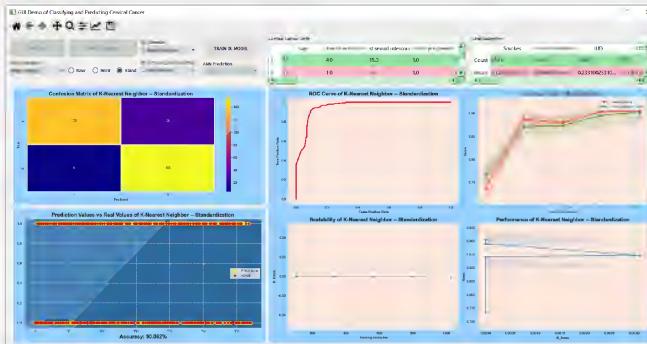


Figure 143 The result using KNN model with standardization feature scaling

Click on **Norm** radio button. Then, choose **K-Nearest Neighbor** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 142.

Click on **Stand** radio button. Then, choose **K-Nearest Neighbor** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 143.

Decision Trees Classifier

Step 1

Define **build_train_dt()** method to build and train Decision Tree (DT) classifier using three feature scaling: Raw, Normalization, and Standardization:

```
1 def build_train_dt(self):
2     if path.isfile('DTRaw.pkl'):
3         #Loads model
4         self.DTRaw = joblib.load('DTRaw.pkl')
5         self.DTNorm = joblib.load('DTNorm.pkl')
6         self.DTStand = joblib.load('DTStand.pkl')
7
8     if self.rbRaw.isChecked():
9         self.run_model('Decision Tree', \
10             'Raw', self.DTRaw, self.X_train_raw, \
11             self.X_test_raw, self.y_train_raw, self.y_test_raw)
12     if self.rbNorm.isChecked():
13         self.run_model('Decision Tree', \
14             'Normalization', self.DTNorm, \
15             self.X_train_norm, self.X_test_norm, \
16             self.y_train_norm, self.y_test_norm)
17
18     if self.rbStand.isChecked():
19         self.run_model('Decision Tree', \
20             'Standardization', self.DTStand, \
21             self.X_train_stand, self.X_test_stand, \
22             self.y_train_stand, self.y_test_stand)
23
24 else:
25     #Builds and trains Decision Tree
26     dt = DecisionTreeClassifier()
27     parameters = { 'max_depth':np.arange(1,5,1),\
28         'random_state':[2021]}
29     self.DTRaw = GridSearchCV(dt, parameters)
30     self.DTNorm = GridSearchCV(dt, parameters)
31     self.DTStand = GridSearchCV(dt, parameters)
32
33     if self.rbRaw.isChecked():
34         self.run_model('Decision Tree', \
35             'Raw', self.DTRaw, self.X_train_raw, \
36             self.X_test_raw, self.y_train_raw, \
37             self.y_test_raw)
38     if self.rbNorm.isChecked():
39         self.run_model('Decision Tree', \
40             'Normalization', self.DTNorm, \
41             self.X_train_norm, self.X_test_norm, \
42             self.y_train_norm, self.y_test_norm)
43
44     if self.rbStand.isChecked():
45         self.run_model('Decision Tree', \
46             'Standardization', self.DTStand, \
47             self.X_train_stand, self.X_test_stand, \
48             self.y_train_stand, self.y_test_stand)
49
```

```

50 #Saves model
51 joblib.dump(self.DTRaw, 'DTRaw.pkl')
52 joblib.dump(self.DTNorm, 'DTNorm.pkl')
53 joblib.dump(self.DTStand, 'DTStand.pkl')
54
55

```

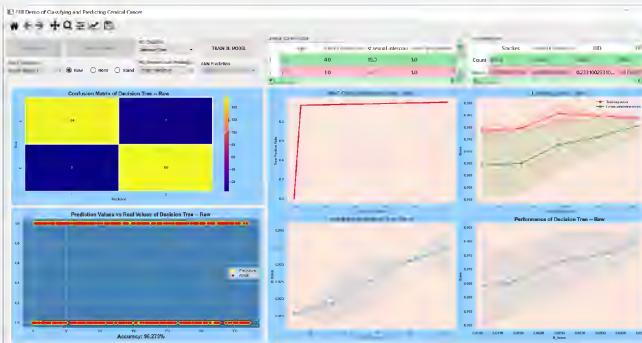


Figure 144 The result using DT model with raw feature scaling

The code is part of program designed for building, training, and evaluating Decision Tree models for predicting cervical cancer. It employs various preprocessing techniques, including using raw data, normalization, and standardization. Let's explore the purpose of this code in detail:

1. Conditional Model Loading: The code begins by checking whether pre-trained Decision Tree models already exist, indicated by the presence of files like 'DTRaw.pkl'. If these files are found, the models are loaded into memory, avoiding redundant training.
2. Model Execution and Evaluation: If pre-trained models do not exist, the code creates instances of Decision Tree classifiers (dt) and sets up hyperparameter search using GridSearchCV. This search optimizes the max_depth parameter within a specified range, aiming to improve model performance. The code then executes the model training and evaluation process for each preprocessing method (raw, normalization, and standardization) based on the user's selected options. The run_model function is used to train the model, assess its performance using various metrics (accuracy, recall, precision, F1-score), and visualize results (confusion

- matrices, ROC curves, learning curves, etc.).
- 3. Model Storage: After training and evaluating the models, the code saves the trained Decision Tree models using `joblib.dump`, creating files like '`DTRaw.pkl`'. This step ensures that the trained models can be easily reused in the future without the need for retraining, saving computational time and resources.
 - 4. Reuse of Existing Models: If pre-trained models are found, the code directly employs them for further analysis, enhancing efficiency by avoiding unnecessary retraining.
 - 5. Parameter Tuning: The use of `GridSearchCV` to optimize the `max_depth` hyperparameter demonstrates a commitment to improving model performance by exploring different parameter combinations and selecting the best one.
 - 6. Consistent Evaluation: The code ensures consistent evaluation and visualization across different preprocessing methods and model configurations, promoting fair comparisons and a thorough understanding of model behavior.
 - 7. User Interaction: The code appears to be part of a larger user interface, where users can select different preprocessing methods and classifiers, offering an interactive and user-friendly experience.
 - 8. Choice of Classifier: While the code focuses on Decision Tree models, its structure suggests that it is part of a broader framework that can accommodate various classifiers, as evident from the similarity of execution patterns for other models like Logistic Regression, Support Vector Machine, and K-Nearest Neighbor.
 - 9. Automated Workflow: The code follows a structured and automated workflow, abstracting away complex tasks such as model training, hyperparameter tuning, and performance evaluation, which are essential for rapid experimentation and analysis.
 - 10. Reproducibility: The code emphasizes model storage and reuse, enhancing

reproducibility by enabling other researchers to replicate experiments and verify results.

In summary, this code segment plays a pivotal role in a comprehensive machine learning pipeline, offering a systematic approach to building, training, evaluating, and storing Decision Tree models for cervical cancer prediction while accommodating various preprocessing techniques.

Step 2

Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'Decision Tree':  
2     self.build_train_dt()
```

Step 3

Run **gui_cervical.py** and click **LOAD DATA** and **ML MODEL** buttons. Click on **Raw** radio button and choose **Decision Tree** item from **cbClassifier** widget. You will see the result as shown in Figure 144.

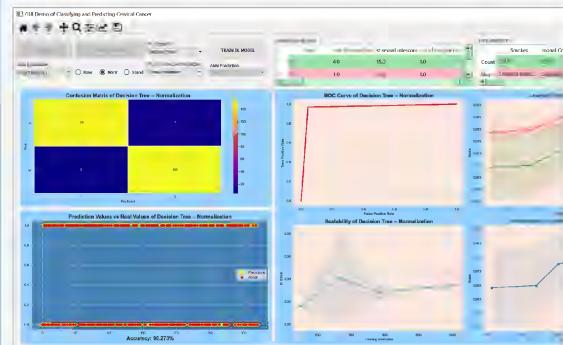


Figure 145 The result using DT model with normalized feature scaling

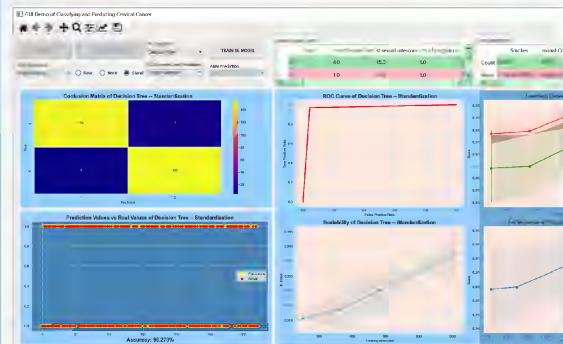


Figure 146 The result using DT model with standard feature scaling

Click on **Norm** radio button. Then, choose **Decision Tree** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 145.

Click on **Stand** radio button. Then, choose **Decis** item from **cbClassifier** widget. Then, you will see as shown in Figure 146.

Random Forest Classifier

- Step 1 Define **build_train_rf()** method to build and train Random Forest (RF) classifier using three feature scaling: Raw, Normalization, and Standardization:

```
1 def build_train_rf(self):
2     if path.isfile('RFRaw.pkl'):
3         #Loads model
4         self.RFRaw = joblib.load('RFRaw.pkl')
5         self.RFNorm = joblib.load('RFNorm.pkl')
6         self.RFStand = joblib.load('RFStand.pkl')
7
8     if self.rbRaw.isChecked():
9         self.run_model('Random Forest', 'Raw', \
10             self.RFRaw, self.X_train_raw, self.X_test_raw, \
11             self.y_train_raw, self.y_test_raw)
12     if self.rbNorm.isChecked():
13         self.run_model('Random Forest', \
14             'Normalization', self.RFNorm, \
15             self.X_train_norm, self.X_test_norm, \
16             self.y_train_norm, self.y_test_norm)
17
18     if self.rbStand.isChecked():
19         self.run_model('Random Forest', \
20             'Standardization', self.RFStand, \
21             self.X_train_stand, self.X_test_stand, \
22             self.y_train_stand, self.y_test_stand)
23
24 else:
25     #Builds and trains Random Forest
26     self.RFRaw =
27     RandomForestClassifier(n_estimators=200, \
28             max_depth=2, random_state=2021)
29     self.RFNorm = RandomForestClassifier(\
30             n_estimators=200, max_depth=2,
31             random_state=2021)
32     self.RFStand = RandomForestClassifier(\
33             n_estimators=200, max_depth=2,
34             random_state=2021)
35
36     if self.rbRaw.isChecked():
37         self.run_model('Random Forest', 'Raw', \
38             self.RFRaw, self.X_train_raw, \
39             self.X_test_raw, self.y_train_raw, \
40             self.y_test_raw)
41     if self.rbNorm.isChecked():
42         self.run_model('Random Forest', \
```

```

43     'Normalization', self.RFNorm, \
44     self.X_train_norm, self.X_test_norm, \
45     self.y_train_norm, self.y_test_norm)
46
47 if self.rbStand.isChecked():
48     self.run_model('Random Forest', \
49     'Standardization', self.RFStand, \
50     self.X_train_stand, self.X_test_stand, \
51     self.y_train_stand, self.y_test_stand)
52
53 #Saves model
54     joblib.dump(self.RFRaw, 'RFRaw.pkl')
55     joblib.dump(self.RFNorm, 'RFNorm.pkl')
      joblib.dump(self.RFStand, 'RFStand.pkl')

```

The code segment serves the purpose of building, training, and evaluating Random Forest models for predicting cervical cancer. It encapsulates an organized approach for both creating new models and reusing existing ones. Let's delve into its detailed purpose:

1. Model Loading and Reuse: The code begins by checking if pre-trained Random Forest models exist (e.g., 'RFRaw.pkl'). If found, it loads these models into memory (self.RFRaw, self.RFNorm, self.RFStand). This process ensures efficient utilization of previously trained models and facilitates comparison between different preprocessing methods.
2. Model Execution and Evaluation: In the absence of pre-trained models, the code creates instances of the RandomForestClassifier with specific hyperparameters (n_estimators=200, max_depth=2, random_state=2021). It employs the run_model function to train and evaluate the model using different preprocessing techniques (raw, normalization, standardization) based on user selections. The metrics for model evaluation include accuracy, recall, precision, F1-score, and classification report. The code also visualizes key results, such as confusion matrices and ROC curves.
3. Hyperparameter Variation: While building new models, the code initializes multiple Random Forest models with different preprocessing methods, promoting comparison and exploration of how preprocessing affects model performance.

4. Consistent Analysis: By following a structured approach and employing similar execution patterns for different models and preprocessing methods, the code ensures a consistent analysis process. This consistency enables accurate comparison and understanding of each model's behavior.
5. Saves Trained Models: After training and evaluating the models, the code saves them using joblib.dump (e.g., 'RFRaw.pkl'). This step ensures that trained models can be reused for future predictions without repeating the training process.
6. User Interaction: The code is part of an interactive user interface where users can select preprocessing methods and classifiers. This user-friendly design facilitates experimentation and exploration of various model configurations.
7. Optimized Random Forest Configuration: For the new models, the code specifies a fixed number of decision trees (`n_estimators=200`) and a limited maximum depth (`max_depth=2`). This configuration can lead to less complex models, mitigating overfitting and promoting generalization.
8. Enhancing Code Efficiency: By checking for pre-existing models and loading them, the code optimizes resource usage by avoiding unnecessary retraining, reducing computational costs, and speeding up the evaluation process.
9. Model Diversity: The code's ability to handle various preprocessing methods (raw, normalization, standardization) and its consistent evaluation process contribute to a comprehensive understanding of Random Forest model performance under different conditions.
10. Reproducibility and Long-term Usage: By saving trained models, the code promotes reproducibility, allowing researchers to use the same models for consistent results in future analyses, as well as facilitating model sharing and collaboration.

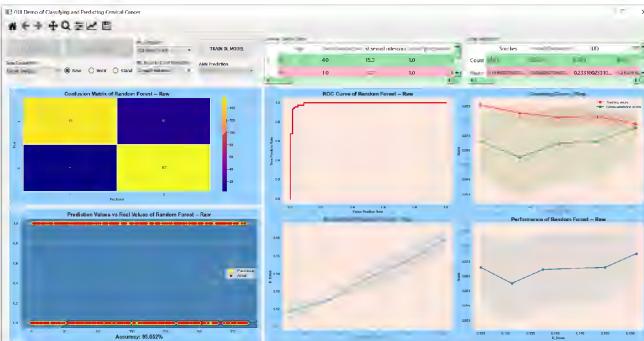


Figure 147 The result using RF model with raw feature scaling

In summary, the code aims to provide a systematic and efficient approach to building, training, and evaluating Random Forest models for cervical cancer prediction. It combines user interaction, model reusability, evaluation consistency, and visualization to create a valuable tool for researchers and practitioners in the field of machine learning and healthcare.

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'Random Forest':
2     self.build_train_rf()
```

Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **Random Forest** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 147.

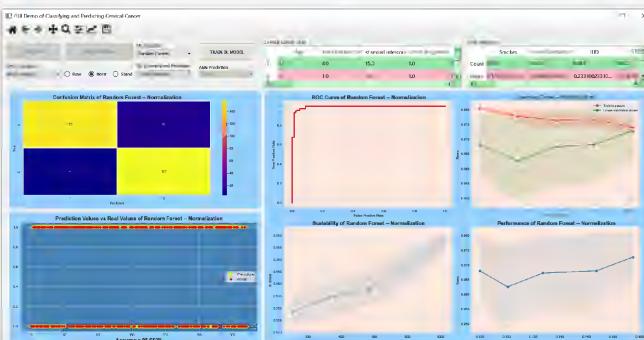


Figure 148 The result using RF model with normalization feature scaling

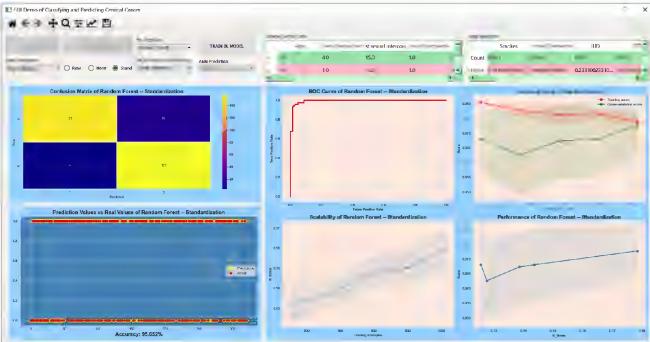


Figure 149 The result using RF model with standardization feature scaling

Click on **Norm** radio button. Then, choose **Random Forest** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 148.

Click on **Stand** radio button. Then, choose **Random Forest** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 149.

