



Department of Statistics



Maratha Vidya Prasarak Samaj's

K.R.T. Arts, B.H. Commerce & A.M. Science (**KTHM**) College, Nashik.

Affiliated to

Savitribai Phule Pune University. (SPPU)

A project Report on

Thyroid Disease Detection Using Machine Learning Techniques

Presented By

- Nikumbh Kalpesh Sanjay
- Khairnar Krushnai Chandrakant
- Bhamare Rutuja Nanaji

Under the Guidance Of

Prof. Mansi Hiray,

Department of Statistics, K.T.H.M College, Nashik

In Partial Fulfilment of degree M.Sc (Statistics)

Academic Year 2022-2023

Certificate

This is to certify that the project entitled “**Thyroid Disease Detection Using Machine Learning Techniques**” is being submitted by Nikumbh Kalpesh Sanjay, Khairnar Krushnai Chandrakant, Bhamare Rutuja Nanaji as partial fulfillment for the award of the degree of the Master of Science (Statistics) at K.R.T. Arts, B. H. Commerce, A. M. Science College, Nashik.

This is a record of their considerable work under my supervision and guidance.

Place: Nashik

Date: 21/06/2023

Signature Valid

Digitally signed by Project Guide,
Mansi Hirey Ma'am
department Of Statistics,
KTHM college Nashik.
Date: 21/06/2023

Signature Valid

Digitally signed by HOD,
Dr. G. S. Phad
department Of Statistics,
KTHM college Nashik.
Date: 21/06/2023

Acknowledgment

We have satisfaction with the completion of our project entitled **“Thyroid Disease Detection Using Machine Learning Technique”** at the Department of Statistics in “M.V.P. Samaj’s K.R.T. Arts, B.H. Commerce, A.M. Science College, Nashik” during the academic year of 2022-2023.

We owe Mansi Hiray our gratitude for his direction and persistent supervision, as well as for providing the project with essential knowledge and for helping us finish the project. We were able to finish the assignment on time because of his consistent direction and desire to impart his wide knowledge.

We would like to show our gratitude to Dr. G. S. Phad, Head of the Statistics Department, for allowing us to work on this project and providing us with all the support we needed.

Additionally, we appreciate the invaluable advice and excellent cooperation from all the faculty in the Department of Statistics.

Abstract

Thyroid disorders are prevalent worldwide, affecting millions of individuals. Early and accurate detection of thyroid dysfunction is crucial for timely intervention and effective management.

In this project, we propose a machine learning-based approach for thyroid detection, leveraging the power of data-driven techniques to enhance diagnostic accuracy.

The data is carefully pre-processed to handle missing values and outliers ensuring its suitability for subsequent analysis.

Multiple machine learning algorithms, such as decision trees, support vector machines, and others, are employed to build predictive models.

This project contributes to the field of medical diagnostics by showcasing the potential of machine learning techniques in improving the detection and diagnosis of thyroid conditions.

Introduction

Thyroid disorders, including hypothyroidism, hyperthyroidism, and thyroid nodules, pose a significant health concern globally. These conditions can lead to various complications if left undiagnosed or untreated. Timely detection and accurate diagnosis are vital for effective management and improving patient outcomes. In recent years, machine learning techniques have shown great potential in the field of medical diagnostics, offering powerful tools for automated detection and classification of diseases. In this project, we aim to leverage machine learning algorithms to develop a robust and accurate system for thyroid detection.

Traditionally, thyroid disorders have been diagnosed through a combination of clinical evaluation, physical examination, and laboratory tests, such as thyroid function tests and imaging studies. While these methods are valuable, they often rely on subjective interpretation and can be time-consuming. Moreover, subtle patterns and relationships within the data might be difficult for human experts to identify.

Machine learning algorithms provide an opportunity to overcome these limitations by utilizing computational power to automatically learn patterns and make predictions from large datasets. By training on comprehensive and diverse datasets, machine learning models can capture complex relationships and hidden patterns that may not be apparent through conventional diagnostic approaches.

The main objective of this project is to develop a machine learning-based system that can accurately classify thyroid disorders based on patient data and clinical features. The system aims to aid healthcare professionals in making informed decisions, enhancing diagnostic accuracy, and potentially reducing the need for unnecessary tests and procedures. To achieve this goal, we will collect a comprehensive dataset comprising anonymized patient records, encompassing relevant clinical attributes such as hormone levels, thyroid function tests, patient demographics, and medical history. The data will be carefully preprocessed to ensure its quality and suitability for analysis.

The successful development of an accurate and reliable thyroid detection system using machine learning holds significant potential for

improving healthcare outcomes. It can enable early detection, facilitate timely interventions, and assist healthcare professionals in making well-informed decisions regarding patient care. Ultimately, this project aims to contribute to the growing field of medical diagnostics by harnessing the power of machine learning in the detection and diagnosis of thyroid disorders.

Motivation

The subject of the project is decided as the need for perfect, precise diagnosis of disease. In the medical field sometimes, it is very hard to give errorless prediction. Some diseases are that type which needs early detection and proper treatment.

The project is influenced by several significant factors, such as improved diagnostic accuracy, efficiency, and time-saving, reduction in healthcare costs, and research and development.