Gradient Boosting Classifier

Step 1 Define **build_train_gb()** method to build and train Gradient Boosting (GB) classifier using three feature scaling: Raw, Normalization, and Standardization:

```

1 def build_train_gb(self):
2     if path.isfile('GBRaw.pkl'):
3         #Loads model
4         self.GBRaw = joblib.load('GBRaw.pkl')
5         self.GBNorm = joblib.load('GBNorm.pkl')
6         self.GBStand = joblib.load('GBStand.pkl')
7
8         if self.rbRaw.isChecked():
9             self.run_model('Gradient Boosting', 'Raw', \
10                         self.GBRaw, self.X_train_raw, \
11                         self.X_test_raw, self.y_train_raw, \
12                         self.y_test_raw)
13         if self.rbNorm.isChecked():
14             self.run_model('Gradient Boosting', \
15                         'Normalization', self.GBNorm, \
16                         self.X_train_norm, self.X_test_norm, \
17                         self.y_train_norm, self.y_test_norm)
18
19         if self.rbStand.isChecked():
20             self.run_model('Gradient Boosting', \
21                         'Standardization', self.GBStand, \
22                         self.X_train_stand, self.X_test_stand, \
23                         self.y_train_stand, self.y_test_stand)
24

```

```

25     else:
26         #Builds and trains Gradient Boosting
27         self.GBRaw = GradientBoostingClassifier(
28             n_estimators = 200, max_depth=3,
29             subsample=0.8, \
30                 max_features=0.2, random_state=2021)
31         self.GBNorm = GradientBoostingClassifier(
32             n_estimators = 200, max_depth=3,
33             subsample=0.8, \
34                 max_features=0.2, random_state=2021)
35         self.GBStand = GradientBoostingClassifier(
36             n_estimators = 200, max_depth=3,
37             subsample=0.8, \
38                 max_features=0.2, random_state=2021)
39
40     if self.rbRaw.isChecked():
41         self.run_model('Gradient Boosting', 'Raw', \
42             self.GBRaw, self.X_train_raw, \
43             self.X_test_raw, self.y_train_raw, \
44             self.y_test_raw)
45     if self.rbNorm.isChecked():
46         self.run_model('Gradient Boosting', \
47             'Normalization', self.GBNorm, \
48             self.X_train_norm, self.X_test_norm, \
49             self.y_train_norm, self.y_test_norm)
50
51     if self.rbStand.isChecked():
52         self.run_model('Gradient Boosting', \
53             'Standardization', self.GBStand, \
54             self.X_train_stand, self.X_test_stand, \
55             self.y_train_stand, self.y_test_stand)
56
57     #Saves model
58     joblib.dump(self.GBRaw, 'GBRaw.pkl')
59     joblib.dump(self.GBNorm, 'GBNorm.pkl')
60     joblib.dump(self.GBStand, 'GBStand.pkl')

```

The code segment is responsible for building, training, and evaluating Gradient Boosting models for the prediction of cervical cancer. It follows a similar pattern to the previously discussed code blocks. Let's break down its detailed purpose:

1. Model Loading and Reuse: The code starts by checking if pre-trained Gradient Boosting models exist (e.g., 'GBRaw.pkl'). If these models are found, they are loaded into memory (self.GBRaw, self.GBNorm, self.GBStand), facilitating efficient use of previously trained models.
2. Model Execution and Evaluation: In the absence of pre-trained models, the code creates instances of GradientBoostingClassifier with specific

hyperparameters (n_estimators=200, max_depth=3, subsample=0.8, max_features=0.2, random_state=2021). Similar to previous code blocks, it employs the run_model function to train and evaluate the model using different preprocessing techniques (raw, normalization, standardization) based on user selections. Evaluation metrics include accuracy, recall, precision, F1-score, and a classification report. The code also visualizes results such as confusion matrices and ROC curves.

3. Hyperparameter Variation: The code's creation of new models encompasses variations in preprocessing techniques, which allows for thorough comparison and exploration of model performance under different data conditions.
4. Consistent Analysis: By maintaining a structured approach and employing a consistent execution pattern, the code ensures reliable analysis of Gradient Boosting model behavior across various preprocessing methods.
5. Model Saving: After training and evaluating the models, the code saves them using joblib.dump (e.g., 'GBRaw.pkl'). This step enables the reuse of trained models for future predictions without needing to retrain, promoting efficiency and reproducibility.
6. User Interaction: Like previous code segments, this code is designed for a user interface where users can select preprocessing methods and classifiers, enhancing user-friendliness and facilitating experimentation.
7. Optimized Gradient Boosting Configuration: The code configures the Gradient Boosting models with specified hyperparameters for improved performance, including a moderate number of estimators, limited maximum depth, and subsampling.
8. Comprehensive Evaluation: The code evaluates models using multiple metrics and visualizations, helping users gain a comprehensive understanding of how Gradient Boosting performs with different preprocessing strategies.

9. Efficient Reuse of Resources: By loading pre-existing models, the code optimizes resource usage and minimizes redundant computations, reducing the time and computational resources required for analysis.
10. Long-term Usability: Saving trained models ensures long-term usability and consistent results in future analyses, making the code valuable for ongoing research and prediction tasks.

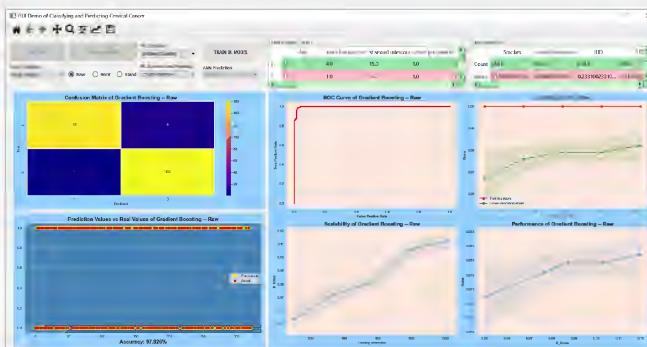


Figure 150 The result using GB model with raw feature scaling

In summary, this code segment provides a systematic approach to building, training, and evaluating Gradient Boosting models for cervical cancer prediction. Through its consistent structure, user-friendly interface, comprehensive evaluation, and resource-efficient practices, the code contributes to the exploration and understanding of Gradient Boosting's performance under different preprocessing scenarios.

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'Gradient Boosting':
2     self.build_train_gb()
```

Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **Gradient Boosting** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 150.

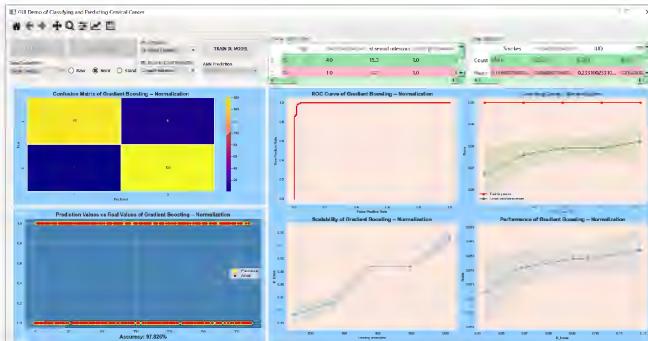


Figure 151 The result using GB model with normalization feature scaling

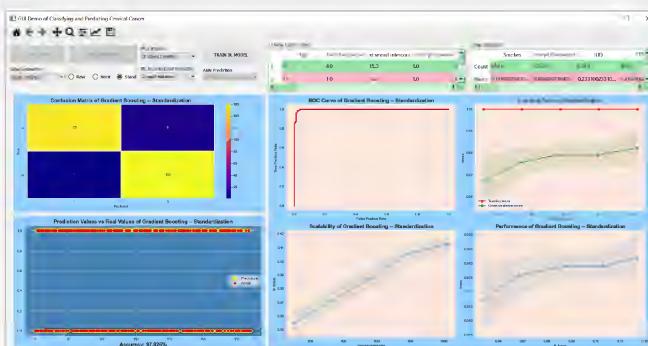


Figure 152 The result using GB model with standardization feature scaling

Click on **Norm** radio button. Then, choose **Gradient Boosting** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 151.

Click on **Stand** radio button. Then, choose **Gradient Boosting** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 152.

Naïve Bayes Classifier

Step 1 Define **build_train_nb()** method to build and train Naïve Bayes (NB) classifier using three feature scaling: Raw, Normalization, and Standardization:

```

1 def build_train_nb(self):
2     if path.isfile('NBRaw.pkl'):
3         #Loads model
4         self.NBRaw = joblib.load('NBRaw.pkl')
5         self.NBNorm = joblib.load('NBNorm.pkl')
6         self.NBStand = joblib.load('NBStand.pkl')
7
8     if self.rbRaw.isChecked():
9         self.run_model('Naive Bayes', 'Raw', \
10             self.NBRaw, self.X_train_raw, \

```

```

11 self.X_test_raw, self.y_train_raw, \
12 self.y_test_raw)
13 if self.rbNorm.isChecked():
14     self.run_model('Naive Bayes', 'Normalization', \
15     self.NBNorm, self.X_train_norm, \
16     self.X_test_norm, self.y_train_norm, \
17     self.y_test_norm)
18
19 if self.rbStand.isChecked():
20     self.run_model('Naive Bayes', \
21     'Standardization', self.NBStand, \
22     self.X_train_stand, self.X_test_stand, \
23     self.y_train_stand, self.y_test_stand)
24
25 else:
26     #Builds and trains Naive Bayes
27     self.NBRaw = GaussianNB()
28     self.NBNorm = GaussianNB()
29     self.NBStand = GaussianNB()
30
31 if self.rbRaw.isChecked():
32     self.run_model('Naive Bayes', 'Raw', \
33     self.NBRaw, self.X_train_raw, \
34     self.X_test_raw, self.y_train_raw, \
35     self.y_test_raw)
36 if self.rbNorm.isChecked():
37     self.run_model('Naive Bayes', \
38     'Normalization', self.NBNorm, \
39     self.X_train_norm, self.X_test_norm, \
40     self.y_train_norm, self.y_test_norm)
41
42 if self.rbStand.isChecked():
43     self.run_model('Naive Bayes', \
44     'Standardization', self.NBStand, \
45     self.X_train_stand, self.X_test_stand, \
46     self.y_train_stand, self.y_test_stand)
47
48 #Saves model
49     joblib.dump(self.NBRaw, 'NBRaw.pkl')
50     joblib.dump(self.NBNorm, 'NBNorm.pkl')
51     joblib.dump(self.NBStand, 'NBStand.pkl')
52
53

```

The code is focused on building, training, and evaluating Naive Bayes models for predicting cervical cancer, utilizing a consistent approach with various preprocessing methods. Here's an overview of its functionality:

1. Model Loading and Reuse: The code begins by checking if pre-trained Naive Bayes models exist (e.g., 'NBRaw.pkl'). If these models are found, they are loaded into memory (self.NBRaw, self.NBNorm,

- self.NBStand). This approach allows for the reuse of previously trained models and minimizes computational overhead.
2. Model Execution and Evaluation: In the absence of pre-trained models, the code creates instances of GaussianNB, which is a type of Naive Bayes classifier. It then uses the run_model function to train and evaluate the Naive Bayes models with different preprocessing techniques (raw, normalization, standardization) based on user selections. The evaluation includes metrics such as accuracy, recall, precision, F1-score, and a classification report.
 3. Consistent Analysis and User Interaction: Similar to other code blocks, this code maintains a structured approach to training and evaluating models, ensuring consistency across different preprocessing methods. It also provides a user-friendly interface where users can select the preferred preprocessing method and classifier for analysis.
 4. Model Saving: After training and evaluating the models, the code saves them using joblib.dump (e.g., 'NBRaw.pkl'). This step enables the retention of trained models for future predictions without the need to retrain, enhancing efficiency and reproducibility.
 5. Efficient Model Creation: By utilizing the same model architecture and hyperparameters for each preprocessing technique, the code streamlines model creation and comparison, reducing the complexity of model initialization and hyperparameter tuning.

In summary, the code's primary objective is to automate the process of building, training, and evaluating Naive Bayes models for cervical cancer prediction. Its systematic approach, combined with the ability to reuse pre-trained models and visualize results, facilitates efficient experimentation and analysis of the Naive Bayes algorithm's performance under different preprocessing scenarios. The code contributes to streamlined model development and provides insights into how Naive Bayes performs on the given dataset while considering various data preprocessing techniques.

Step 1 Add this code to the end of **choose_ML_model()** method:

2

```
1 if strCB == 'Naive Bayes':  
2     self.build_train_nb()
```

Step 2 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **Naive Bayes** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 153.

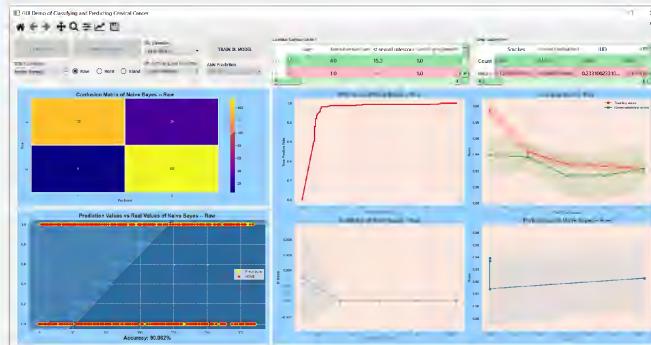


Figure 153 The result using Naïve Bayes model with raw feature scaling

Click on **Norm** radio button. Then, choose **Naive Bayes** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 154.

Click on **Stand** radio button. Then, choose **Naive Bayes** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 155.

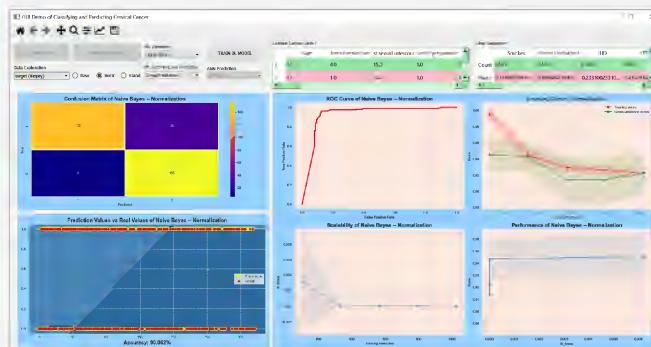


Figure 154 The result using Naïve Bayes model with normalization feature scaling

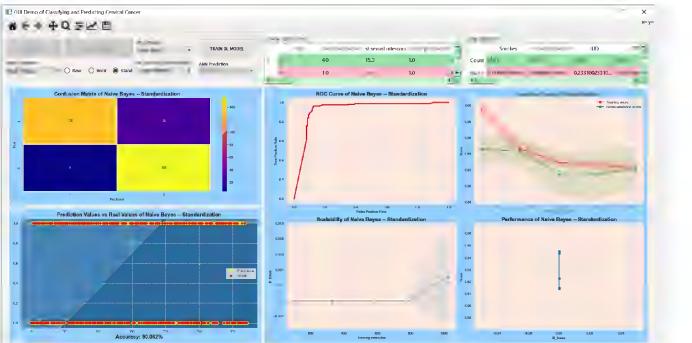


Figure 155 The result using Naïve Bayes model with standardization feature scaling

AdaBoost Classifier

Step 1 Define **build_train_ada()** method to build and train Adaboost classifier using three feature scaling: Raw, Normalization, and Standardization:

```

1  def build_train_ada(self):
2      if path.isfile('ADARaw.pkl'):
3          #Loads model
4          self.ADARaw = joblib.load('ADARaw.pkl')
5          self.ADANorm = joblib.load('ADANorm.pkl')
6          self.ADAStand = joblib.load('ADAStand.pkl')
7
8          if self.rbRaw.isChecked():
9              self.run_model('Adaboost', 'Raw', \
10                  self.ADARaw, self.X_train_raw, \
11                  self.X_test_raw, self.y_train_raw, \
12                  self.y_test_raw)
13          if self.rbNorm.isChecked():
14              self.run_model('Adaboost', 'Normalization', \
15                  self.ADANorm, self.X_train_norm, \
16                  self.X_test_norm, self.y_train_norm, \
17                  self.y_test_norm)
18
19          if self.rbStand.isChecked():
20              self.run_model('Adaboost', \
21                  'Standardization', self.ADAStand, \
22                  self.X_train_stand, self.X_test_stand, \
23                  self.y_train_stand, self.y_test_stand)
24
25      else:
26          #Builds and trains Adaboost
27          self.ADARaw = AdaBoostClassifier(\
28              n_estimators = 100, learning_rate=0.01)
29          self.ADANorm = AdaBoostClassifier(\
30              n_estimators = 100, learning_rate=0.01)
31          self.ADAStand = AdaBoostClassifier(\
32              n_estimators = 100, learning_rate=0.01)
```

```

33
34 if self.rbRaw.isChecked():
35     self.run_model('Adaboost', 'Raw', self.ADARaw, \
36     self.X_train_raw, self.X_test_raw, \
37     self.y_train_raw, self.y_test_raw)
38 if self.rbNorm.isChecked():
39     self.run_model('Adaboost', 'Normalization',\
40     self.ADANorm, self.X_train_norm, \
41     self.X_test_norm, self.y_train_norm, \
42     self.y_test_norm)
43
44 if self.rbStand.isChecked(): \
45     self.run_model('Adaboost', 'Standardization', \
46     self.ADAStand, self.X_train_stand, \
47     self.X_test_stand, self.y_train_stand, \
48     self.y_test_stand)
49
50 #Saves model
51     joblib.dump(self.ADARaw, 'ADARaw.pkl')
52     joblib.dump(self.ADANorm, 'ADANorm.pkl')
53     joblib.dump(self.ADAStand, 'ADAStand.pkl')
54
55

```

The code is designed to construct, train, and evaluate Adaboost classifiers for the purpose of predicting cervical cancer. It operates with a focus on two main scenarios:

1. Model Loading and Reuse: Initially, the code checks if previously trained Adaboost models exist, as indicated by files such as 'ADARaw.pkl'. If these models are found, they are loaded into memory as instances of the AdaboostClassifier (self.ADARaw, self.ADANorm, self.ADAStand). This approach permits the efficient reuse of pre-trained models, potentially saving computational time and resources.
2. Model Creation and Evaluation: In cases where pre-trained models are unavailable, the code proceeds to construct new AdaboostClassifier instances. These instances are created with specific hyperparameters (e.g., n_estimators, learning_rate) and are subsequently trained and evaluated using the run_model function. The evaluation process considers different preprocessing techniques, such as raw data, normalization, and standardization, based on user selections. It calculates and presents essential classification metrics, including

accuracy, recall, precision, F1-score, and classification reports.

3. Model Persistence: After completing the training and evaluation steps, the code saves the newly trained Adaboost models using `joblib.dump` (e.g., 'ADARaw.pkl'). This allows the models to be stored for future use, enabling seamless predictions without the need for retraining. By consistently following a structured approach across preprocessing methods, the code promotes efficient model development, comparison, and analysis within the Adaboost framework, and enhances reproducibility and consistency throughout the process.

Step 2 Add this code to the end of `choose_ML_model()` method:

```
1 if strCB == 'Adaboost':  
2     self.build_train_ada()
```

Step 3 Run `gui_cervical.py` and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **Adaboost** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 156.

Click on **Norm** radio button. Then, choose **Adaboost** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 157.

Click on **Stand** radio button. Then, choose **Adaboost** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 158.

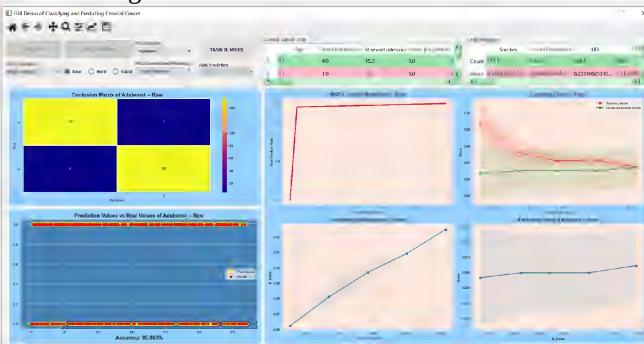


Figure 156 The result using Adaboost model with raw feature scaling

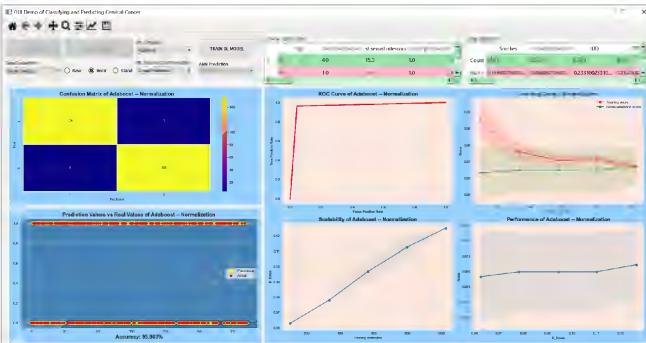


Figure 157 The result using Adaboost model with normalization feature scaling

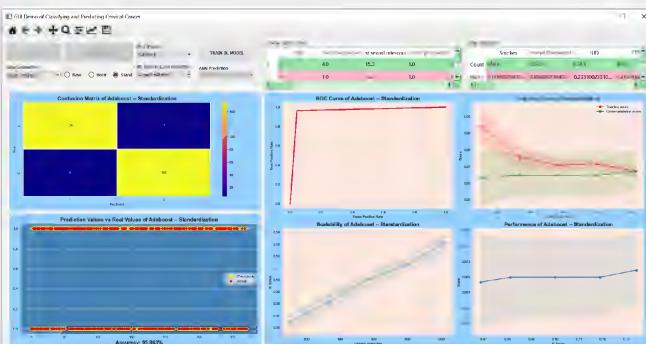


Figure 158 The result using Adaboost model with standardization feature scaling

Extreme Gradient Boosting Classifier

Step 1 Define **build_train_xgb()** method to build and train XGB classifier using three feature scaling: Raw, Normalization, and Standardization:

```

1  def build_train_xgb(self):
2      if path.isfile('XGBRaw.pkl'):
3          #Loads model
4          self.XGBRaw = joblib.load('XGBRaw.pkl')
5          self.XGBNorm = joblib.load('XGBNorm.pkl')
6          self.XGBStand = joblib.load('XGBStand.pkl')
7
8      if self.rbRaw.isChecked():
9          self.run_model('XGB', 'Raw', self.XGBRaw, \
10             self.X_train_raw, self.X_test_raw, \
11             self.y_train_raw, self.y_test_raw)
12      if self.rbNorm.isChecked():
13          self.run_model('XGB', 'Normalization', \
14             self.XGBNorm, self.X_train_norm, \
15             self.X_test_norm, self.y_train_norm, \
16             self.y_test_norm)
17
18      if self.rbStand.isChecked():

```

```

19 self.run_model('XGB', 'Standardization', \
20 self.XGBStand, self.X_train_stand, \
21 self.X_test_stand, self.y_train_stand, \
22 self.y_test_stand)
23
24 else:
25 #Builds and trains XGB classifier
26 self.XGBRaw = XGBClassifier(n_estimators = 200,
27 \
28     max_depth=2, random_state=2021, \
29     use_label_encoder=False,
30 eval_metric='mlogloss')
31 self.XGBNorm = XGBClassifier(n_estimators =
32 200, \
33     max_depth=2, random_state=2021, \
34     use_label_encoder=False,
35 eval_metric='mlogloss')
36 self.XGBStand = XGBClassifier(n_estimators =
37 200, \
38     max_depth=2, random_state=2021, \
39     use_label_encoder=False,
40 eval_metric='mlogloss')
41
42 if self.rbRaw.isChecked():
43 self.run_model('XGB', 'Raw', self.XGBRaw, \
44 self.X_train_raw, self.X_test_raw, \
45 self.y_train_raw, self.y_test_raw)
46 if self.rbNorm.isChecked():
47 self.run_model('XGB', 'Normalization', \
48 self.XGBNorm, self.X_train_norm, \
49 self.X_test_norm, self.y_train_norm, \
50 self.y_test_norm)
51
52 if self.rbStand.isChecked():
53 self.run_model('XGB', 'Standardization', \
54 self.XGBStand, self.X_train_stand, \
55 self.X_test_stand, self.y_train_stand, \
56 self.y_test_stand)
57
#Saves model
    joblib.dump(self.XGBRaw, 'XGBRaw.pkl')
    joblib.dump(self.XGBNorm, 'XGBNorm.pkl')
    joblib.dump(self.XGBStand, 'XGBStand.pkl')

```

The code segment serves the purpose of constructing, training, and evaluating XGBoost classifiers for cervical cancer prediction. It operates in two main modes:

In the first mode, the code checks for the existence of pre-trained XGBoost models stored as files like 'XGBRaw.pkl'. If these models are found, they are loaded into memory (as self.XGBRaw, self.XGBNorm, self.XGBStand) for potential reuse.

In the second mode, when pre-trained models are absent, the code creates new instances of XGBoostClassifier, customizes their hyperparameters, and proceeds to train and assess them using various preprocessing techniques (raw, normalization, and standardization). The evaluation process involves key metrics like accuracy, recall, precision, and F1-score, offering a comprehensive understanding of model performance. Once training and evaluation are complete, the models are saved using 'joblib.dump' for future access and utilization. Overall, this approach streamlines model development, facilitates comparison, and ensures model persistence while predicting cervical cancer.

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'XGB Classifier':
2     self.build_train_xgb()
```

Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **XGB Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 159.

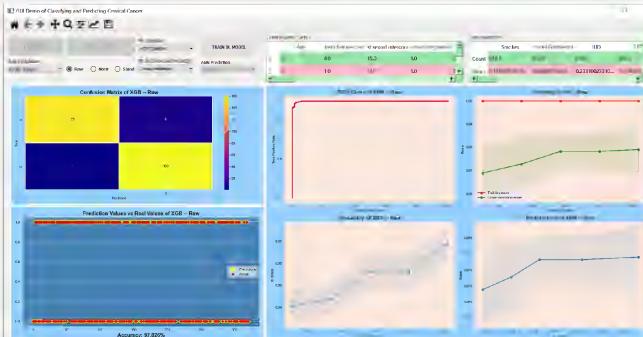


Figure 159 The result using XGB model with raw feature scaling

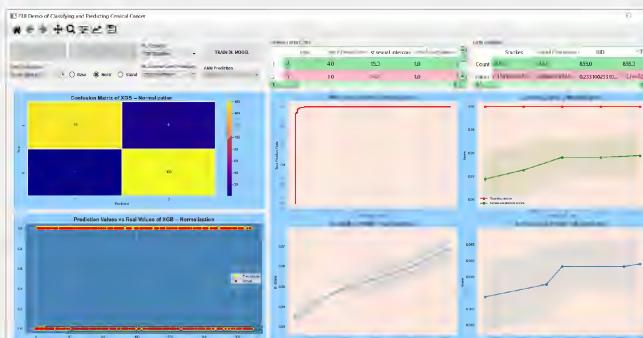


Figure 160 The result using XGB model with normalization feature scaling

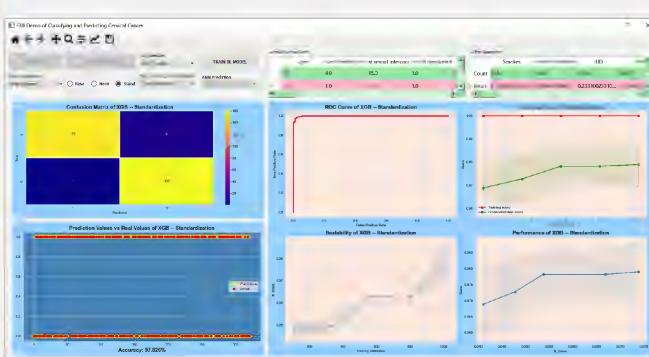


Figure 161 The result using XGB model with standardization feature scaling

Click on **Norm** radio button. Then, choose **XGB Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 160.

Click on **Stand** radio button. Then, choose **XGB Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 161.

Light Gradient Boosting Classifier

Step 1 Define **build_train_lgbm()** method to build and train LGBM classifier using three feature scaling: Raw, Normalization, and Standardization:

```

1  def build_train_lgbm(self):
2      if path.isfile('LGBMRaw.pkl'):
3          #Loads model
4          self.LGBMRaw = joblib.load('LGBMRaw.pkl')
5          self.LGBMNorm = joblib.load('LGBMNorm.pkl')
6          self.LGBMStand = joblib.load('LGBMStand.pkl')
7
8      if self.rbRaw.isChecked():
9          self.run_model('LGBM Classifier', 'Raw', \
10             self.LGBMRaw, self.X_train_raw, \
11             self.X_test_raw, self.y_train_raw, \
12             self.y_test_raw)
13     if self.rbNorm.isChecked():
14         self.run_model('LGBM Classifier', \
15             'Normalization', self.LGBMNorm, \
16             self.X_train_norm, self.X_test_norm, \
17             self.y_train_norm, self.y_test_norm)
18
19     if self.rbStand.isChecked():
20         self.run_model('LGBM Classifier', \
21             'Standardization', self.LGBMStand, \
22             self.X_train_stand, self.X_test_stand, \
23             self.y_train_stand, self.y_test_stand)
```

```

24
25     else:
26         #Builds and trains LGBMClassifier classifier
27         self.LGBMRaw = LGBMClassifier(max_depth = 2,
28         \
29             n_estimators=500, subsample=0.8,
30             random_state=2021)
31         self.LGBMNorm = LGBMClassifier(max_depth =
32         2, \
33             n_estimators=500, subsample=0.8,
34             random_state=2021)
35         self.LGBMStand = LGBMClassifier(max_depth =
36         2, \
37             n_estimators=500, subsample=0.8,
38             random_state=2021)
39
40     if self.rbRaw.isChecked():
41         self.run_model('LGBM Classifier', 'Raw', \
42             self.LGBMRaw, self.X_train_raw, \
43             self.X_test_raw, self.y_train_raw, \
44             self.y_test_raw)
45     if self.rbNorm.isChecked():
46         self.run_model('LGBM Classifier', \
47             'Normalization', self.LGBMNorm, \
48             self.X_train_norm, self.X_test_norm, \
49             self.y_train_norm, self.y_test_norm)
50
51     if self.rbStand.isChecked():
52         self.run_model('LGBM Classifier', \
53             'Standardization', self.LGBMStand, \
54             self.X_train_stand, self.X_test_stand, \
55             self.y_train_stand, self.y_test_stand)
56
#Saves model
    joblib.dump(self.LGBMRaw, 'LGBMRaw.pkl')
    joblib.dump(self.LGBMNorm,
'LGBMNorm.pkl')
    joblib.dump(self.LGBMStand,
'LGBMStand.pkl')

```

The code segment is dedicated to constructing, training, and evaluating LightGBM (LGBM) classifiers for the prediction of cervical cancer. It follows a similar two-mode process:

In the first mode, the code checks for the presence of pre-trained LGBM models stored as files like 'LGBMRaw.pkl'. If these models are found, they are loaded into memory (as self.LGBMRaw, self.LGBMNorm, self.LGBMStand) for potential reuse.

In the second mode, when pre-trained models are unavailable, the code initializes new instances of LGBMClassifier. It customizes hyperparameters such as

`max_depth`, `n_estimators`, and `subsample` for better performance. These classifiers are then trained and evaluated using different preprocessing techniques (raw, normalization, and standardization). Similar to previous sections, the evaluation encompasses crucial metrics like accuracy, recall, precision, and F1-score. Following this, the trained models are saved using '`joblib.dump`' for future access and application.

In summary, this code streamlines the development, training, and evaluation of LGBM classifiers, providing a robust approach to predict cervical cancer. It ensures model reusability and persistence while accommodating various preprocessing strategies for enhanced performance.

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'LGBM Classifier':  
2     self.build_train_lgbm()
```

Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **LGBM Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 162.

Click on **Norm** radio button. Then, choose **LGBM Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 163.

Click on **Stand** radio button. Then, choose **LGBM Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 164.

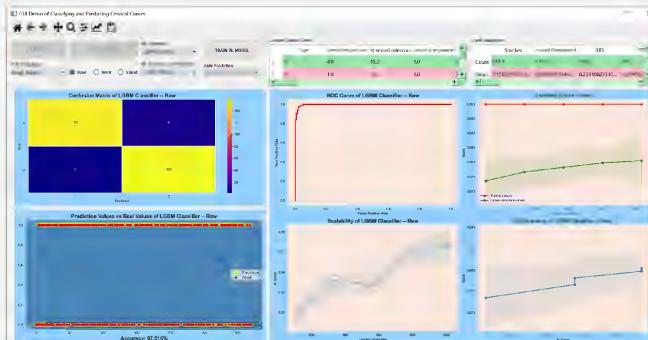


Figure 162 The result using LGBM model with raw feature scaling

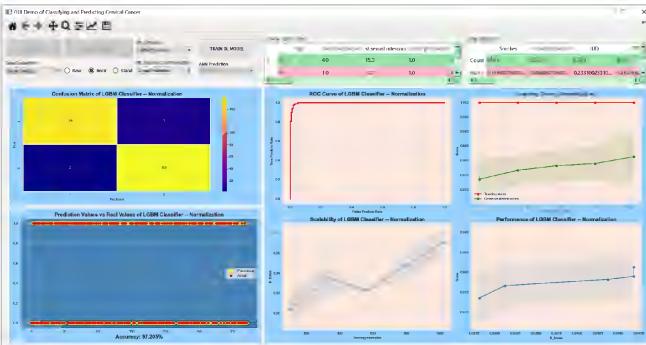


Figure 163 The result using LGBM model with normalization feature scaling

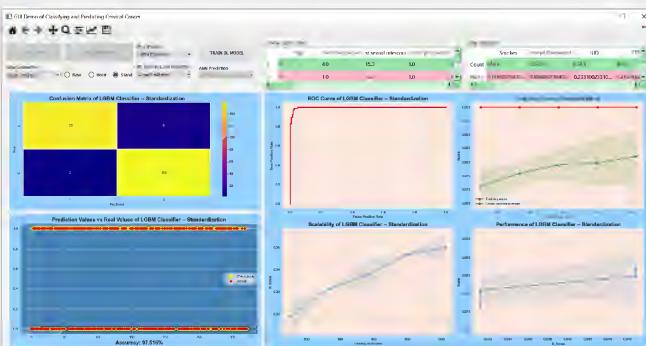


Figure 166 The result using LGBM model with standardization feature scaling

Multi-Layer Perceptron Classifier

Step 1 Define **build_train_mlp()** method to build and train LGBM classifier using three feature scaling: Raw, Normalization, and Standardization:

```

1 def build_train_mlp(self):
2     if path.isfile('MLPRaw.pkl'):
3         #Loads model
4         self.MLPRaw = joblib.load('MLPRaw.pkl')
5         self.MLPNorm = joblib.load('MLPNorm.pkl')
6         self.MLPStand = joblib.load('MLPStand.pkl')
7
8     if self.rbRaw.isChecked():
9         self.run_model('MLP Classifier', 'Raw', \
10                     self.MLPRaw, self.X_train_raw, \
11                     self.X_test_raw, self.y_train_raw, \
12                     self.y_test_raw)
13     if self.rbNorm.isChecked():
14         self.run_model('MLP Classifier', \
15                     'Normalization', self.MLPNorm, \
16                     self.X_train_norm, self.X_test_norm, \
17                     self.y_train_norm, self.y_test_norm)
18

```

```

19 if self.rbStand.isChecked():
20     self.run_model('MLP Classifier', \
21     'Standardization', self.MLPStand, \
22     self.X_train_stand, self.X_test_stand, \
23     self.y_train_stand, self.y_test_stand)
24
25 else:
26     #Builds and trains MLP classifier
27     self.MLPRaw = MLPClassifier(random_state=2021)
28     self.MLPNorm =
29     MLPClassifier(random_state=2021)
30     self.MLPStand =
31     MLPClassifier(random_state=2021)
32
33 if self.rbRaw.isChecked():
34     self.run_model('MLP Classifier', 'Raw', \
35     self.MLPRaw, self.X_train_raw, self.X_test_raw, \
36     self.y_train_raw, self.y_test_raw)
37 if self.rbNorm.isChecked():
38     self.run_model('MLP Classifier', \
39     'Normalization', self.MLPNorm, \
40     self.X_train_norm, self.X_test_norm, \
41     self.y_train_norm, self.y_test_norm)
42
43 if self.rbStand.isChecked():
44     self.run_model('MLP Classifier', \
45     'Standardization', self.MLPStand, \
46     self.X_train_stand, self.X_test_stand, \
47     self.y_train_stand, self.y_test_stand)
48
49 #Saves model
50     joblib.dump(self.MLPRaw, 'MLPRaw.pkl')
51     joblib.dump(self.MLPNorm, 'MLPNorm.pkl')
52     joblib.dump(self.MLPStand, 'MLPStand.pkl')

```

The code segment focuses on constructing, training, and evaluating Multi-Layer Perceptron (MLP) classifiers for predicting cervical cancer. It follows a similar pattern as seen in previous sections:

In the first mode, the code checks for the presence of pre-trained MLP models saved as files like 'MLPRaw.pkl'. If these models exist, they are loaded into memory (self.MLPRaw, self.MLPNorm, self.MLPStand) for potential reuse and further evaluation.

In the second mode, when pre-trained models are not available, the code initializes new instances of MLPClassifier. These classifiers have a random initialization state for reproducibility. The code then proceeds to train and evaluate these classifiers using different preprocessing techniques, such as raw data, normalization, and standardization. Similar to previous sections, the evaluation

includes critical metrics like accuracy, recall, precision, and F1-score.

Following the evaluation, the trained MLP models are saved using 'joblib.dump'. This enables the models to be stored and reused later without the need to retrain, offering efficiency and convenience.

In summary, this code streamlines the process of developing, training, and evaluating MLP classifiers for predicting cervical cancer. It emphasizes reusability and performance evaluation through various preprocessing approaches, and ensures the trained models are persisted for future use.

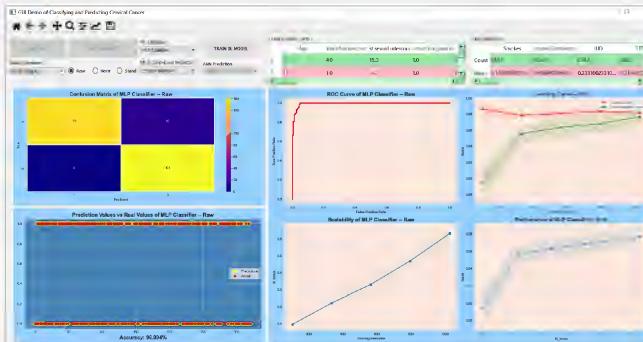


Figure 165 The result using MLP model with raw feature scaling

Step 2 Add this code to the end of **choose_ML_model()** method:

```
1 if strCB == 'MLP Classifier':  
2     self.build_train_mlp()
```

Step 3 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN ML MODEL** buttons. Click on **Raw** radio button. Then, choose **MLP Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 165.

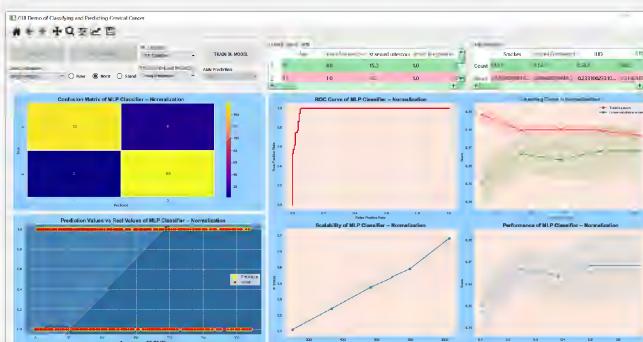


Figure 166 The result using MLP model with normalization feature scaling

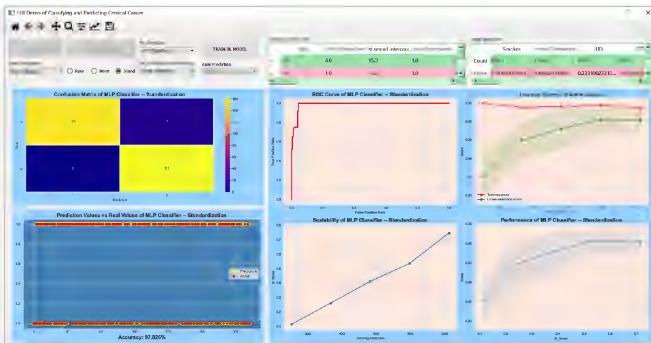


Figure 167 The result using MLP model with standardization feature scaling

Click on **Norm** radio button. Then, choose **MLP Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 166.

Click on **Stand** radio button. Then, choose **MLP Classifier** item from **cbClassifier** widget. Then, you will see the result as shown in Figure 167.

ANN Classifier

Step 1 Define **train_test_ANN()** to split dataset into train and test data for ANN:

```

1  def train_test_DL(self):
2      X = self.df.drop('Biopsy', axis =1).apply(
3          pd.to_numeric, errors='coerce').astype('float64')
4      Y = self.df["Biopsy"]
5      sm = SMOTE(random_state=42)
6      X,Y = sm.fit_resample(X, Y.ravel())
7
8      #Splits dataframe into X_train, X_test, y_train and
9      y_test
10     X_train, X_test, y_train_DL, y_test_DL = \
11         train_test_split(X, Y, test_size = 0.2, \
12         random_state = 2021)
13
14     #Splits dataframe into X_train, X_test, y_train and
15     y_test
16     X_train, X_test, self.y_train_DL, self.y_test_DL =
17     \
18         train_test_split(X, Y, test_size = 0.2, \
19         random_state = 2021)
20
21     #Deletes any outliers in the data using
22     StandardScaler from SKLearn
23     sc = StandardScaler()
24     self.X_train_DL = sc.fit_transform(X_train)
```

```
25 self.X_test_DL = sc.transform(X_test)
26
27 #Saves data
    np.save('X_train_DL.npy', X_train_DL)
    np.save('X_test_DL.npy', X_test_DL)
    np.save('y_train_DL.npy', y_train_DL)
    np.save('y_test_DL.npy', y_test_DL)
```

The code segment is dedicated to the preparation and preprocessing of data for training and testing Deep Learning models for predicting cervical cancer. It carries out the following steps:

1. Data Preprocessing: The code starts by preparing the input features and target variable. It converts the non-numeric columns in the DataFrame (excluding the 'Biopsy' column) to numeric values using `pd.to_numeric` with 'coerce' option to handle any parsing errors. Then, the 'Biopsy' column is separated as the target variable (Y), while the rest of the columns are used as input features (X).
2. Resampling for Class Imbalance: The code utilizes the Synthetic Minority Over-sampling Technique (SMOTE) to address class imbalance by oversampling the minority class. This helps ensure a balanced representation of classes in the dataset. The SMOTE function is applied to both the input features X and the target variable Y.
3. Data Splitting: The preprocessed data is then split into training and testing sets using the `train_test_split` function from `sklearn`. This generates `X_train`, `X_test`, `y_train_DL`, and `y_test_DL` arrays.
4. Data Standardization: Standardization is performed using `StandardScaler` to scale the input features in order to have zero mean and unit variance. It is applied separately to the training and testing sets, resulting in `X_train_DL` and `X_test_DL`.
5. Data Saving: The preprocessed data arrays are saved using the `np.save` function to separate binary files with '.npy' extensions. This enables easy storage and reuse of the preprocessed data for Deep Learning model training.

Overall, this code segment ensures the data is appropriately preprocessed, balanced, split, and standardized to be ready for training and testing Deep Learning models for the prediction of cervical cancer.

Step Define **build ANN()** method to build CNN 1D:

2

```
1 def build_ANN(self,X_train, y_train, NBATCH,
2 NEPOCH):
3     #Imports Tensorflow and create a Sequential Model
4     to add layer for the ANN
5     ann = tf.keras.models.Sequential()
6
7     #Input layer
8     ann.add(tf.keras.layers.Dense(units=500,
9         input_dim=33,
10        kernel_initializer='uniform',
11        activation='relu'))
12    ann.add(tf.keras.layers.Dropout(0.5))
13
14    #Hidden layer 1
15    ann.add(tf.keras.layers.Dense(units=200,
16        kernel_initializer='uniform',
17        activation='relu'))
18    ann.add(tf.keras.layers.Dropout(0.5))
19
20    #Output layer
21    ann.add(tf.keras.layers.Dense(units=1,
22        kernel_initializer='uniform',
23        activation='sigmoid'))
24
25    print(ann.summary()) #for showing the structure
26    and parameters
27
28    #Compiles the ANN using ADAM optimizer.
29    ann.compile(optimizer = 'adam', \
30        loss = 'binary_crossentropy', metrics =
31    ['accuracy'])
32
33    #Trains the ANN with 100 epochs.
34    history = ann.fit(X_train, y_train, batch_size = 64,
35    \
36        validation_split=0.20, epochs = 250,
37    shuffle=True)
38
#Saves model
ann.save('cervical_model.h5')

#Saves history into npy file
np.save('cervical_history.npy', history.history)
```

The code defines a function **build_ANN()** which is responsible for constructing, training, and saving an Artificial Neural Network (ANN) model for predicting cervical cancer using TensorFlow. Here's a breakdown of the steps performed in the code:

1. Import Libraries and Create Model: The code imports TensorFlow as `tf` and creates a sequential model named `ann` using

- `tf.keras.models.Sequential()`. The sequential model allows for stacking layers sequentially.
- 2. Add Layers to the Model: The code adds layers to the model using the `add` method. The layers include:
 - 3. Input Layer: Consisting of 500 units, using the '`relu`' activation function and initialized with a uniform distribution. A dropout layer with a dropout rate of 0.5 is added after this layer.
 - 4. Hidden Layer 1: Consisting of 200 units, '`relu`' activation, and a uniform initializer. Another dropout layer with a dropout rate of 0.5 follows this layer.
 - 5. Output Layer: A single unit with the '`sigmoid`' activation function for binary classification, initialized uniformly.
 - 6. Model Summary: The `ann.summary()` method is used to print a summary of the model's architecture, including the number of parameters and layer shapes.
 - 7. Compile Model: The model is compiled using the '`adam`' optimizer and '`binary_crossentropy`' as the loss function. Additionally, '`accuracy`' is chosen as the evaluation metric.
 - 8. Train Model: The model is trained using the `fit` method. It's trained on `X_train` and `y_train` with a batch size of 64 and a validation split of 20%. The training is carried out for 250 epochs, and the `shuffle` parameter is set to True.
 - 9. Save Model and History: After training, the model is saved to a file named '`cervical_model.h5`' using `ann.save()`. Additionally, the training history is saved to a numpy file named '`cervical_history.npy`' using `np.save()`.

In summary, the `build_ANN()` function defines and trains a multi-layer feedforward neural network (ANN) model for cervical cancer prediction using TensorFlow. The model architecture includes an input layer, hidden layers with dropout, and an output layer. The model is compiled, trained, and then saved along with its training history for future analysis and evaluation.

Step Define `train ANN()` method to execute the training:

3

```
1 def train ANN(self):  
2     if path.isfile('X_train_DL.npy'):  
3         #Loads files  
4         self.X_train_DL = \
```

```

5      np.load('X_train_DL.npy',allow_pickle=True)
6  self.X_test_DL = \
7      np.load('X_test_DL.npy',allow_pickle=True)
8  self.y_train_DL = \
9      np.load('y_train_DL.npy',allow_pickle=True)
10 self.y_test_DL = \
11     np.load('y_test_DL.npy',allow_pickle=True)
12
13 else:
14     self.train_test_ANN()
15 #Loads files
16 self.X_train_DL = \
17     np.load('X_train_DL.npy',allow_pickle=True)
18 self.X_test_DL = \
19     np.load('X_test_DL.npy',allow_pickle=True)
20 self.y_train_DL = \
21     np.load('y_train_DL.npy',allow_pickle=True)
22 self.y_test_DL = \
23     np.load('y_test_DL.npy',allow_pickle=True)
24
25 if path.isfile('cervical_model.h5') == False:
26     self.build_ANN(self.X_train_DL, self.y_train_DL,
27 32, 250)
28
29 #Turns on cbPredictionDL
30 self.cbPredictionDL.setEnabled(True)
31
32 #Turns off pbTrainDL
33 self.pbTrainDL.setEnabled(False)

```

The code defines a function `train_ANN()` which is responsible for training an Artificial Neural Network (ANN) model for predicting cervical cancer using saved data files. Here's a breakdown of the steps performed in the code:

1. Check Data File Existence: The code first checks if the data files '`X_train_DL.npy`', '`X_test_DL.npy`', '`y_train_DL.npy`', and '`y_test_DL.npy`' exist using `path.isfile()`. If they exist, the data is loaded into the corresponding variables.
2. Load Data or Train: If the data files do not exist, the function calls the `train_test_ANN()` function (which is assumed to be defined elsewhere) to generate and save the required data files. Afterward, it loads the newly created data files into the corresponding variables.
3. Check Model Existence: The code then checks if the ANN model file '`cervical_model.h5`' exists using `path.isfile()`. If the model file does not exist, it calls the `build_ANN` function (assumed

to be defined elsewhere) to construct and train the ANN model using the loaded data.

4. Enable/Disable Widgets: The function enables the widget cbPredictionDL (presumably a checkbox) and disables the widget pbTrainDL (presumably a button) to control the user interface.

In summary, the train_ANN() function loads or generates the required data files, checks for the existence of the ANN model file, constructs and trains the model if necessary, and manages the state of certain user interface widgets. This function essentially prepares the data, builds, trains, and initializes the ANN model for cervical cancer prediction.

- Step 4 Define **plot_loss_acc()** method to plot loss and accuracy of the model:

```
1 def plot_loss_acc(self,train_loss, val_loss, train_acc,
2 val_acc, widget,strPlot):
3     widget.canvas.figure.clf()
4     widget.canvas.axis1 = \
5
6     widget.canvas.figure.add_subplot(211,facecolor="#fbe7dd")
7     widget.canvas.axis1.plot(train_loss, \
8         label='Training Loss',color='blue', linewidth=3.0)
9     widget.canvas.axis1.plot(val_loss, 'b--', \
10        label='Validation Loss',color='red', linewidth=3.0)
11    widget.canvas.axis1.set_title('Loss',\
12        fontweight ="bold",fontsize=20)
13    widget.canvas.axis1.set_xlabel('Epoch')
14    widget.canvas.axis1.grid(True, alpha=0.75, lw=1, ls='-.')
15    widget.canvas.axis1.legend()
16
17    widget.canvas.axis1 = \
18
19    widget.canvas.figure.add_subplot(212,facecolor="#fbe7dd")
20    widget.canvas.axis1.plot(train_acc, \
21        label='Training Accuracy',color='blue', linewidth=3.0)
22    widget.canvas.axis1.plot(val_acc, 'b--', \
23        label='Validation Accuracy',color='red',
24        linewidth=3.0)
25    widget.canvas.axis1.set_title('Accuracy',\
26        fontweight ="bold",fontsize=20)
27    widget.canvas.axis1.set_xlabel('Epoch')
    widget.canvas.axis1.grid(True, alpha=0.75, lw=1, ls='-.')
    widget.canvas.axis1.legend()
    widget.canvas.draw()
```

The plot_loss_acc() function is designed to generate a graphical representation of the training and validation process for a machine learning model. It takes input parameters such as training and

validation loss values, as well as training and validation accuracy values. With these inputs, the function creates a plot using a widget's canvas.

The canvas is divided into two subplots: one for displaying the loss trends and another for showing accuracy trends. The first subplot visualizes the training and validation loss over epochs, with training loss shown in blue and validation loss displayed as dashed red lines. The second subplot illustrates the training and validation accuracy, using the same color scheme. Both subplots have titles, x-axis and y-axis labels, grid lines, and legends to differentiate between the training and validation metrics. Once the plot is set up, the function updates the canvas to render the graphical representation, providing a clear view of the model's performance during training and validation.

Step 5 Define **pred_ann()** to calculate the predicted values:

```
1 def pred_ann(self, xtest, ytest):
2     self.train_ANN()
3     self.ann = load_model('cervical_model.h5')
4
5     prediction = self.ann.predict(xtest)
6     label = [int(p>=0.5) for p in prediction]
7
8     print("test_target:", ytest)
9     print("pred_val:", label)
10
11    #Performance Evaluation - Accuracy,
12    #Classification Report & Confusion Matrix
13    #Accuracy Score
14    print ('Accuracy Score : ', \
15           accuracy_score(label, ytest), '\n')
16
17    #precision, recall report
18    print ('Classification Report :\n\n' ,\
19           classification_report(label, ytest))
20
21    return label
```

The **pred_ann()** function performs prediction using a pre-trained artificial neural network (ANN) model. It first triggers the **train_ANN()** function to ensure the model is available for prediction. Once the ANN model is loaded from the saved file '**cervical_model.h5**', the function makes predictions on the provided test data **xtest**. The predictions are then converted to binary labels using a threshold of 0.5.

The function prints out the actual test target values and the predicted labels to compare the results. It proceeds to evaluate the model's performance by calculating the accuracy score, classification report, and confusion matrix. The accuracy score measures the correctness of the predictions, and the classification

report provides a summary of precision, recall, and other metrics for each class. This information helps assess the model's predictive capability and overall performance on the test data.

Finally, the function returns the predicted labels for further analysis or use. It serves as a comprehensive tool for assessing the ANN's predictions, aiding in understanding the model's behavior and its accuracy in predicting the target variable.

- Step 6 Define **choose_prediction_ANN()** to read selected item from **cbPredictionDL** widget:

```
1 def choose_prediction_ANN(self):
2     strCB = self.cbPredictionDL.currentText()
3
4     if strCB == 'CNN 1D':
5         pred_val = self.pred_ann(self.X_test_DL,
6             self.y_test_DL)
7
8         #Plots true values versus predicted values
9         self.widgetPlot2.canvas.figure.clf()
10        self.widgetPlot2.canvas.axis1 =
11            self.widgetPlot2.canvas.figure.add_subplot(111,
12                facecolor = '#fbe7dd')
13        self.plot_real_pred_val(pred_val, self.y_test_DL,
14            self.widgetPlot2, 'CNN 1D')
15        self.widgetPlot2.canvas.figure.tight_layout()
16        self.widgetPlot2.canvas.draw()
17
18        #Plot confusion matrix
19        self.widgetPlot1.canvas.figure.clf()
20        self.widgetPlot1.canvas.axis1 =
21            self.widgetPlot1.canvas.figure.add_subplot(111,
22                facecolor = '#fbe7dd')
23        self.plot_cm(pred_val, self.y_test_DL,
24            self.widgetPlot1, 'CNN 1D')
25        self.widgetPlot1.canvas.figure.tight_layout()
26        self.widgetPlot1.canvas.draw()
27
28        #Loads history
29        history = np.load('cervical_history.npy',
30            allow_pickle=True).item()
31        train_loss = history['loss']
32        train_acc = history['accuracy']
33        val_acc = history['val_accuracy']
34        val_loss = history['val_loss']
35        self.plot_loss_acc(train_loss, val_loss, train_acc,
            val_acc, self.widgetPlot3, "History of " +
            strCB)
```

The **choose_prediction_ANN()** function is responsible for selecting and evaluating different prediction methods for an artificial neural network (ANN) model based on the chosen option

from the combo box (cbPredictionDL). If the selected option is 'CNN 1D', the function performs the following steps:

1. It calls the pred_ann() function to obtain predicted values (pred_val) using the pre-trained ANN model on the test data (X_test_DL).
2. The true values (y_test_DL) and the predicted values are then used to create two visualizations:
 - a. A plot showing the comparison between true values and predicted values using the plot_real_pred_val function. The result is displayed in widgetPlot2.
 - b. A confusion matrix is generated using the plot_cm function and displayed in widgetPlot1.
3. The function also loads the training history of the ANN from the file 'cervical_history.npy'. It extracts and utilizes data such as training loss, training accuracy, validation accuracy, and validation loss to create a plot that illustrates the history of the training process. This plot is generated using the plot_loss_acc function and displayed in widgetPlot3.

Overall, the choose_prediction_ANN() function provides a comprehensive analysis and visualization of the ANN's prediction performance, including comparisons between predicted and actual values, confusion matrix, and the history of the training process. This helps users gain insights into the model's behavior and its effectiveness in making predictions.

Step 7 Connect **clicked()** event of **pbTrainDL** widget with **train_ANN()** method as shown in line 14 and **currentIndexChanged()** event of **cbPredictionDL** widget with **choose_prediction_ANN()** method as shown in line 15-16:

```
1 def __init__(self):  
2     QMainWindow.__init__(self)  
3     loadUi("gui_cervical.ui",self)  
4     self.setWindowTitle(  
5     "GUI Demo of Classifying and Predicting Cervical  
6     Cancer")  
7     self.addToolBar(NavigationToolbar(\br/>8     self.widgetPlot1.canvas, self))  
9     self.pbLoad.clicked.connect(self.import_dataset)  
10    self.initial_state(False)  
11    self.pbTrainML.clicked.connect(self.train_model_ML)  
12    self.cbData.currentIndexChanged.connect(self.choose_plot  
13    )  
14    self.cbClassifier.currentIndexChanged.connect(  
15    self.choose_DL_model)  
16    self.pbTrainDL.clicked.connect(self.train_ANN)
```

```
self.cbPredictionDL.currentIndexChanged.connect(\n    self.choose_prediction_ANN)
```

- Step 8 Run **gui_cervical.py** and click **LOAD DATA** and **TRAIN DL MODEL** buttons. Then, choose **CNN 1D** item from **cbPredictionDL** widget. Then, you will see the result as shown in Figure 168.

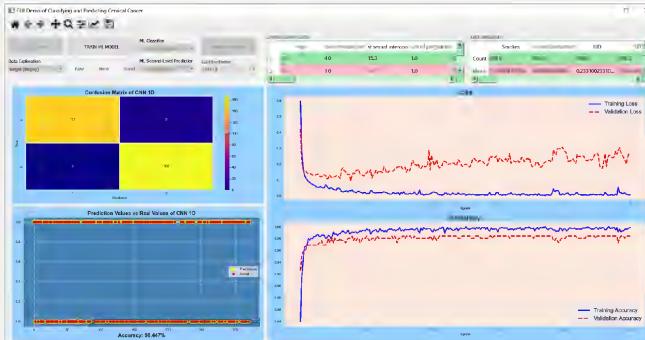


Figure 168 The result using sequential CNN

Following is the full version of **gui_cervical.py**:

```
#gui_cervical.py\nfrom PyQt5.QtWidgets import *\nfrom PyQt5.uic import loadUi\nfrom matplotlib.backends.backend_qt5agg import\n(NavigationToolbar2QT as NavigationToolbar)\nfrom matplotlib.colors import ListedColormap\n\nimport numpy as np\nimport pandas as pd\nimport matplotlib.pyplot as plt\nimport seaborn as sns\nsns.set_style('darkgrid')\nimport warnings\nimport mglearn\nwarnings.filterwarnings('ignore')\nimport os\nimport joblib\nfrom numpy import save\nfrom numpy import load\nfrom os import path\nfrom sklearn.metrics import roc_auc_score,roc_curve\nfrom sklearn.model_selection import train_test_split,\nRandomizedSearchCV, GridSearchCV, StratifiedKFold\nfrom sklearn.preprocessing import StandardScaler,\nMinMaxScaler\nfrom sklearn.linear_model import LogisticRegression\nfrom sklearn.naive_bayes import GaussianNB\nfrom sklearn.tree import DecisionTreeClassifier\nfrom sklearn.svm import SVC
```

```
from sklearn.ensemble import RandomForestClassifier,  
ExtraTreesClassifier  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.ensemble import AdaBoostClassifier,  
GradientBoostingClassifier  
from xgboost import XGBClassifier
```

```

from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler, LabelEncoder,
OneHotEncoder
from sklearn.metrics import confusion_matrix, accuracy_score,
recall_score, precision_score
from sklearn.metrics import classification_report, f1_score,
plot_confusion_matrix
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import learning_curve
from mlxtend.plotting import plot_decision_regions
import tensorflow as tf
from sklearn.base import clone
from sklearn.decomposition import PCA
from tensorflow.keras.models import Sequential, Model, load_model

class DemoGUI_Cervical(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        loadUi("gui_cervical.ui", self)
        self.setWindowTitle(
            "GUI Demo of Classifying and Predicting Cervical Cancer")
        self.addToolBar(NavigationToolbar(
            self.widgetPlot1.canvas, self))
        self.pbLoad.clicked.connect(self.import_dataset)
        self.initial_state(False)
        self.pbTrainML.clicked.connect(self.train_model_ML)
        self.cbData.currentIndexChanged.connect(self.choose_plot)
        self.cbClassifier.currentIndexChanged.connect(self.choose_ML_model
        )
        self.pbTrainDL.clicked.connect(self.train_ANN)
        self.cbPredictionDL.currentIndexChanged.connect(
            self.choose_prediction_ANN)

```

```
# Takes a df and writes it to a qtable provided. df headers become
qtable headers
@staticmethod
def write_df_to_qtable(df,table):
    headers = list(df)
    table.setRowCount(df.shape[0])
    table.setColumnCount(df.shape[1])
    table.setHorizontalHeaderLabels(headers)

# getting data from df is computationally costly so convert it to array
first
    df_array = df.values
    for row in range(df.shape[0]):
        for col in range(df.shape[1]):
            table.setItem(row, col,\n                QTableWidgetItem(str(df_array[row,col])))

def populate_table(self,data, table):
    #Populates two tables
    self.write_df_to_qtable(data,table)

    table.setAlternatingRowColors(True)
    table.setStyleSheet(\n        "alternate-background-color: #ffbacd;background-color: #9be5aa;");

def initial_state(self, state):
    self.pbTrainML.setEnabled(state)
    self.cbData.setEnabled(state)
    self.cbClassifier.setEnabled(state)
    self.cbPredictionML.setEnabled(state)
    self.cbPredictionDL.setEnabled(state)
    self.pbTrainDL.setEnabled(state)
    self.rbRaw.setEnabled(state)
    self.rbNorm.setEnabled(state)
    self.rbStand.setEnabled(state)
```

```

def read_dataset(self, dir):
    #Loads csv file
    df = pd.read_csv(dir)

    #Checks null values
    df = df.replace('?', np.NaN)
    print(df.isnull().sum())
    print('Total number of null values: ', df.isnull().sum().sum())

    #Drops two columns of STDs
    df.drop(['STDs: Time since first diagnosis', \
              'STDs: Time since last diagnosis'], inplace=True, axis=1)

    numerical_df = ['Age', 'Number of sexual partners', \
                    'First sexual intercourse', 'Num of pregnancies', \
                    'Smokes (years)', 'Smokes (packs/year)', \
                    'Hormonal Contraceptives (years)', 'IUD (years)', 'STDs (number)']

    categorical_df = ['Smokes', 'Hormonal Contraceptives', \
                      'IUD', 'STDs', 'STDs:condylomatosis', \
                      'STDs:cervical condylomatosis', \
                      'STDs:vaginal condylomatosis', \
                      'STDs:vulvo-perineal condylomatosis', \
                      'STDs:syphilis', 'STDs:pelvic inflammatory disease', \
                      'STDs:genital herpes', 'STDs:molluscum contagiosum', \
                      'STDs:AIDS', 'STDs:HIV', 'STDs:Hepatitis B', 'STDs:HPV', \
                      'STDs: Number of diagnosis', 'Dx:Cancer', 'Dx:CIN', \
                      'Dx:HPV', 'Dx', 'Hinselmann', 'Schiller', 'Citology', 'Biopsy']

    #Fills the missing values of numeric data columns with mean of the
    #column data.
    for feature in numerical_df:
        print(feature, ",df[feature].apply(pd.to_numeric, \
                                             errors='coerce').mean())
        feature_mean = round(df[feature].apply(pd.to_numeric, \
                                             errors='coerce').mean())
        df[feature] = df[feature].fillna(feature_mean)

```

```
        errors='coerce').mean(),1)
df[feature] = df[feature].fillna(feature_mean)

for feature in categorical_df:
    df[feature] = df[feature].apply(pd.to_numeric, \
        errors='coerce').fillna(1.0)

return df

def import_dataset(self):
    curr_path = os.getcwd()
    dataset_dir = curr_path + "/kag_risk_factors_cervical_cancer.csv"
self.EPOCHS = 250
self.BATCH_SIZE = 32

#Loads csv file
self.df = self.read_dataset(dataset_dir)

#Populates tables with data
self.populate_table(self.df, self.twData1)
self.label1.setText('Cervical Cancer Data')

self.populate_table(self.df.iloc[:,1:].describe(), self.twData2)
self.twData2.setVerticalHeaderLabels(['Count', 'Mean', \
    'Std', 'Min', '25%', '50%', '75%', 'Max'])
self.label2.setText('Data Description')

#Turns on pbTrainML widget
self.pbTrainML.setEnabled(True)
self.pbTrainDL.setEnabled(True)

#Turns off pbLoad
self.pbLoad.setEnabled(False)

#Populates cbData
self.populate_cbData()
```

```
def populate_cbData(self):  
    self.cbData.addItem(["target (Biopsy)"])  
    self.cbData.addItem(self.df.iloc[:,0:-1])  
    self.cbData.addItem(["Features Importance"])  
    self.cbData.addItem(["Correlation Matrix", \  
    "Pairwise Relationship", "Features Correlation"])  
  
def fit_dataset(self, df):  
    X = df.drop('Biopsy', axis=1).apply(pd.to_numeric, \  
        errors='coerce').astype('float64')  
    y = df['Biopsy']  
    sm = SMOTE(random_state=42)  
    X,y = sm.fit_resample(X, y.ravel())  
  
return X, y  
  
def train_test(self):  
    X, y = self.fit_dataset(self.df)  
  
    #Splits the data into training and testing  
    X_train, X_test, y_train, y_test = train_test_split(X, y, \  
        test_size=0.2, random_state=2021, stratify=y)  
    self.X_train_raw = X_train.copy()  
    self.X_test_raw = X_test.copy()  
    self.y_train_raw = y_train.copy()  
    self.y_test_raw = y_test.copy()  
  
    #Saves into npy files  
    save('X_train_raw.npy', self.X_train_raw)  
    save('y_train_raw.npy', self.y_train_raw)  
    save('X_test_raw.npy', self.X_test_raw)  
    save('y_test_raw.npy', self.y_test_raw)  
  
    self.X_train_norm = X_train.copy()  
    self.X_test_norm = X_test.copy()
```

```

self.y_train_norm = y_train.copy()
self.y_test_norm = y_test.copy()
    norm = MinMaxScaler()
self.X_train_norm = norm.fit_transform(self.X_train_norm)
self.X_test_norm = norm.transform(self.X_test_norm)

#Saves into npy files
    save('X_train_norm.npy', self.X_train_norm)
    save('y_train_norm.npy', self.y_train_norm)
    save('X_test_norm.npy', self.X_test_norm)
    save('y_test_norm.npy', self.y_test_norm)

self.X_train_stand = X_train.copy()
self.X_test_stand = X_test.copy()
self.y_train_stand = y_train.copy()
self.y_test_stand = y_test.copy()
    scaler = StandardScaler()
self.X_train_stand = scaler.fit_transform(self.X_train_stand)
self.X_test_stand = scaler.transform(self.X_test_stand)

#Saves into npy files
    save('X_train_stand.npy', self.X_train_stand)
    save('y_train_stand.npy', self.y_train_stand)
    save('X_test_stand.npy', self.X_test_stand)
    save('y_test_stand.npy', self.y_test_stand)

def split_data_ML(self):
if path.isfile('X_train_raw.npy'):
#Loads npy files
self.X_train_raw = np.load('X_train_raw.npy',allow_pickle=True)
self.y_train_raw = np.load('y_train_raw.npy',allow_pickle=True)
self.X_test_raw = np.load('X_test_raw.npy',allow_pickle=True)
self.y_test_raw = np.load('y_test_raw.npy',allow_pickle=True)

self.X_train_norm = np.load('X_train_norm.npy',allow_pickle=True)
self.y_train_norm = np.load('y_train_norm.npy',allow_pickle=True)

```

```
self.X_test_norm = np.load('X_test_norm.npy',allow_pickle=True)
self.y_test_norm = np.load('y_test_norm.npy',allow_pickle=True)

self.X_train_stand = \
    np.load('X_train_stand.npy',allow_pickle=True)
self.y_train_stand = \
    np.load('y_train_stand.npy',allow_pickle=True)
self.X_test_stand = np.load('X_test_stand.npy',allow_pickle=True)
self.y_test_stand = np.load('y_test_stand.npy',allow_pickle=True)

else:
    self.train_test()

#Prints each shape
print('X train raw shape: ', self.X_train_raw.shape)
print('Y train raw shape: ', self.y_train_raw.shape)
print('X test raw shape: ', self.X_test_raw.shape)
print('Y test raw shape: ', self.y_test_raw.shape)

#Prints each shape
print('X train norm shape: ', self.X_train_norm.shape)
print('Y train norm shape: ', self.y_train_norm.shape)
print('X test norm shape: ', self.X_test_norm.shape)
print('Y test norm shape: ', self.y_test_norm.shape)

#Prints each shape
print('X train stand shape: ', self.X_train_stand.shape)
print('Y train stand shape: ', self.y_train_stand.shape)
print('X test stand shape: ', self.X_test_stand.shape)
print('Y test stand shape: ', self.y_test_stand.shape)

def train_model_ML(self):
    self.split_data_ML()

#Turns on three widgets
self.cbData.setEnabled(True)
```

```

self.cbClassifier.setEnabled(True)
self.cbPredictionML.setEnabled(True)

#Turns off pbTrainML
self.pbTrainML.setEnabled(False)

#Turns on three radio buttons
self.rbRaw.setEnabled(True)
self.rbNorm.setEnabled(True)
self.rbStand.setEnabled(True)
self.rbRaw.setChecked(True)

def dist_target(self, df_target, var_target, labels, widget):
    df_target.value_counts().plot.pie(
        ax = widget.canvas.axis1, labels=labels,\n
        startangle=40, explode=[0,0.15], shadow=True,\n
        colors=['#ff6666','#F5C7B8FF'], autopct = '%1.1f%%',\n
        textprops={'fontsize': 14})
    widget.canvas.axis1.set_title(\n
'The distribution of target variable ('+ var_target+')', \n
        fontweight ="bold", fontsize=14)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

def dist_target_versus_features(self,df_target,var_target, labels):
#Plots distribution of target variable in pie chart
self.widgetPlot1.canvas.figure.clf()
self.widgetPlot1.canvas.axis1 =
self.widgetPlot1.canvas.figure.add_subplot(111,
    facecolor = '#fbe7dd')
self.dist_target(df_target,var_target, labels, self.widgetPlot1)

#Plots distribution: Hormonal Contraceptives (years) versus var_target
self.widgetPlot2.canvas.figure.clf()
self.widgetPlot2.canvas.axis1 =
self.widgetPlot2.canvas.figure.add_subplot(211,

```

```

        facecolor = '#fbe7dd')
self.widgetPlot2.canvas.axis1.set_title(\n'Distribution: Hormonal Contraceptives (years) versus ' + \
    var_target, fontweight ="bold", fontsize=16)
sns.histplot(data=self.df, \
    x=self.df["Hormonal Contraceptives (years)"].apply(\n
        pd.to_numeric, errors='coerce').astype('float64'),\
    ax= self.widgetPlot2.canvas.axis1,zorder=2,\n
    kde=False,hue=df_target,multiple="stack", \
    shrink=.8,linewidth=0.3,alpha=1)
self.widgetPlot2.canvas.axis1.set_xlabel(\n
    "Hormonal Contraceptives (years)",\n
    fontweight ="bold", fontsize=16)

self.widgetPlot2.canvas.axis1 = \
self.widgetPlot2.canvas.figure.add_subplot(212,\n
    facecolor = '#fbe7dd')
self.widgetPlot2.canvas.axis1.set_title(\n'Distribution: IUD (years) versus ' + var_target, \
    fontweight ="bold", fontsize=16)
sns.histplot(data=self.df, x=self.df["IUD (years)"].apply(\n
    pd.to_numeric, errors='coerce').astype('float64'),\
    ax= self.widgetPlot2.canvas.axis1,zorder=2,\n
    kde=False,hue=df_target,multiple="stack", \
    shrink=.8,linewidth=0.3,alpha=1)
self.widgetPlot2.canvas.axis1.set_xlabel("IUD (years)",\n
    fontweight ="bold", fontsize=16)
self.widgetPlot2.canvas.figure.tight_layout()
self.widgetPlot2.canvas.draw()

#Plots distribution: Age versus var_target
self.widgetPlot3.canvas.figure.clf()
self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(221,\n
    facecolor = '#fbe7dd')
self.widgetPlot3.canvas.axis1.set_title(\n

```

```

'Distribution: Age versus ' + var_target, \
    fontweight ="bold",fontsize=16)
sns.histplot(data=self.df, x=self.df.Age,\n
    ax= self.widgetPlot3.canvas.axis1,zorder=2,\n
    kde=False,hue=df_target,multiple="stack", \
    shrink=.8,linewidth=0.3,alpha=1)
self.widgetPlot3.canvas.axis1.set_xlabel("Age",\n
    fontweight ="bold",fontsize=16)