- 1) Improved diagnostic accuracy - By improving the accuracy of thyroid disorder detection, early intervention, and treatment can be initiated, leading to better patient outcomes.
- 2) Efficiency and Time-Saving - Instead of relying solely on manual analysis by radiologists or physicians, a trained model can quickly analyze medical images and provide preliminary findings. This can save time for healthcare professionals and enable them to focus on interpreting complex cases or making critical
- 3) Reduction in healthcare costs - By minimizing the need for extensive manual analysis and improving the accuracy of detection, unnecessary follow-up tests and procedures can be avoided. This can lead to cost savings for both patients and healthcare systems.

Objectives

- To build a machine learning model capable of more precisely detecting thyroid abnormalities or diseases based on input data.
- To improve the accuracy of thyroid disease diagnosis compared to traditional methods.
- To develop a personalized approach that considers individual patient characteristics, such as age, gender, and medical history.
- To create a reliable and efficient system that can assist medical professionals.
- To enhance the accuracy of detecting thyroid disease using several ML techniques.

Data Description

We collected secondary data (Thyroid dataset & Thyroid attribute names) from the [UCI repository](#). Most of the attributes are Categorical variables. There is a total of 29 attributes such as age, sex, on thyroxine, query on thyroxine, on antithyroid medication, sick, pregnant, thyroid surgery, I131 treatment, query hypothyroid, query hyperthyroid, lithium, goiter, tumor, hypo pituitary, psych, TSH measured, TSH, T3 measured, T3, TT4 measured, TT4, T4U measured, T4U, FTI measured, FTI, TBG measured, TBG. Our dataset contains 9771 rows and 29 columns.

Attribute Name	Containing variable	Data type	Term
Age	21,36,43, etc.	integer	
Sex	Male, female	object	
On thyroxine	true, false	object	taking treatment on thyroxine
query on thyroxine	true, false	object	doubt on taking treatment on thyroxine or not
on antithyroid medication	true, false	object	taking treatment for antithyroid
pregnant	true, false	object	whether pregnant or not
thyroid surgery	true, false	object	whether surgery for thyroid is done or not
I131 treatment	true, false	object	whether radiotherapy is taken or not
query hypothyroid	true, false	object	not confirmed for hypothyroid
query hyperthyroid	true, false	object	not confirmed for hyperthyroid
lithium	true, false	object	whether lithium is present or not in the body
goiter	true, false	object	inflammation of the thyroid gland
tumor	true, false	object	thyroid cancer or not
Hypo pituitary	true, false	object	short supply of pituitary hormone
psych	true, false	object	having psychiatric symptoms of mental illness
TSH measured	1.2,0.6, etc.	integer	count of TSH
TSH	0.5,1.2, etc.	integer	Thyroid-stimulating hormone
T3 measured	1.2,0.6, etc.	integer	count of T3
T3	0.5,1.2, etc.	integer	Triiodothyronine
TT4 measured	1.2,0.6, etc.	integer	count of Total Thyroxine
TT4	0.5,1.2, etc.	integer	Total Thyroxine
T4U measured	1.2,0.6, etc.	integer	count of T4U
T4U	0.5,1.2, etc.	integer	Thyroxine uptake
FTI measured	42,56, etc.	integer	count of FTI
FTI	42,56, etc.	integer	Free Thyroxine Index
TBG measured	2.5,1.5,etc.	integer	count of TBG
TBG	2.5,1.6,etc.	integer	Thyroxine binding globulin
Target	A, B, C, etc.	object	

Methodologies

1) Logistic Regression:

What is logistic regression?

- Logistic regression is one of the most popular Machine Learning algorithms, which comes under the Supervised Learning technique. It is used for predicting the categorical dependent variable using a given set of independent variables.
- Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.
- Logistic Regression is much similar to Linear Regression except that how they are used.

Why do we use logistic regression?

Logistic regression is a widely used statistical model in machine learning (ML), particularly for binary classification problems. It is preferred for several reasons:

1. **Simplicity and interpretability:** Logistic regression is a straightforward model that is easy to implement and understand. It assumes a linear relationship between the input variables and the logarithm of the odds of the target variable, making the coefficients interpretable.
2. **Probability estimation:** Logistic regression models estimate the probability of an instance belonging to a particular class. By applying a sigmoid function to the linear combination of input features, the model outputs a value between 0 and 1, representing the probability of the positive class. This can be useful when you need to assess the confidence or uncertainty associated with predictions.
3. **Robustness to noise:** Logistic regression is less prone to overfitting compared to more complex models, such as decision trees or neural

networks. It works well even when the data is not perfectly clean or when there is a small number of informative features.

4. Feature importance: The coefficients learned by logistic regression can indicate the importance of each input feature in predicting the target variable. Positive coefficients suggest a positive relationship with the target, while negative coefficients imply a negative relationship.

5. Low computational cost: Logistic regression is computationally efficient, making it suitable for large datasets or real-time applications. The model can be trained quickly and can handle high-dimensional data efficiently.

6. Assumptions and interpretability: Logistic regression has assumptions related to linearity, independence of errors, and absence of multicollinearity among the predictors. These assumptions allow for easier interpretation and provide insights into the relationship between the input features and the target variable.

It's important to note that logistic regression is primarily used for binary classification problems, where the target variable has two classes. However, it can be extended to handle multiclass classification problems using techniques like one-vs-rest or SoftMax regression.