#Plots distribution: First sexual intercourse versus var_target
self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(222,\n
    facecolor = '#fbe7dd')
self.widgetPlot3.canvas.axis1.set_title(\n
'Distribution: First sexual intercourse versus ' + \
    var_target, fontweight ="bold",fontsize=16)
sns.histplot(data=self.df, \
    x=self.df["First sexual intercourse"].apply(\n
        pd.to_numeric, errors='coerce').astype('float64'),\n
    ax= self.widgetPlot3.canvas.axis1,zorder=2,\n
    kde=False,hue=df_target,multiple="stack", \
    shrink=.8,linewidth=0.3,alpha=1)
self.widgetPlot3.canvas.axis1.set_xlabel(\n
    "First sexual intercourse",fontweight ="bold",fontsize=16)

#Plots distribution: Num of pregnancies versus var_target
self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(223,\n
    facecolor = '#fbe7dd')
self.widgetPlot3.canvas.axis1.set_title(\n
'Distribution: Num of pregnancies versus ' + var_target, \
    fontweight ="bold",fontsize=16)
sns.histplot(data=self.df, \
    x=self.df["Num of pregnancies"].apply(pd.to_numeric, \
        errors='coerce').astype('float64'),\n
    ax= self.widgetPlot3.canvas.axis1,zorder=2,\n

```

```

    kde=False,hue=df_target,multiple="stack", \
    shrink=.8,linewidth=0.3,alpha=1)
self.widgetPlot3.canvas.axis1.set_xlabel("Num of pregnancies",\
fontweight ="bold",fontsize=16)

#Plots distribution: Hormonal Contraceptives (years) versus var_target
self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(224, \
facecolor = '#fbe7dd')
self.widgetPlot3.canvas.axis1.set_title(\n'Distribution: Hormonal Contraceptives (years) versus ' + \
var_target, fontweight ="bold",fontsize=16)
sns.histplot(data=self.df, \
x=self.df["Hormonal Contraceptives (years)"].apply(\npd.to_numeric, errors='coerce').astype('float64'),\
ax= self.widgetPlot3.canvas.axis1,zorder=2, \
kde=False,hue=df_target,multiple="stack", \
shrink=.8,linewidth=0.3,alpha=1)
self.widgetPlot3.canvas.axis1.set_xlabel(\n'Hormonal Contraceptives (years)",fontweight ="bold",\
fontsize=16)

self.widgetPlot3.canvas.figure.tight_layout()
self.widgetPlot3.canvas.draw()

def hist_feat_versus_other(self,df,feat,another,legend,ax0):
    background_color = "#fbe7dd"
    sns.set_palette(['#ff355d','#ffd514'])
for s in ["right", "top"]:
    ax0.spines[s].set_visible(False)

    ax0.set_facecolor(background_color)
    ax0_sns = sns.histplot(data=df, \
x=feat,ax=ax0,zorder=2,kde=False,hue=another,\n
multiple="stack", shrink=.8,linewidth=0.3,alpha=1)

```

```

ax0_sns.set_xlabel("", fontsize=10, weight='bold')
ax0_sns.set_ylabel("", fontsize=10, weight='bold')

ax0_sns.grid(which='major', axis='x', zorder=0, \
    color="#EEEEEE", linewidth=0.4)
ax0_sns.grid(which='major', axis='y', zorder=0, \
    color="#EEEEEE", linewidth=0.4)

ax0_sns.tick_params(labelsize=8, width=0.5, length=1.5)
ax0_sns.legend(legend, ncol=2, facecolor="#D8D8D8", \
    edgecolor=background_color, fontsize=12, \
    bbox_to_anchor=(1, 0.989), loc='upper right')
ax0.set_facecolor(background_color)

def prob_feat_versus_other(self, df, feat, another, legend, ax0):
    background_color = "#fbe7dd"
    sns.set_palette(['#ff355d', '#ffd514'])
    for s in ["right", "top"]:
        ax0.spines[s].set_visible(False)

    ax0.set_facecolor(background_color)

    ax0_sns = sns.kdeplot(x=feat, ax=ax0, hue=another, \
        linewidth=0.3, fill=True, cbar='g', zorder=2, alpha=1, \
        multiple='stack')

    ax0_sns.set_xlabel("", fontsize=10, weight='bold')
    ax0_sns.set_ylabel("", fontsize=10, weight='bold')

    ax0_sns.grid(which='major', axis='x', zorder=0, \
        color="#EEEEEE", linewidth=0.4)
    ax0_sns.grid(which='major', axis='y', zorder=0, \
        color="#EEEEEE", linewidth=0.4)

    ax0_sns.tick_params(labelsize=8, width=0.5, length=1.5)
    ax0_sns.legend(legend, ncol=2, facecolor="#D8D8D8", \

```

```
    edgecolor=background_color, fontsize=12, \
    bbox_to_anchor=(1, 0.989), loc='upper right')
ax0.set_facecolor(background_color)

def dist_feat_versus_others(self,df_target,title):
    self.widgetPlot3.canvas.figure.clf()
    print(self.df.Smokes.value_counts())
    self.widgetPlot3.canvas.axis1 = \
        self.widgetPlot3.canvas.figure.add_subplot(231, \
            facecolor = '#fbe7dd')
    self.widgetPlot3.canvas.axis1.set_title('Smokes versus ' + \
        title, fontweight ="bold",fontsize=16)
    self.hist_feat_versus_other(self.df, \
        df_target,self.df.Smokes,['Smokes','No Smokes'],\
        self.widgetPlot3.canvas.axis1)
    self.widgetPlot3.canvas.axis1.set_xlabel(title, \
        fontweight ="bold",fontsize=16)

    print(self.df["Hormonal Contraceptives"].value_counts())
    self.widgetPlot3.canvas.axis1 = \
        self.widgetPlot3.canvas.figure.add_subplot(232, \
            facecolor = '#fbe7dd')
    self.widgetPlot3.canvas.axis1.set_title(\
        'Hormonal Contraceptives versus ' + title, \
        fontweight ="bold",fontsize=16)
    self.hist_feat_versus_other(self.df,df_target, \
        self.df["Hormonal Contraceptives"],['Hormonal',\
        'No Hormonal'],self.widgetPlot3.canvas.axis1)
    self.widgetPlot3.canvas.axis1.set_xlabel(title, \
        fontweight ="bold",fontsize=16)

    print(self.df.IUD.value_counts())
    self.widgetPlot3.canvas.axis1 = \
        self.widgetPlot3.canvas.figure.add_subplot(233, \
            facecolor = '#fbe7dd')
    self.widgetPlot3.canvas.axis1.set_title('IUD versus ' + title, \
```

```
        fontweight = "bold", fontsize=16)
self.hist_feat_versus_other(self.df, df_target, self.df.IUD,\n    ['IUD', 'No IUD'], self.widgetPlot3.canvas.axis1)
self.widgetPlot3.canvas.axis1.set_xlabel(title,\n    fontweight = "bold", fontsize=16)

print(self.df.STDs.value_counts())
self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(234,\n    facecolor = '#fbe7dd')
self.widgetPlot3.canvas.axis1.set_title('STDs versus ' + title, \
    fontweight = "bold", fontsize=16)
self.hist_feat_versus_other(self.df, df_target, self.df.STDs,\n    ['STDs', 'No STDs'], self.widgetPlot3.canvas.axis1)
self.widgetPlot3.canvas.axis1.set_xlabel(title,\n    fontweight = "bold", fontsize=16)

print(self.df.Dx.value_counts())
self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(235,\n    facecolor = '#fbe7dd')
self.widgetPlot3.canvas.axis1.set_title('Dx versus ' + title, \
    fontweight = "bold", fontsize=16)
self.hist_feat_versus_other(self.df, df_target, self.df.Dx,\n    ['Dx', 'No Dx'], self.widgetPlot3.canvas.axis1)
self.widgetPlot3.canvas.axis1.set_xlabel(title,\n    fontweight = "bold", fontsize=16)

print(self.df.Hinselmann.value_counts())
self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(236,\n    facecolor = '#fbe7dd')
self.widgetPlot3.canvas.axis1.set_title(\n    'Hinselmann versus ' + title, fontweight = "bold", fontsize=16)
self.hist_feat_versus_other(self.df, df_target,\n    self.df.Hinselmann, ['Hinselmann=1', 'Hinselmann=0'],\n    
```

```
self.widgetPlot3.canvas.axis1)
self.widgetPlot3.canvas.axis1.set_xlabel(title,\n    fontweight ="bold",fontsize=16)

self.widgetPlot3.canvas.figure.tight_layout()
self.widgetPlot3.canvas.draw()

self.widgetPlot2.canvas.figure.clf()
print(self.df.Schiller.value_counts())
self.widgetPlot2.canvas.axis1 = \
self.widgetPlot2.canvas.figure.add_subplot(211,\n    facecolor = '#fbe7dd')
self.widgetPlot2.canvas.axis1.set_title('Schiller versus ' + title, \
    fontweight ="bold",fontsize=16)
self.hist_feat_versus_other(self.df,df_target,\nself.df.Schiller,['Schiller=1','Schiller=0'],\
self.widgetPlot2.canvas.axis1)
self.widgetPlot2.canvas.axis1.set_xlabel(title,\n    fontweight ="bold",fontsize=16)

print(self.df.Citology.value_counts())
self.widgetPlot2.canvas.axis1 = \
self.widgetPlot2.canvas.figure.add_subplot(212,\n    facecolor = '#fbe7dd')
self.widgetPlot2.canvas.axis1.set_title('Citology versus ' + \
    title, fontweight ="bold",fontsize=16)
self.hist_feat_versus_other(self.df,df_target,\nself.df.Citology,['Citology=1','Citology=0'],\
self.widgetPlot2.canvas.axis1)
self.widgetPlot2.canvas.axis1.set_xlabel(title,\n    fontweight ="bold",fontsize=16)

self.widgetPlot2.canvas.figure.tight_layout()
self.widgetPlot2.canvas.draw()

self.widgetPlot1.canvas.figure.clf()
```

```
print(self.df.Biopsy.value_counts())
self.widgetPlot1.canvas.axis1 =
self.widgetPlot1.canvas.figure.add_subplot(211, \
    facecolor = '#fbe7dd')
self.widgetPlot1.canvas.axis1.set_title('Biopsy versus ' + \
    title, fontweight ="bold", fontsize=16)
self.prob_feat_versus_other(self.df,df_target,\ 
self.df.Biopsy,['Biopsy=1','Biopsy=0'],\ 
self.widgetPlot1.canvas.axis1)
self.widgetPlot1.canvas.axis1.set_xlabel(title,\ 
    fontweight ="bold", fontsize=16)

print(self.df["Dx:HPV"].value_counts())
self.widgetPlot1.canvas.axis1 =
self.widgetPlot1.canvas.figure.add_subplot(212, \
    facecolor = '#fbe7dd')
self.widgetPlot1.canvas.axis1.set_title('Dx:HPV versus ' + \
    title, fontweight ="bold", fontsize=16)
self.prob_feat_versus_other(self.df,df_target,\ 
self.df["Dx:HPV"],['Dx:HPV=1','Dx:HPV=0'],\ 
self.widgetPlot1.canvas.axis1)
self.widgetPlot1.canvas.axis1.set_xlabel(title,\ 
    fontweight ="bold", fontsize=16)

self.widgetPlot1.canvas.figure.tight_layout()
self.widgetPlot1.canvas.draw()

def plot_corr(self, data, widget):
    corrdata = data.corr()
    sns.heatmap(corrdata, ax = widget.canvas.axis1, \
        lw=1, annot=True, cmap="Reds")
    widget.canvas.axis1.set_title('Correlation Matrix', \
        fontweight ="bold", fontsize=20)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()
```

```
def choose_plot(self):
    strCB = self.cbData.currentText()

    if strCB == 'target (Biopsy)':
        self.dist_target_versus_features(self.df.Biopsy, \
        "Biopsy", ['Biopsy=0','Biopsy=1'])

    if strCB == 'Hinselmann':
        self.dist_target_versus_features(self.df.Hinselmann, \
        "Hinselmann", ['Hinselmann=0','Hinselmann=1'])

    if strCB == 'Schiller':
        self.dist_target_versus_features(self.df.Schiller, \
        "Schiller", ['Schiller=0','Schiller=1'])

    if strCB == 'Citology':
        self.dist_target_versus_features(self.df.Citology, \
        "Citology", ['Citology=0','Citology=1'])

    if strCB == 'Age':
        self.dist_feat_versus_others(self.df.Age, "Age")

    if strCB == 'Smokes':
        self.dist_feat_versus_others(self.df.Smokes, "Smokes")

    if strCB == 'Smokes (years)':
        self.dist_feat_versus_others(\n
            self.df["Smokes (years)"].apply(pd.to_numeric, \
                errors='coerce').astype('float64'), "Smokes (years)")

    if strCB == 'First sexual intercourse':
        self.dist_feat_versus_others(\n
            self.df["First sexual intercourse"].apply(\n
                pd.to_numeric, errors='coerce').astype('float64'), \
            "First sexual intercourse")
```

```
if strCB == 'Num of pregnancies':  
    self.dist_feat_versus_others(\n        self.df["Num of pregnancies"].apply(\n            pd.to_numeric, errors='coerce').astype('float64'), \  
        "Num of pregnancies")  
  
if strCB == 'Hormonal Contraceptives (years)':  
    self.dist_feat_versus_others(\n        self.df["Hormonal Contraceptives (years)"].apply(\n            pd.to_numeric, errors='coerce').astype('float64'), \  
        "Hormonal Contraceptives (years)")  
  
if strCB == 'IUD (years)':  
    self.dist_feat_versus_others(\n        self.df["IUD (years)"].apply(pd.to_numeric, \  
            errors='coerce').astype('float64'), "IUD (years)")  
  
if strCB == 'Number of sexual partners':  
    self.dist_feat_versus_others(\n        self.df["Number of sexual partners"].apply(\n            pd.to_numeric, errors='coerce').astype('float64'), \  
        "Number of sexual partners")  
  
if strCB == 'Correlation Matrix':  
    self.widgetPlot3.canvas.figure.clf()  
    self.widgetPlot3.canvas.axis1 = \  
    self.widgetPlot3.canvas.figure.add_subplot(111)  
    X,_ = self.fit_dataset(self.df)  
    self.plot_corr(X, self.widgetPlot3)  
  
if strCB == 'Features Importance':  
    self.widgetPlot3.canvas.figure.clf()  
    self.widgetPlot3.canvas.axis1 = \  
    self.widgetPlot3.canvas.figure.add_subplot(111)  
    self.plot_importance(self.widgetPlot3)
```

```

def plot_importance(self, widget):
#Compares different feature importances
    r = ExtraTreesClassifier(random_state=0)
    X,y = self.fit_dataset(self.df)
    r.fit(X, y)
    feature_importance_normalized = np.std(
        [tree.feature_importances_ for tree in r.estimators_], axis = 0)

    sns.barplot(feature_importance_normalized, \
        X.columns, ax = widget.canvas.axis1)
    widget.canvas.axis1.set_ylabel('Feature Labels',\
        fontweight ="bold",fontsize=15)
    widget.canvas.axis1.set_xlabel('Features Importance',\
        fontweight ="bold",fontsize=15)
    widget.canvas.axis1.set_title(\
        'Comparison of different Features Importance',\
        fontweight ="bold",fontsize=20)
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

```

```

def plot_real_pred_val(self, Y_pred, Y_test, widget, title):
#Calculate Metrics
    acc=accuracy_score(Y_test,Y_pred)

#Output plot
    widget.canvas.figure.clf()
    widget.canvas.axis1 = \
        widget.canvas.figure.add_subplot(111,facecolor='steelblue')
    widget.canvas.axis1.scatter(range(len(Y_pred)),\
        Y_pred,color="yellow",lw=5,label="Predictions")
    widget.canvas.axis1.scatter(range(len(Y_test)), \
        Y_test,color="red",label="Actual")
    widget.canvas.axis1.set_title(\
        "Prediction Values vs Real Values of " + title, \
        fontweight ="bold",fontsize=15)
    widget.canvas.axis1.set_xlabel("Accuracy: " + \

```

```

str(round((acc*100),3)) + "%",fontweight ="bold",fontsize=15)
    widget.canvas.axis1.legend()
    widget.canvas.axis1.grid(True, alpha=0.75, lw=1, ls='-.')
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

def plot_cm(self, Y_pred, Y_test, widget, title):
    cm=confusion_matrix(Y_test,Y_pred)
    widget.canvas.figure.clf()
    widget.canvas.axis1 = widget.canvas.figure.add_subplot(111)
    class_label = ["1", "0"]
    df_cm = pd.DataFrame(cm,
index=class_label,columns=class_label)
    sns.heatmap(df_cm, ax=widget.canvas.axis1, annot=True, \
        cmap='plasma', linewidths=2,fmt='d')
    widget.canvas.axis1.set_title("Confusion Matrix of " + title, \
        fontweight ="bold",fontsize=15)
    widget.canvas.axis1.set_xlabel("Predicted")
    widget.canvas.axis1.set_ylabel("True")
    widget.canvas.draw()

def plot_roc(self, clf, xtest, ytest, title, widget):
    pred_prob = clf.predict_proba(xtest)
    pred_prob = pred_prob[:, 1]
    fpr, tpr, thresholds = roc_curve(ytest, pred_prob)
    widget.canvas.axis1.plot(fpr,tpr, label='ANN',\
        color='crimson', linewidth=3)
    widget.canvas.axis1.set_xlabel('False Positive Rate')
    widget.canvas.axis1.set_ylabel('True Positive Rate')
    widget.canvas.axis1.set_title('ROC Curve of ' + title, \
        fontweight ="bold",fontsize=15)
    widget.canvas.axis1.grid(True, alpha=0.75, lw=1, ls='-.')
    widget.canvas.figure.tight_layout()
    widget.canvas.draw()

def plot_decision(self, cla, feat1, feat2, widget, title=""):

```

```

#Plots decision boundary of two features
feat_boundary = [feat1, feat2]
X_feature = self.df[feat_boundary]
X_train_feature, X_test_feature, y_train_feature, \
y_test_feature= train_test_split(X_feature, \
self.df['stroke'], test_size=0.3, random_state = 42)
cla.fit(X_train_feature, y_train_feature)

plot_decision_regions(X_test_feature.values, \
y_test_feature.ravel(), clf=cla, legend=2, \
ax=widget.canvas.axis1)
widget.canvas.axis1.set_title(title, fontweight
="bold", fontsize=15)
widget.canvas.axis1.set_xlabel(feat1)
widget.canvas.axis1.set_ylabel(feat2)
widget.canvas.figure.tight_layout()
widget.canvas.draw()

def plot_learning_curve(self, estimator, title, X, y, widget,
ylim=None, cv=None, n_jobs=None, train_sizes=np.linspace(.1, 1.0,
5)):
    widget.canvas.axis1.set_title(title, fontweight
="bold", fontsize=15)
    if ylim is not None:
        widget.canvas.axis1.set_ylim(*ylim)
        widget.canvas.axis1.set_xlabel("Training examples")
        widget.canvas.axis1.set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = \
        learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
                      train_sizes=train_sizes,
                      return_times=True)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

```