Code For logistic regression:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# AHere 'df' is our DataFrame and 'target' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a logistic regression model
logreg = LogisticRegression()

# Fit the model on the training data
logreg.fit(X_train, y_train)

# Make predictions on the training and testing data
y_train_pred = logreg.predict(X_train)
y_test_pred = logreg.predict(X_test)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy for LogisticRegression:", train_accuracy)
print("Testing Accuracy for LogisticRegression:", test_accuracy)
```

OutPut:

```
Training Accuracy for LogisticRegression: 0.8590112477112215
Testing Accuracy for LogisticRegression: 0.8682008368200836
```

Using logistic regression, we achieved 86 % accuracy.

2) Support Vector Machine:

What is a support vector machine?

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms used for Classification and Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called support vectors, and hence algorithm is termed a Support Vector Machine.

Why do we use SVM?

Support Vector Machines (SVMs) are widely used in machine learning for various reasons:

1. **Effective in high-dimensional spaces:** SVMs can effectively handle datasets with a large number of features, even when the number of features exceeds the number of samples. This makes SVMs suitable for tasks involving text classification, image recognition, and genomics, where the number of dimensions can be extremely high.
2. **Strong generalization:** SVMs aim to find a decision boundary that maximizes the margin between classes. This margin maximization helps SVMs to have better generalization ability, meaning they can perform well on unseen data by avoiding overfitting.
3. **Versatility with kernels:** SVMs can handle both linearly separable and nonlinearly separable data by using kernel functions. Kernels transform the data into a higher-dimensional space, where linear separation is possible. Commonly used kernel functions include linear, polynomial, Gaussian radial basis function (RBF), and sigmoid. This flexibility allows SVMs to capture complex patterns and decision boundaries.

4. Robust to noise: SVMs are robust to noisy data and outliers. Since SVMs focus on maximizing the margin, they are less affected by individual data points that are far away from the decision boundary. This property makes SVMs suitable for datasets with noise or outliers.

5. Ability to handle large datasets: SVMs can handle large datasets efficiently. The computational complexity of SVMs is primarily dependent on the number of support vectors, which are the data points that lie closest to the decision boundary. Therefore, SVMs can be memory-efficient and computationally feasible for large-scale problems.

6. Interpretability: SVMs provide interpretability by identifying the support vectors, which are the critical data points for defining the decision boundary. The support vectors help understand the influence of specific instances on the classification or regression outcome. Additionally, the learned hyperplane coefficients can indicate the importance of different features in the decision-making process.

7. Effective for both classification and regression: SVMs can be applied to both classification and regression tasks. For classification, SVMs aim to find a decision boundary that separates the classes, while for regression, SVMs aim to find a hyperplane that approximates the target values.

Overall, SVMs offer a combination of flexibility, generalization ability, and robustness, making them suitable for a wide range of machine-learning tasks.

Code for Support vector machine:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create an SVM classifier
svm = SVC()

# Fit the model on the training data
svm.fit(X_train, y_train)

# Make predictions on the training and testing data
y_train_pred = svm.predict(X_train)
y_test_pred = svm.predict(X_test)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)

OutPut:
Training Accuracy: 0.8608422704682187
Testing Accuracy: 0.8702928870292888
```

Using SVM, we achieved 87 % accuracy.

3)Decision Tree:

What is a Decision tree?

A decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome. In a Decision tree, there are two nodes, which are the Decision Node and Leaf Node. Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.

Why do we use a decision tree?

There are various algorithms in Machine learning, so choosing the best algorithm for the given dataset and problem is the main point to remember while creating a machine learning model. Below are the two reasons for using the Decision tree:

- 1) Decision Trees usually mimic human thinking ability while deciding, so it is easy to understand
- 2) The logic behind the decision tree can be easily understood because it shows a tree-like structure.

Code for the Decision Tree:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Decision Tree classifier
dt = DecisionTreeClassifier()

# Fit the model on the training data
dt.fit(X_train, y_train)

# Make predictions on the training and testing data
y_train_pred = dt.predict(X_train)
y_test_pred = dt.predict(X_test)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

OutPut:

Training Accuracy: 1.0

Testing Accuracy: 0.950836820083682

Using logistic regression we achieved 95.08 % accuracy.

4) Random Forest:

What is Random Forest?

Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset." Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and predicts the final output. The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

Why do we use Random Forest?

Random Forest is a popular machine learning algorithm that offers several advantages, which make it a valuable tool in various applications. Here are some reasons why Random Forest is used:

1. High predictive accuracy: Random Forest often provides excellent predictive accuracy across a wide range of tasks. It combines the predictions of multiple decision trees, reducing the risk of overfitting and improving generalization performance. By averaging the predictions of multiple trees, Random Forest minimizes the impact of individual noisy or biased trees.
2. Robustness to outliers and noise: Random Forest is robust to outliers and noisy data points due to its ensemble nature. Outliers and noisy data typically have a limited impact on the overall predictions because they are averaged out across the ensemble of trees.
3. Nonlinear relationships: Random Forest can capture complex nonlinear relationships between features and the target variable. Each decision tree in the ensemble can model different aspects of the data, allowing Random Forest to capture diverse patterns and interactions among features.
4. Handling high-dimensional data: Random Forest can effectively handle datasets with a large number of features. It automatically performs feature selection by considering a random subset of features at each split of a decision

tree. This property reduces the risk of overfitting and improves the algorithm's efficiency on high-dimensional datasets.

5. Interpretable feature importance: Random Forest provides a measure of feature importance based on the average impurity reduction or mean decrease in accuracy caused by each feature across all the trees. This information helps identify the most influential features in the prediction process, providing insights into the underlying data.

6. Robust against overfitting: Random Forest uses bootstrap aggregating (bagging) and random feature selection during tree construction, which helps reduce the risk of overfitting. Bagging involves training each decision tree on a random subset of the training data with replacement. Random feature selection ensures that different subsets of features are considered at each split, reducing the correlation between trees and promoting diversity in the ensemble.

7. Handling missing data: Random Forest can handle missing data without the need for imputation. During prediction, the algorithm uses the available features in a tree to make predictions for instances with missing values.

Random Forest is a versatile algorithm that can be used for classification, regression, feature selection, and outlier detection tasks. Its ability to handle complex data, provide interpretability, and deliver high predictive accuracy makes it a valuable tool in machine learning.

Code for the random forest:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Random Forest classifier
rf = RandomForestClassifier()

# Fit the model on the training data
rf.fit(X_train, y_train)

# Make predictions on the training and testing data
y_train_pred = rf.predict(X_train)
y_test_pred = rf.predict(X_test)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

OutPut:

Training Accuracy: 1.0

Testing Accuracy: 0.9644351464435147

Using random forest, we achieved 96.44 % accuracy.

5)AdaBoost:

What is AdaBoost?

AdaBoost is an iterative ensemble method. AdaBoost classifier builds a strong classifier by combining multiple poorly performing classifiers so that you will get high accurate strong classifier. The basic concept behind AdaBoost is to set the weights of classifiers and train the data sample in each iteration such that it ensures accurate predictions of unusual observations.

Why do we use AdaBoost?

AdaBoost (Adaptive Boosting) is a popular machine learning algorithm that is used for boosting, a technique that combines multiple weak learners (often decision trees) into a strong ensemble model. AdaBoost offers several advantages, which make it useful in various applications. Here are some reasons why AdaBoost is used:

1. Improved predictive accuracy: AdaBoost improves predictive accuracy by combining the predictions of multiple weak learners. The algorithm assigns higher weights to the instances that are difficult to classify correctly, focusing subsequent weak learners on those instances. This adaptive nature of AdaBoost helps to build a strong ensemble model that outperforms individual weak learners.
2. Handles complex relationships: AdaBoost can effectively handle complex relationships between features and the target variable. By combining multiple weak learners, each focusing on different aspects of the data, AdaBoost can capture intricate patterns and interactions that may be missed by individual models.
3. Robust against overfitting: AdaBoost uses a weighted voting mechanism to combine the predictions of weak learners. The algorithm assigns higher weights to misclassified instances, making subsequent weak learners focus on those instances and learn from their mistakes. This iterative process helps to reduce overfitting by continually adjusting the model's attention towards challenging instances.

4. Feature importance: AdaBoost provides a measure of feature importance based on how often or how much each feature is used in the ensemble of weak learners. This information helps identify the most relevant features and can provide insights into the underlying data.

5. Handling imbalanced datasets: AdaBoost can handle imbalanced datasets by assigning higher weights to the minority class instances during training. This approach allows the algorithm to focus on correctly classifying the minority class, addressing the challenge of imbalanced class distributions.

6. Versatility: AdaBoost can be applied to both classification and regression tasks. The algorithm can handle binary classification problems and can be extended to handle multi-class classification using techniques like one-vs-rest or multi-class AdaBoost.

7. Efficient and scalable: AdaBoost is computationally efficient and scalable, as each weak learner is trained independently. Weak learners can be trained in parallel, making AdaBoost suitable for large-scale datasets and distributed computing environments.

It's important to note that AdaBoost can be sensitive to noisy or outlier data, and it may not perform well if the weak learners are too complex or prone to overfitting. Proper tuning of hyperparameters and careful selection of weak learners are necessary for optimal performance.

Code for AdaBoost:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create an AdaBoost classifier
adaboost = AdaBoostClassifier()

# Fit the model on the training data
adaboost.fit(X_train, y_train)

# Make predictions on the training and testing data
y_train_pred = adaboost.predict(X_train)
y_test_pred = adaboost.predict(X_test)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

OutPut:

Training Accuracy: 0.9278053884383992

Testing Accuracy: 0.94979079497907957

Using Adaboost, we achieved 94.97 % accuracy.

6) XGboost:

What is XGboost?

XGBoost, which stands for Extreme Gradient Boosting, is a scalable, distributed gradient-boosted decision tree (GBDT) machine learning library. It provides a parallel tree boosting and is the leading machine-learning library for regression, classification, and ranking problems. It's vital to an understanding of XGBoost to first grasp the machine learning concepts and algorithms that XGBoost builds upon supervised machine learning, decision trees, ensemble learning, and gradient boosting.

Why do we use XGboost?