```

# Plot learning curve
    widget.canvas.axis1.grid()
    widget.canvas.axis1.fill_between(train_sizes, \
        train_scores_mean - train_scores_std,\n        train_scores_mean + train_scores_std, alpha=0.1, color="r")
    widget.canvas.axis1.fill_between(train_sizes, \
        test_scores_mean - test_scores_std,\n        test_scores_mean + test_scores_std, alpha=0.1, color="g")
    widget.canvas.axis1.plot(train_sizes, \
        train_scores_mean, 'o-', color="r",\n        label="Training score")
    widget.canvas.axis1.plot(train_sizes, \
        test_scores_mean, 'o-', color="g",\n        label="Cross-validation score")
    widget.canvas.axis1.legend(loc="best")

def plot_scalability_curve(self, estimator, title, X, y, widget,
    ylim=None, cv=None, n_jobs=None, train_sizes=np.linspace(.1, 1.0,
5)):
    widget.canvas.axis1.set_title(title, fontweight
    ="bold", fontsize=15)
    if ylim is not None:
        widget.canvas.axis1.set_ylim(*ylim)
        widget.canvas.axis1.set_xlabel("Training examples")
        widget.canvas.axis1.set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = \
        learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
            train_sizes=train_sizes,
            return_times=True)
    fit_times_mean = np.mean(fit_times, axis=1)
    fit_times_std = np.std(fit_times, axis=1)

# Plot n_samples vs fit_times
    widget.canvas.axis1.grid()

```

```

        widget.canvas.axis1.plot(train_sizes, fit_times_mean, 'o-')
        widget.canvas.axis1.fill_between(train_sizes, \
            fit_times_mean - fit_times_std,\ 
            fit_times_mean + fit_times_std, alpha=0.1)
        widget.canvas.axis1.set_xlabel("Training examples")
        widget.canvas.axis1.set_ylabel("fit_times")

def plot_performance_curve(self, estimator, title, X, y, widget,
    ylim=None, cv=None, n_jobs=None, train_sizes=np.linspace(.1, 1.0,
5)):
    widget.canvas.axis1.set_title(title, fontweight
    ="bold", fontsize=15)
    if ylim is not None:
        widget.canvas.axis1.set_ylim(*ylim)
        widget.canvas.axis1.set_xlabel("Training examples")
        widget.canvas.axis1.set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = \
        learning_curve(estimator, X, y, cv=cv, n_jobs=n_jobs,
            train_sizes=train_sizes,
            return_times=True)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    fit_times_mean = np.mean(fit_times, axis=1)

    # Plot n_samples vs fit_times
    widget.canvas.axis1.grid()
    widget.canvas.axis1.plot(fit_times_mean, test_scores_mean, 'o-')
    widget.canvas.axis1.fill_between(fit_times_mean, \
        test_scores_mean - test_scores_std,\ 
        test_scores_mean + test_scores_std, alpha=0.1)
    widget.canvas.axis1.set_xlabel("fit_times")
    widget.canvas.axis1.set_ylabel("Score")

def train_model(self, model, X, y):
    model.fit(X, y)

```

```
return model

def predict_model(self, model, X, proba=False):
    if ~proba:
        y_pred = model.predict(X)
    else:
        y_pred_proba = model.predict_proba(X)
        y_pred = np.argmax(y_pred_proba, axis=1)

    return y_pred

def run_model(self, name, scaling, model, X_train, X_test, y_train,
             y_test, train=True, proba=True):
    if train == True:
        model = self.train_model(model, X_train, y_train)
        y_pred = self.predict_model(model, X_test, proba)

        accuracy = accuracy_score(y_test, y_pred)
        recall = recall_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred)
```

```
print('accuracy: ', accuracy)
print('recall: ', recall)
print('precision: ', precision)
print('f1: ', f1)
print(classification_report(y_test, y_pred))

self.widgetPlot1.canvas.figure.clf()
self.widgetPlot1.canvas.axis1 = \
self.widgetPlot1.canvas.figure.add_subplot(111, \
    facecolor = '#fbe7dd')
self.plot_cm(y_pred, y_test, self.widgetPlot1, \
    name + " -- " + scaling)
self.widgetPlot1.canvas.figure.tight_layout()
self.widgetPlot1.canvas.draw()

self.widgetPlot2.canvas.figure.clf()
self.widgetPlot2.canvas.axis1 = \
self.widgetPlot2.canvas.figure.add_subplot(111, \
    facecolor = '#fbe7dd')
self.plot_real_pred_val(y_pred, y_test, self.widgetPlot2, \
    name + " -- " + scaling)
self.widgetPlot2.canvas.figure.tight_layout()
self.widgetPlot2.canvas.draw()

self.widgetPlot3.canvas.figure.clf()
self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(221, \
    facecolor = '#fbe7dd')
self.plot_roc(model, X_test, y_test, \
    name + " -- " + scaling, self.widgetPlot3)
self.widgetPlot3.canvas.figure.tight_layout()

self.widgetPlot3.canvas.axis1 = \
self.widgetPlot3.canvas.figure.add_subplot(222, \
```

```
        facecolor = '#fbe7dd')
self.plot_learning_curve(model, 'Learning Curve' + " -- " + \
    scaling, X_train, y_train, self.widgetPlot3)
self.widgetPlot3.canvas.figure.tight_layout()

self.widgetPlot3.canvas.axis1 =
self.widgetPlot3.canvas.figure.add_subplot(223,\n
    facecolor = '#fbe7dd')
self.plot_scalability_curve(model, 'Scalability of ' + name \
    + " -- " + scaling, X_train, y_train, self.widgetPlot3)
self.widgetPlot3.canvas.figure.tight_layout()

self.widgetPlot3.canvas.axis1 =
self.widgetPlot3.canvas.figure.add_subplot(224,\n
    facecolor = '#fbe7dd')
self.plot_performance_curve(model,\n
'Performance of ' + name + " -- " + scaling, X_train,\n
    y_train, self.widgetPlot3)
self.widgetPlot3.canvas.figure.tight_layout()

self.widgetPlot3.canvas.draw()

def build_train_lr(self):
if path.isfile('logregRaw.pkl'):
#Loads model
    self.logregRaw = joblib.load('logregRaw.pkl')
    self.logregNorm = joblib.load('logregNorm.pkl')
    self.logregStand = joblib.load('logregStand.pkl')

if self.rbRaw.isChecked():
    self.run_model('Logistic Regression', 'Raw', \
self.logregRaw, self.X_train_raw, self.X_test_raw, \
self.y_train_raw, self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Logistic Regression', 'Normalization', \
self.logregNorm, self.X_train_norm, self.X_test_norm,\n
```

```
self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Logistic Regression', 'Standardization', \
        self.logregStand, self.X_train_stand, \
        self.X_test_stand, self.y_train_stand, \
        self.y_test_stand)

else:
    #Builds and trains Logistic Regression
    self.logregRaw = LogisticRegression(solver='lbfgs', \
        max_iter=1000, random_state=2021)
    self.logregNorm = LogisticRegression(solver='lbfgs', \
        max_iter=1000, random_state=2021)
    self.logregStand = LogisticRegression(solver='lbfgs', \
        max_iter=1000, random_state=2021)

if self.rbRaw.isChecked():
    self.run_model('Logistic Regression', 'Raw', \
        self.logregRaw, self.X_train_raw, self.X_test_raw, \
        self.y_train_raw, self.y_test_raw)
    if self.rbNorm.isChecked():
        self.run_model('Logistic Regression', \
            'Normalization', self.logregNorm, \
            self.X_train_norm, self.X_test_norm, \
            self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Logistic Regression', \
        'Standardization', self.logregStand, \
        self.X_train_stand, self.X_test_stand, \
        self.y_train_stand, self.y_test_stand)

#Saves model
    joblib.dump(self.logregRaw, 'logregRaw.pkl')
    joblib.dump(self.logregNorm, 'logregNorm.pkl')
```

```
joblib.dump(self.logregStand, 'logregStand.pkl')

def choose_ML_model(self):
    strCB = self.cbClassifier.currentText()

    if strCB == 'Logistic Regression':
        self.build_train_lr()

    if strCB == 'Support Vector Machine':
        self.build_train_svm()

    if strCB == 'K-Nearest Neighbor':
        self.build_train_knn()

    if strCB == 'Decision Tree':
        self.build_train_dt()

    if strCB == 'Random Forest':
        self.build_train_rf()

    if strCB == 'Gradient Boosting':
        self.build_train_gb()

    if strCB == 'Naive Bayes':
        self.build_train_nb()

    if strCB == 'Adaboost':
        self.build_train_ada()

    if strCB == 'XGB Classifier':
        self.build_train_xgb()

    if strCB == 'LGBM Classifier':
        self.build_train_lgbm()

    if strCB == 'MLP Classifier':
```

```
self.build_train_mlp()

def build_train_svm(self):
    if path.isfile('SVMRaw.pkl'):
        #Loads model
        self.SVMRaw = joblib.load('SVMRaw.pkl')
        self.SVMNorm = joblib.load('SVMNorm.pkl')
        self.SVMStand = joblib.load('SVMStand.pkl')

        if self.rbRaw.isChecked():
            self.run_model('Support Vector Machine', 'Raw', \
                self.SVMRaw, self.X_train_raw, self.X_test_raw, \
                self.y_train_raw, self.y_test_raw)
        if self.rbNorm.isChecked():
            self.run_model('Support Vector Machine', \
                'Normalization', self.SVMNorm, self.X_train_norm, \
                self.X_test_norm, self.y_train_norm, self.y_test_norm)

        if self.rbStand.isChecked():
            self.run_model('Support Vector Machine', \
                'Standardization', self.SVMStand, self.X_train_stand, \
                self.X_test_stand, self.y_train_stand, \
                self.y_test_stand)

    else:
        #Builds and trains Logistic Regression
        self.SVMRaw = SVC(random_state=2021,probability=True)
        self.SVMNorm = SVC(random_state=2021,probability=True)
        self.SVMStand = SVC(random_state=2021,probability=True)

        if self.rbRaw.isChecked():
            self.run_model('Support Vector Machine', 'Raw', \
                self.SVMRaw, self.X_train_raw, self.X_test_raw, \
                self.y_train_raw, self.y_test_raw)
        if self.rbNorm.isChecked():
            self.run_model('Support Vector Machine', 'Normalization', \
```

```
self.SVMNorm, self.X_train_norm, self.X_test_norm, \
self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Support Vector Machine', \
'Standardization', self.SVMStand, self.X_train_stand, \
self.X_test_stand, self.y_train_stand, self.y_test_stand)

#Saves model
    joblib.dump(self.SVMRaw, 'SVMRaw.pkl')
    joblib.dump(self.SVMNorm, 'SVMNorm.pkl')
    joblib.dump(self.SVMStand, 'SVMStand.pkl')

def build_train_knn(self):
if path.isfile('KNNRaw.pkl'):
    #Loads model
    self.KNNRaw = joblib.load('KNNRaw.pkl')
    self.KNNNorm = joblib.load('KNNNorm.pkl')
    self.KNNStand = joblib.load('KNNStand.pkl')

if self.rbRaw.isChecked():
    self.run_model('K-Nearest Neighbor', 'Raw', \
self.KNNRaw, self.X_train_raw, self.X_test_raw, \
self.y_train_raw, self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('K-Nearest Neighbor', 'Normalization', \
self.KNNNorm, self.X_train_norm, self.X_test_norm, \
self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('K-Nearest Neighbor', 'Standardization', \
self.KNNStand, self.X_train_stand, self.X_test_stand, \
self.y_train_stand, self.y_test_stand)

else:
    #Builds and trains K-Nearest Neighbor
```

```

self.KNNRaw = KNeighborsClassifier(n_neighbors = 50)
self.KNNNorm = KNeighborsClassifier(n_neighbors = 50)
self.KNNSStand = KNeighborsClassifier(n_neighbors = 50)

if self.rbRaw.isChecked():
    self.run_model('K-Nearest Neighbor', 'Raw', \
    self.KNNRaw, self.X_train_raw, self.X_test_raw, \
    self.y_train_raw, self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('K-Nearest Neighbor', 'Normalization', \
    self.KNNNorm, self.X_train_norm, self.X_test_norm, \
    self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('K-Nearest Neighbor', 'Standardization', \
    self.KNNSStand, self.X_train_stand, self.X_test_stand, \
    self.y_train_stand, self.y_test_stand)

#Saves model
joblib.dump(self.KNNRaw, 'KNNRaw.pkl')
joblib.dump(self.KNNNorm, 'KNNNorm.pkl')
joblib.dump(self.KNNSStand, 'KNNSStand.pkl')

def build_train_dt(self):
if path.isfile('DTRaw.pkl'):
#Loads model
self.DTRaw = joblib.load('DTRaw.pkl')
self.DTNorm = joblib.load('DTNorm.pkl')
self.DTStand = joblib.load('DTStand.pkl')

if self.rbRaw.isChecked():
    self.run_model('Decision Tree', 'Raw', \
    self.DTRaw, self.X_train_raw, self.X_test_raw, \
    self.y_train_raw, self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Decision Tree', 'Normalization', \

```

```
self.DTNorm, self.X_train_norm, self.X_test_norm, \
self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Decision Tree', 'Standardization', \
    self.DTStand, self.X_train_stand, self.X_test_stand, \
    self.y_train_stand, self.y_test_stand)

else:
    #Builds and trains Decision Tree
    dt = DecisionTreeClassifier()
    parameters = { \
        'max_depth':np.arange(1,5,1), 'random_state':[2021]}
    self.DTRaw = GridSearchCV(dt, parameters)
    self.DTNorm = GridSearchCV(dt, parameters)
    self.DTStand = GridSearchCV(dt, parameters)

if self.rbRaw.isChecked():
    self.run_model('Decision Tree', 'Raw', \
    self.DTRaw, self.X_train_raw, self.X_test_raw, \
    self.y_train_raw, self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Decision Tree', 'Normalization', \
    self.DTNorm, self.X_train_norm, self.X_test_norm, \
    self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Decision Tree', 'Standardization', \
    self.DTStand, self.X_train_stand, self.X_test_stand, \
    self.y_train_stand, self.y_test_stand)

#Saves model
    joblib.dump(self.DTRaw, 'DTRaw.pkl')
    joblib.dump(self.DTNorm, 'DTNorm.pkl')
    joblib.dump(self.DTStand, 'DTStand.pkl')
```

```
def build_train_rf(self):
    if path.isfile('RFRaw.pkl'):
        #Loads model
        self.RFRaw = joblib.load('RFRaw.pkl')
        self.RFNorm = joblib.load('RFNorm.pkl')
        self.RFStand = joblib.load('RFStand.pkl')

        if self.rbRaw.isChecked():
            self.run_model('Random Forest', 'Raw', self.RFRaw, \
                          self.X_train_raw, self.X_test_raw, self.y_train_raw, \
                          self.y_test_raw)
        if self.rbNorm.isChecked():
            self.run_model('Random Forest', 'Normalization', \
                          self.RFNorm, self.X_train_norm, self.X_test_norm, \
                          self.y_train_norm, self.y_test_norm)

        if self.rbStand.isChecked():
            self.run_model('Random Forest', 'Standardization', \
                          self.RFStand, self.X_train_stand, self.X_test_stand, \
                          self.y_train_stand, self.y_test_stand)

    else:
        #Builds and trains Random Forest
        self.RFRaw = RandomForestClassifier(n_estimators=200, \
                                             max_depth=2, random_state=2021)
        self.RFNorm = RandomForestClassifier(n_estimators=200, \
                                             max_depth=2, random_state=2021)
        self.RFStand = RandomForestClassifier(n_estimators=200, \
                                             max_depth=2, random_state=2021)

        if self.rbRaw.isChecked():
            self.run_model('Random Forest', 'Raw', self.RFRaw, \
                          self.X_train_raw, self.X_test_raw, self.y_train_raw, \
                          self.y_test_raw)
        if self.rbNorm.isChecked():
            self.run_model('Random Forest', 'Normalization', \
```

```
self.RFNorm, self.X_train_norm, self.X_test_norm, \
self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Random Forest', 'Standardization', \
        self.RFStand, self.X_train_stand, self.X_test_stand, \
        self.y_train_stand, self.y_test_stand)