XGBoost (Extreme Gradient Boosting) is a highly optimized and powerful gradient-boosting algorithm that has gained significant popularity in machine learning. It offers several advantages, which make it a preferred choice in various applications. Here are some reasons why XGBoost is used:

1. Enhanced performance: XGBoost is designed to deliver superior performance compared to other gradient boosting implementations. It utilizes a variety of optimization techniques and algorithmic enhancements, such as parallel processing, tree pruning, and regularization, to achieve faster training and prediction times.
2. High predictive accuracy: XGBoost consistently demonstrates state-of-the-art performance on a wide range of machine learning tasks. By iteratively adding weak learners (decision trees) and adjusting the weights of misclassified instances, XGBoost builds a strong ensemble model that can capture complex patterns and relationships in the data, leading to improved predictive accuracy.
3. Regularization techniques: XGBoost incorporates regularization techniques to prevent overfitting and enhance generalization. It includes both L1 (Lasso) and L2 (Ridge) regularization terms in the objective function, allowing automatic feature selection and reducing the impact of irrelevant features.

4. Handling missing values: XGBoost has built-in capabilities to handle missing data. During tree construction, XGBoost can learn to handle missing values by assigning them to the most appropriate direction in the tree based on the training data statistics.

5. Feature importance: XGBoost provides a measure of feature importance based on the number of times a feature is used in the ensemble of trees and the average gain (improvement in loss function) achieved by splits involving that feature. This information helps identify the most influential features in the prediction process and can guide feature selection and engineering efforts.

6. Flexibility in objective functions: XGBoost allows users to define custom objective functions, making it suitable for a wide range of machine learning tasks beyond classification and regression. This flexibility enables XGBoost to be applied to ranking, recommendation, and other advanced modeling problems.

7. Handling imbalanced datasets: XGBoost includes parameters that can help address class imbalance issues. By adjusting the class weights or utilizing subsampling strategies, XGBoost can effectively handle imbalanced datasets and prevent the dominant class from overwhelming the model.

Due to its impressive performance, scalability, and versatility, XGBoost has become a go-to algorithm for various machine learning competitions, industry applications, and research projects.

Code for XGboost:

```
import pandas as pd
from sklearn.model_selection import train_test_split
import xgboost as xgb
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create an XGBoost classifier
xgb_model = xgb.XGBClassifier()

# Fit the model on the training data
xgb_model.fit(X_train, y_train)

# Make predictions on the training and testing data
y_train_pred = xgb_model.predict(X_train)
y_test_pred = xgb_model.predict(X_test)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

OutPut:

Training Accuracy: 0.983782369866597

Testing Accuracy: 0.9602510460251046

Using XGBoost, we achieved 96.02 % accuracy.

7) PCA:

What is PCA?

Principal Component Analysis is an unsupervised learning algorithm that is used for dimensionality reduction in machine learning. It is a statistical process that converts the observations of correlated features into a set of linearly uncorrelated features with the help of orthogonal transformation. These new transformed features are called the Principal Components. It is one of the popular tools that is used for exploratory data analysis and predictive modeling. It is a technique to draw strong patterns from the given dataset by reducing the variances.

Why do we use PCA?

Principal Component Analysis (PCA) is a widely used dimensionality reduction technique in machine learning and data analysis. It offers several benefits and is used for various purposes. Here are some reasons why PCA is used:

1. Dimensionality reduction: PCA helps in reducing the number of features or variables in a dataset while retaining the most important information. It achieves this by transforming the original set of correlated variables into a new set of uncorrelated variables called principal components. These principal components are ranked in terms of the amount of variance they explain in the data, allowing for dimensionality reduction while preserving the most significant patterns and trends.
2. Feature extraction: PCA can be used as a feature extraction technique to generate a smaller set of uncorrelated features that capture the essential information from the original dataset. By identifying the principal components that contribute the most to the overall variance, PCA helps identify the key features that explain the data's variability. These extracted features can be used as input for subsequent machine learning algorithms, simplifying the modeling process and potentially improving performance.
3. Noise reduction: PCA can help reduce the impact of noise and irrelevant information in the data. By focusing on the principal components that capture the most significant variation, PCA can filter out noise and retain the

underlying patterns and structures. This noise reduction can lead to better generalization and improved performance in subsequent modeling tasks.

4. Visualization: PCA facilitates the visualization of high-dimensional data in lower-dimensional spaces. By reducing the dimensionality of the data to two or three principal components, PCA enables the plotting of data points in a scatter plot or 3D plot, aiding in understanding the data's structure and identifying clusters or patterns.

5. Multicollinearity detection: PCA can be used to identify and handle multicollinearity, which occurs when predictor variables in a dataset are highly correlated. By transforming the original variables into orthogonal principal components, PCA eliminates the issue of multicollinearity, making subsequent modeling more stable and interpretable.

6. Interpretability: PCA provides interpretability in terms of the importance of each principal component. The variance explained by each component serves as a measure of its significance. This information helps identify the most influential features or variables in the dataset, providing insights into the underlying data structure.

7. Data preprocessing: PCA can be used as a data preprocessing step to reduce the dimensionality and complexity of large datasets. By selecting a subset of the most informative principal components, computational efficiency can be improved without significant loss of information.

8. Reducing memory storage: PCA can help reduce memory storage requirements by reducing the number of variables in the dataset. This is particularly useful when dealing with large-scale datasets or limited computing resources.

It's important to note that while PCA offers several advantages, it may not be suitable for all situations. For example, in some cases, preserving the interpretability of the original variables or maintaining the exact meaning of each feature may be crucial, making other dimensionality reduction methods more appropriate. Additionally, PCA assumes a linear relationship between variables, which may limit its effectiveness in capturing non-linear patterns.

Code for Logistic Regression with PCA:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Here 'df' is our DataFrame and 'target' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Perform PCA for dimensionality reduction
pca = PCA(n_components=8) # Set the number of principal components to retain
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Create a logistic regression model
logreg = LogisticRegression()

# Fit the model on the training data
logreg.fit(X_train_pca, y_train)

# Make predictions on the training and testing data
y_train_pred = logreg.predict(X_train_pca)
y_test_pred = logreg.predict(X_test_pca)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

OutPut:

Training Accuracy: 0.775568924928067

Testing Accuracy: 0.7761506276150628

Using Logistic regression with PCA, we achieved 77.61 % accuracy.

Code for SVC with PCA:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Perform PCA for dimensionality reduction
pca = PCA(n_components=9) # Set the number of principal components to retain
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Create an SVM classifier
svm = SVC()

# Fit the model on the training data
svm.fit(X_train_pca, y_train)

# Make predictions on the training and testing data
y_train_pred = svm.predict(X_train_pca)
y_test_pred = svm.predict(X_test_pca)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

OutPut:

Training Accuracy: 0.8608422704682187
Testing Accuracy: 0.8702928870292888

Using SVC with PCA, we achieved 87.02 % accuracy.

Code for Decision Tree with PCA:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Perform PCA for dimensionality reduction
pca = PCA(n_components=14) # Set the number of principal components to retain
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Create a Decision Tree classifier
dt = DecisionTreeClassifier()

# Fit the model on the training data
dt.fit(X_train_pca, y_train)

# Make predictions on the training and testing data
y_train_pred = dt.predict(X_train_pca)
y_test_pred = dt.predict(X_test_pca)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

OutPut:

Training Accuracy: 1.0

Testing Accuracy: 0.944560669456067

Using DT with PCA, we achieved 94.45 % accuracy.

Code for Random Forest with PCA:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Perform PCA for dimensionality reduction
pca = PCA(n_components=9) # Set the number of principal components to retain
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Create a Random Forest classifier
rf = RandomForestClassifier()

# Fit the model on the training data
rf.fit(X_train_pca, y_train)

# Make predictions on the training and testing data
y_train_pred = rf.predict(X_train_pca)
y_test_pred = rf.predict(X_test_pca)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)

OutPut:
Training Accuracy: 1.0
Testing Accuracy: 0.9633891213389121
```