#Saves model
    joblib.dump(self.RFRaw, 'RFRaw.pkl')
    joblib.dump(self.RFNorm, 'RFNorm.pkl')
    joblib.dump(self.RFStand, 'RFStand.pkl')

def build_train_gb(self):
if path.isfile('GBRaw.pkl'):
    #Loads model
    self.GBRaw = joblib.load('GBRaw.pkl')
    self.GBNorm = joblib.load('GBNorm.pkl')
    self.GBStand = joblib.load('GBStand.pkl')

if self.rbRaw.isChecked():
    self.run_model('Gradient Boosting', 'Raw', self.GBRaw, \
        self.X_train_raw, self.X_test_raw, self.y_train_raw, \
        self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Gradient Boosting', 'Normalization', \
        self.GBNorm, self.X_train_norm, self.X_test_norm, \
        self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Gradient Boosting', 'Standardization', \
        self.GBStand, self.X_train_stand, self.X_test_stand, \
        self.y_train_stand, self.y_test_stand)

else:
    #Builds and trains Gradient Boosting
```

```
self.GBRaw = GradientBoostingClassifier(n_estimators = 200, \
    max_depth=3, subsample=0.8, max_features=0.2, \
    random_state=2021)
self.GBNorm = GradientBoostingClassifier(n_estimators = 200, \
    max_depth=3, subsample=0.8, max_features=0.2, \
    random_state=2021)
self.GBStand = GradientBoostingClassifier(n_estimators = 200, \
    max_depth=3, subsample=0.8, max_features=0.2, \
    random_state=2021)

if self.rbRaw.isChecked():
    self.run_model('Gradient Boosting', 'Raw', \
        self.GBRaw, self.X_train_raw, self.X_test_raw, \
        self.y_train_raw, self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Gradient Boosting', 'Normalization', \
        self.GBNorm, self.X_train_norm, self.X_test_norm, \
        self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Gradient Boosting', 'Standardization', \
        self.GBStand, self.X_train_stand, self.X_test_stand, \
        self.y_train_stand, self.y_test_stand)

#Saves model
    joblib.dump(self.GBRaw, 'GBRaw.pkl')
    joblib.dump(self.GBNorm, 'GBNorm.pkl')
    joblib.dump(self.GBStand, 'GBStand.pkl')

def build_train_nb(self):
    if path.isfile('NBRaw.pkl'):
        #Loads model
        self.NBRaw = joblib.load('NBRaw.pkl')
        self.NBNorm = joblib.load('NBNorm.pkl')
        self.NBStand = joblib.load('NBStand.pkl')
```

```
if self.rbRaw.isChecked():
    self.run_model('Naive Bayes', 'Raw', self.NBRaw, \
    self.X_train_raw, self.X_test_raw, self.y_train_raw, \
    self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Naive Bayes', 'Normalization', \
    self.NBNorm, self.X_train_norm, self.X_test_norm, \
    self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Naive Bayes', 'Standardization', \
    self.NBStand, self.X_train_stand, self.X_test_stand, \
    self.y_train_stand, self.y_test_stand)

else:
    #Builds and trains Naive Bayes
    self.NBRaw = GaussianNB()
    self.NBNorm = GaussianNB()
    self.NBStand = GaussianNB()

if self.rbRaw.isChecked():
    self.run_model('Naive Bayes', 'Raw', self.NBRaw, \
    self.X_train_raw, self.X_test_raw, self.y_train_raw, \
    self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Naive Bayes', 'Normalization', \
    self.NBNorm, self.X_train_norm, self.X_test_norm, \
    self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Naive Bayes', 'Standardization', \
    self.NBStand, self.X_train_stand, self.X_test_stand, \
    self.y_train_stand, self.y_test_stand)

#Saves model
joblib.dump(self.NBRaw, 'NBRaw.pkl')
```

```
joblib.dump(self.NBNorm, 'NBNorm.pkl')
joblib.dump(self.NBStand, 'NBStand.pkl')

def build_train_ada(self):
if path.isfile('ADARaw.pkl'):
#Loads model
    self.ADARaw = joblib.load('ADARaw.pkl')
    self.ADANorm = joblib.load('ADANorm.pkl')
    self.ADAStand = joblib.load('ADAStand.pkl')

if self.rbRaw.isChecked():
    self.run_model('Adaboost', 'Raw', self.ADARaw, \
        self.X_train_raw, self.X_test_raw, self.y_train_raw, \
        self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Adaboost', 'Normalization', \
        self.ADANorm, self.X_train_norm, self.X_test_norm, \
        self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Adaboost', 'Standardization', \
        self.ADAStand, self.X_train_stand, self.X_test_stand, \
        self.y_train_stand, self.y_test_stand)

else:
#Builds and trains Adaboost
    self.ADARaw = AdaBoostClassifier(n_estimators = 100, \
        learning_rate=0.01)
    self.ADANorm = AdaBoostClassifier(n_estimators = 100, \
        learning_rate=0.01)
    self.ADAStand = AdaBoostClassifier(n_estimators = 100, \
        learning_rate=0.01)

if self.rbRaw.isChecked():
    self.run_model('Adaboost', 'Raw', self.ADARaw, \
        self.X_train_raw, self.X_test_raw, self.y_train_raw, \
```

```
self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('Adaboost', 'Normalization', \
        self.ADANorm, self.X_train_norm, self.X_test_norm, \
        self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('Adaboost', 'Standardization', \
        self.ADAStand, self.X_train_stand, \
        self.X_test_stand, self.y_train_stand, self.y_test_stand)

#Saves model
joblib.dump(self.ADARaw, 'ADARaw.pkl')
joblib.dump(self.ADANorm, 'ADANorm.pkl')
joblib.dump(self.ADAStand, 'ADAStand.pkl')

def build_train_xgb(self):
if path.isfile('XGBRaw.pkl'):
    #Loads model
    self.XGBRaw = joblib.load('XGBRaw.pkl')
    self.XGBNorm = joblib.load('XGBNorm.pkl')
    self.XGBStand = joblib.load('XGBStand.pkl')

if self.rbRaw.isChecked():
    self.run_model('XGB', 'Raw', self.XGBRaw, \
        self.X_train_raw, self.X_test_raw, self.y_train_raw, \
        self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('XGB', 'Normalization', self.XGBNorm, \
        self.X_train_norm, self.X_test_norm, self.y_train_norm, \
        self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('XGB', 'Standardization', \
        self.XGBStand, self.X_train_stand, self.X_test_stand, \
        self.y_train_stand, self.y_test_stand)
```

```
else:
#Builds and trains XGB classifier
self.XGBRaw = XGBClassifier(n_estimators = 200, \
    max_depth=2, random_state=2021,
use_label_encoder=False,\ 
    eval_metric='mlogloss')
self.XGBNorm = XGBClassifier(n_estimators = 200, \
    max_depth=2, random_state=2021,
use_label_encoder=False,\ 
    eval_metric='mlogloss')
self.XGBStand = XGBClassifier(n_estimators = 200, \
    max_depth=2, random_state=2021,
use_label_encoder=False,\ 
    eval_metric='mlogloss')

if self.rbRaw.isChecked():
self.run_model('XGB', 'Raw', self.XGBRaw, \
self.X_train_raw, self.X_test_raw, \
self.y_train_raw, self.y_test_raw)
if self.rbNorm.isChecked():
self.run_model('XGB', 'Normalization', self.XGBNorm, \
self.X_train_norm, self.X_test_norm, \
self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
self.run_model('XGB', 'Standardization', self.XGBStand, \
self.X_train_stand, self.X_test_stand, \
self.y_train_stand, self.y_test_stand)

#Saves model
joblib.dump(self.XGBRaw, 'XGBRaw.pkl')
joblib.dump(self.XGBNorm, 'XGBNorm.pkl')
joblib.dump(self.XGBStand, 'XGBStand.pkl')

def build_train_lgbm(self):
```

```
if path.isfile('LGBMRaw.pkl'):
    #Loads model
    self.LGBMRaw = joblib.load('LGBMRaw.pkl')
    self.LGBMNorm = joblib.load('LGBMNorm.pkl')
    self.LGBMStand = joblib.load('LGBMStand.pkl')

if self.rbRaw.isChecked():
    self.run_model('LGBM Classifier', 'Raw', self.LGBMRaw, \
    self.X_train_raw, self.X_test_raw, self.y_train_raw, \
    self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('LGBM Classifier', 'Normalization', \
    self.LGBMNorm, self.X_train_norm, self.X_test_norm, \
    self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('LGBM Classifier', 'Standardization', \
    self.LGBMStand, self.X_train_stand, self.X_test_stand, \
    self.y_train_stand, self.y_test_stand)

else:
    #Builds and trains LGBMClassifier classifier
    self.LGBMRaw = LGBMClassifier(max_depth = 2, \
        n_estimators=500, subsample=0.8, random_state=2021)
    self.LGBMNorm = LGBMClassifier(max_depth = 2, \
        n_estimators=500, subsample=0.8, random_state=2021)
    self.LGBMStand = LGBMClassifier(max_depth = 2, \
        n_estimators=500, subsample=0.8, random_state=2021)

if self.rbRaw.isChecked():
    self.run_model('LGBM Classifier', 'Raw', self.LGBMRaw, \
    self.X_train_raw, self.X_test_raw, self.y_train_raw, \
    self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('LGBM Classifier', 'Normalization', \
    self.LGBMNorm, self.X_train_norm, self.X_test_norm, \
```

```
self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('LGBM Classifier', 'Standardization', \
        self.LGBMStand, self.X_train_stand, self.X_test_stand, \
        self.y_train_stand, self.y_test_stand)

#Saves model
    joblib.dump(self.LGBMRaw, 'LGBMRaw.pkl')
    joblib.dump(self.LGBMNorm, 'LGBMNorm.pkl')
    joblib.dump(self.LGBMStand, 'LGBMStand.pkl')

def build_train_mlp(self):
    if path.isfile('MLPRaw.pkl'):
        #Loads model
        self.MLPRaw = joblib.load('MLPRaw.pkl')
        self.MLPNorm = joblib.load('MLPNorm.pkl')
        self.MLPStand = joblib.load('MLPStand.pkl')

    if self.rbRaw.isChecked():
        self.run_model('MLP Classifier', 'Raw', self.MLPRaw, \
            self.X_train_raw, self.X_test_raw, self.y_train_raw, \
            self.y_test_raw)
    if self.rbNorm.isChecked():
        self.run_model('MLP Classifier', 'Normalization', \
            self.MLPNorm, self.X_train_norm, self.X_test_norm, \
            self.y_train_norm, self.y_test_norm)

    if self.rbStand.isChecked():
        self.run_model('MLP Classifier', 'Standardization', \
            self.MLPStand, self.X_train_stand, self.X_test_stand, \
            self.y_train_stand, self.y_test_stand)

else:
    #Builds and trains MLP classifier
    self.MLPRaw = MLPClassifier(random_state=2021)
```

```

self.MLPNorm = MLPClassifier(random_state=2021)
self.MLPStand = MLPClassifier(random_state=2021)

if self.rbRaw.isChecked():
    self.run_model('MLP Classifier', 'Raw', self.MLPRaw, \
    self.X_train_raw, self.X_test_raw, self.y_train_raw, \
    self.y_test_raw)
if self.rbNorm.isChecked():
    self.run_model('MLP Classifier', 'Normalization', \
    self.MLPNorm, self.X_train_norm, self.X_test_norm, \
    self.y_train_norm, self.y_test_norm)

if self.rbStand.isChecked():
    self.run_model('MLP Classifier', 'Standardization', \
    self.MLPStand, self.X_train_stand, self.X_test_stand, \
    self.y_train_stand, self.y_test_stand)

#Saves model
    joblib.dump(self.MLPRaw, 'MLPRaw.pkl')
    joblib.dump(self.MLPNorm, 'MLPNorm.pkl')
    joblib.dump(self.MLPStand, 'MLPStand.pkl')

def train_test_ANN(self):
    X = self.df.drop('Biopsy', axis=1).apply(pd.to_numeric, \
        errors='coerce').astype('float64')
    Y = self.df["Biopsy"]
    sm = SMOTE(random_state=42)
    X, Y = sm.fit_resample(X, Y.ravel())

#Splits dataframe into X_train, X_test, y_train and y_test
    X_train, X_test, y_train_DL, y_test_DL = train_test_split(X, \
        Y, test_size=0.2, random_state=2021)

#Deletes any outliers in the data using StandardScaler from
SKLearn
    sc = StandardScaler()

```

```

X_train_DL = sc.fit_transform(X_train)
X_test_DL = sc.transform(X_test)

#Saves data
np.save('X_train_DL.npy', X_train_DL)
np.save('X_test_DL.npy', X_test_DL)
np.save('y_train_DL.npy', y_train_DL)
np.save('y_test_DL.npy', y_test_DL)

def train_ANN(self):
if path.isfile('X_train_DL.npy'):
#Loads files
    self.X_train_DL = np.load('X_train_DL.npy',allow_pickle=True)
    self.X_test_DL = np.load('X_test_DL.npy',allow_pickle=True)
    self.y_train_DL = np.load('y_train_DL.npy',allow_pickle=True)
    self.y_test_DL = np.load('y_test_DL.npy',allow_pickle=True)

else:
    self.train_test_ANN()
#Loads files
    self.X_train_DL = np.load('X_train_DL.npy',allow_pickle=True)
    self.X_test_DL = np.load('X_test_DL.npy',allow_pickle=True)
    self.y_train_DL = np.load('y_train_DL.npy',allow_pickle=True)
    self.y_test_DL = np.load('y_test_DL.npy',allow_pickle=True)

if path.isfile('cervical_model.h5') == False:
    self.build_ANN(self.X_train_DL, self.y_train_DL, 32, 250)

#Turns on cbPredictionDL
self.cbPredictionDL.setEnabled(True)

#Turns off pbTrainDL
self.pbTrainDL.setEnabled(False)

def build_ANN(self,X_train, y_train, NBATCH, NEPOCH):

```

```
#Imports Tensorflow and create a Sequential Model to add layer  
for the ANN  
    ann = tf.keras.models.Sequential()  
  
#Input layer  
    ann.add(tf.keras.layers.Dense(units=500,  
        input_dim=33,  
        kernel_initializer='uniform',  
        activation='relu'))  
    ann.add(tf.keras.layers.Dropout(0.5))  
  
#Hidden layer 1  
    ann.add(tf.keras.layers.Dense(units=200,  
        kernel_initializer='uniform',  
        activation='relu'))  
    ann.add(tf.keras.layers.Dropout(0.5))  
  
#Output layer  
    ann.add(tf.keras.layers.Dense(units=1,  
        kernel_initializer='uniform',  
        activation='sigmoid'))  
  
print(ann.summary()) #for showing the structure and parameters  
  
#Compiles the ANN using ADAM optimizer.  
    ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', \  
        metrics = ['accuracy'])  
  
#Trains the ANN with 100 epochs.  
    history = ann.fit(X_train, y_train, batch_size = 64, \  
        validation_split=0.20, epochs = 250, shuffle=True)  
  
#Saves model  
    ann.save('cervical_model.h5')  
  
#Saves history into npy file
```

```
    np.save('cervical_history.npy', history.history)

def choose_prediction ANN(self):
    strCB = self.cbPredictionDL.currentText()

    if strCB == 'CNN 1D':
        pred_val = self.pred_ann(self.X_test_DL, self.y_test_DL)

        #Plots true values versus predicted values
        self.widgetPlot2.canvas.figure.clf()
        self.widgetPlot2.canvas.axis1 =
        self.widgetPlot2.canvas.figure.add_subplot(111,
            facecolor = '#fbe7dd')
        self.plot_real_pred_val(pred_val, \
        self.y_test_DL, self.widgetPlot2, 'CNN 1D')
        self.widgetPlot2.canvas.figure.tight_layout()
        self.widgetPlot2.canvas.draw()

        #Plot confusion matrix
        self.widgetPlot1.canvas.figure.clf()
        self.widgetPlot1.canvas.axis1 =
        self.widgetPlot1.canvas.figure.add_subplot(111,
            facecolor = '#fbe7dd')
        self.plot_cm(pred_val, self.y_test_DL, \
        self.widgetPlot1, 'CNN 1D')
        self.widgetPlot1.canvas.figure.tight_layout()
        self.widgetPlot1.canvas.draw()

    #Loads history
    history = np.load('cervical_history.npy',\
        allow_pickle=True).item()
    train_loss = history['loss']
    train_acc = history['accuracy']
    val_acc = history['val_accuracy']
    val_loss = history['val_loss']
    self.plot_loss_acc(train_loss, val_loss, train_acc, \
```

```

    val_acc, self.widgetPlot3, "History of " + strCB)

def plot_loss_acc(self,train_loss, val_loss, train_acc, val_acc,
widget,strPlot):
    widget.canvas.figure.clf()
    widget.canvas.axis1 = \
        widget.canvas.figure.add_subplot(211, facecolor='#fbe7dd')
    widget.canvas.axis1.plot(train_loss, \
        label='Training Loss', color='blue', linewidth=3.0)
    widget.canvas.axis1.plot(val_loss, 'b--', \
        label='Validation Loss', color='red', linewidth=3.0)
    widget.canvas.axis1.set_title('Loss', fontweight
    ="bold", fontsize=20)
    widget.canvas.axis1.set_xlabel('Epoch')
    widget.canvas.axis1.grid(True, alpha=0.75, lw=1, ls='-.')
    widget.canvas.axis1.legend(fontsize=16)

    widget.canvas.axis1 = \
        widget.canvas.figure.add_subplot(212, facecolor='#fbe7dd')
    widget.canvas.axis1.plot(train_acc, \
        label='Training Accuracy', color='blue', linewidth=3.0)
    widget.canvas.axis1.plot(val_acc, 'b--', \
        label='Validation Accuracy', color='red', linewidth=3.0)
    widget.canvas.axis1.set_title('Accuracy', \
        fontweight ="bold", fontsize=20)
    widget.canvas.axis1.set_xlabel('Epoch')
    widget.canvas.axis1.grid(True, alpha=0.75, lw=1, ls='-.')
    widget.canvas.axis1.legend(fontsize=16)
    widget.canvas.draw()

def pred_ann(self, xtest, ytest):
    self.train_ANN()
    self.ann = load_model('cervical_model.h5')

    prediction = self.ann.predict(xtest)
    label = [int(p>=0.5) for p in prediction]

```

```
print("test_target:", ytest)
print("pred_val:", label)

#Performance Evaluation - Accuracy, Classification Report &
Confusion Matrix
#Accuracy Score
print ('Accuracy Score : ', accuracy_score(label, ytest), '\n')

#precision, recall report
print ('Classification Report :\n\n',\
classification_report(label, ytest))

return label

if __name__ == '__main__':
    import sys
    app = QApplication(sys.argv)
    ex = DemoGUI_Cervical()
    ex.show()
    sys.exit(app.exec_())
```