Using Random forest with PCA, we achieved 96.33 % accuracy.

Code for xgboost with PCA:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
import xgboost as xgb
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Perform PCA for dimensionality reduction
pca = PCA(n_components=9) # Set the number of principal components to retain
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Create an XGBoost classifier
xgboost = xgb.XGBClassifier()

# Fit the model on the training data
xgboost.fit(X_train_pca, y_train)

# Make predictions on the training and testing data
y_train_pred = xgboost.predict(X_train_pca)
y_test_pred = xgboost.predict(X_test_pca)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)

OutPut:
Training Accuracy: 0.9793356003138897
Testing Accuracy: 0.9560669456066946
```

Using XGboost with PCA, we achieved 95.60 % accuracy.

Code for AdaBoost with PCA:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'target_column' is the column to predict

# Separate the features (X) and the target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# One-hot encode categorical variables if necessary
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Perform PCA for dimensionality reduction
pca = PCA(n_components=9) # Set the number of principal components to retain
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Create an AdaBoost classifier
adaboost = AdaBoostClassifier()

# Fit the model on the training data
adaboost.fit(X_train_pca, y_train)

# Make predictions on the training and testing data
y_train_pred = adaboost.predict(X_train_pca)
y_test_pred = adaboost.predict(X_test_pca)

# Calculate the accuracies
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

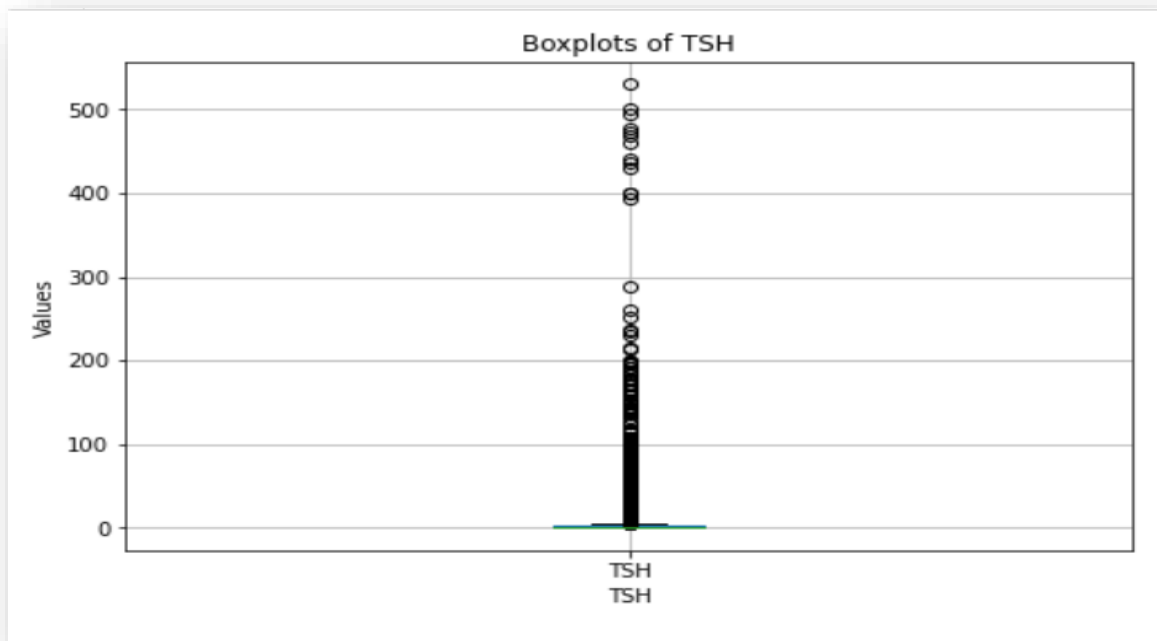
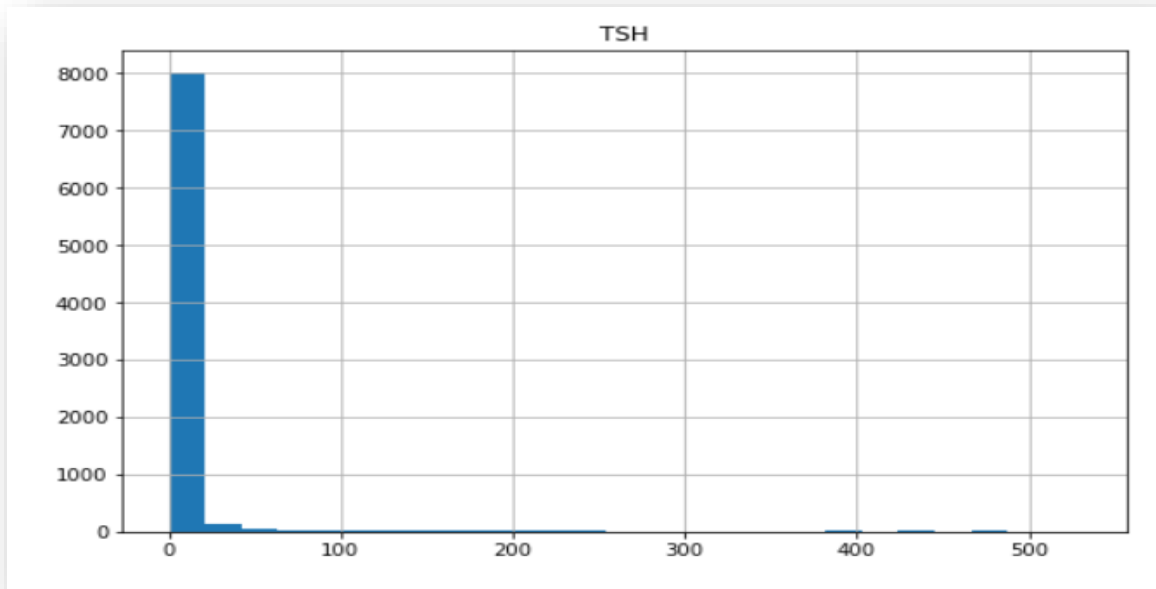
# Print the accuracies
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)

OutPut:
Training Accuracy: 0.9217891708082657
Testing Accuracy: 0.9456066945606695
```

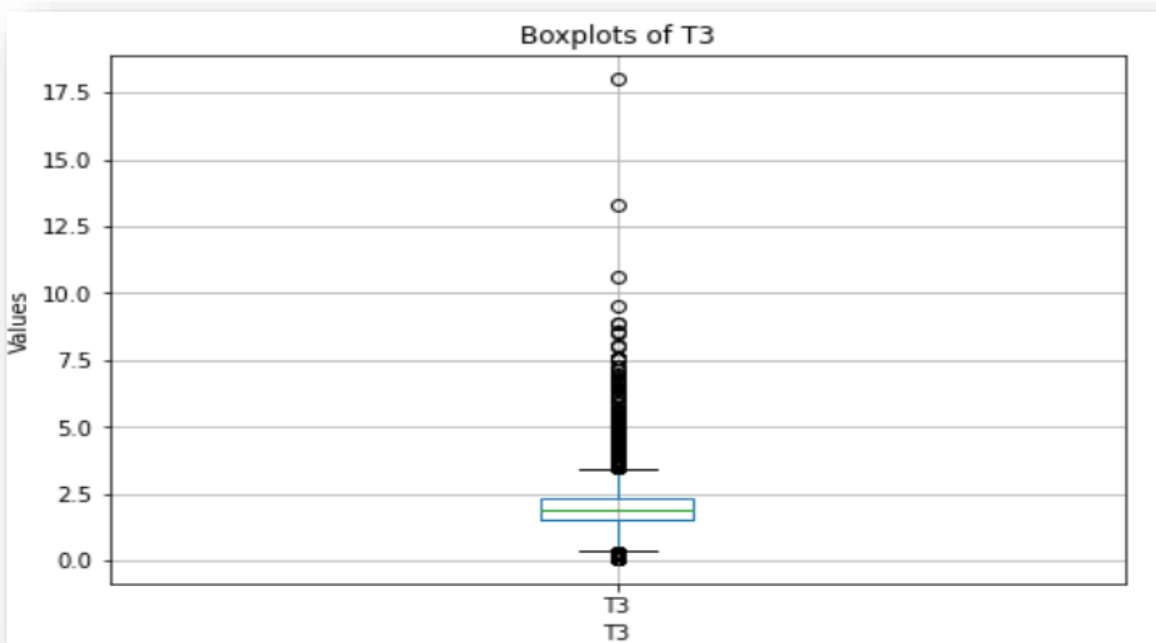
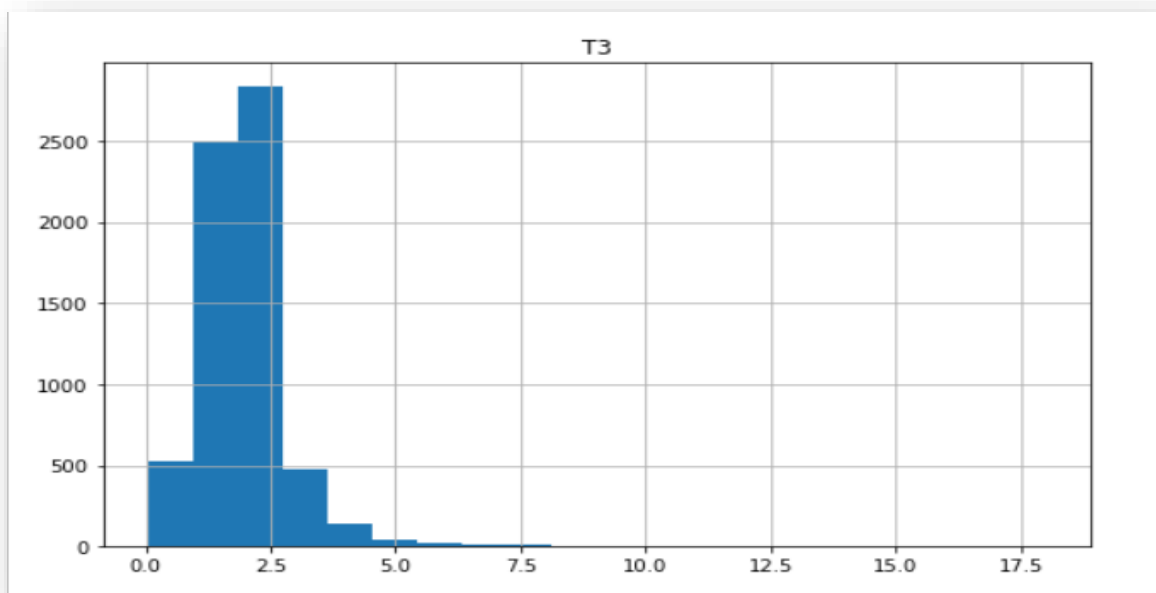
Using Adaboost with PCA, we achieved 94.56 % accuracy.

Graphical analysis:

Before cleaning:

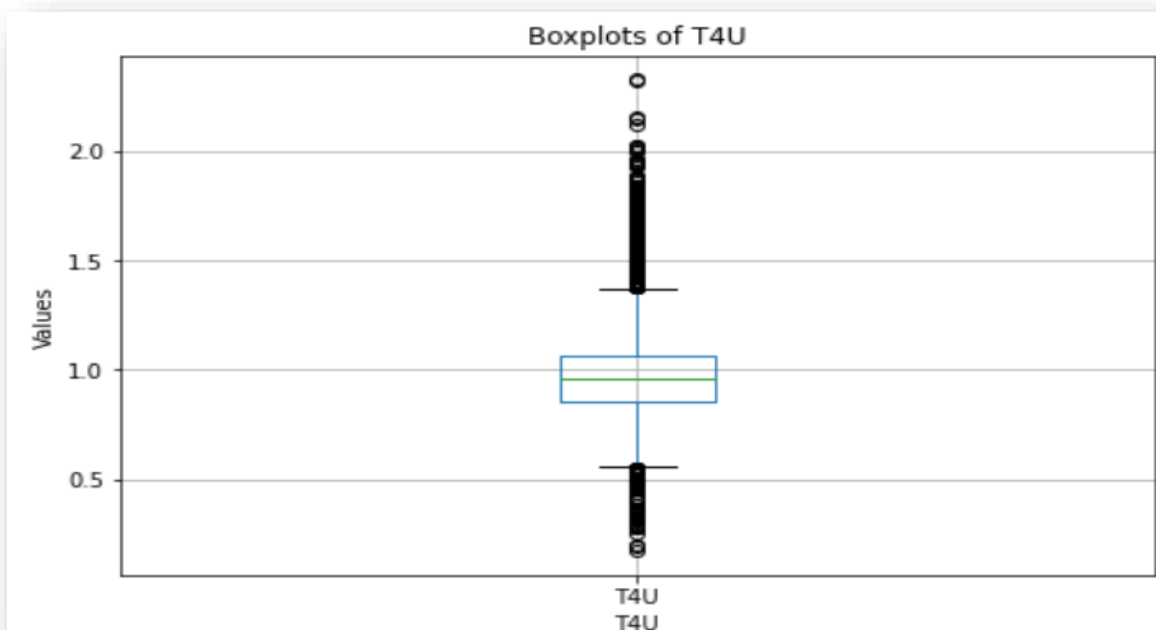
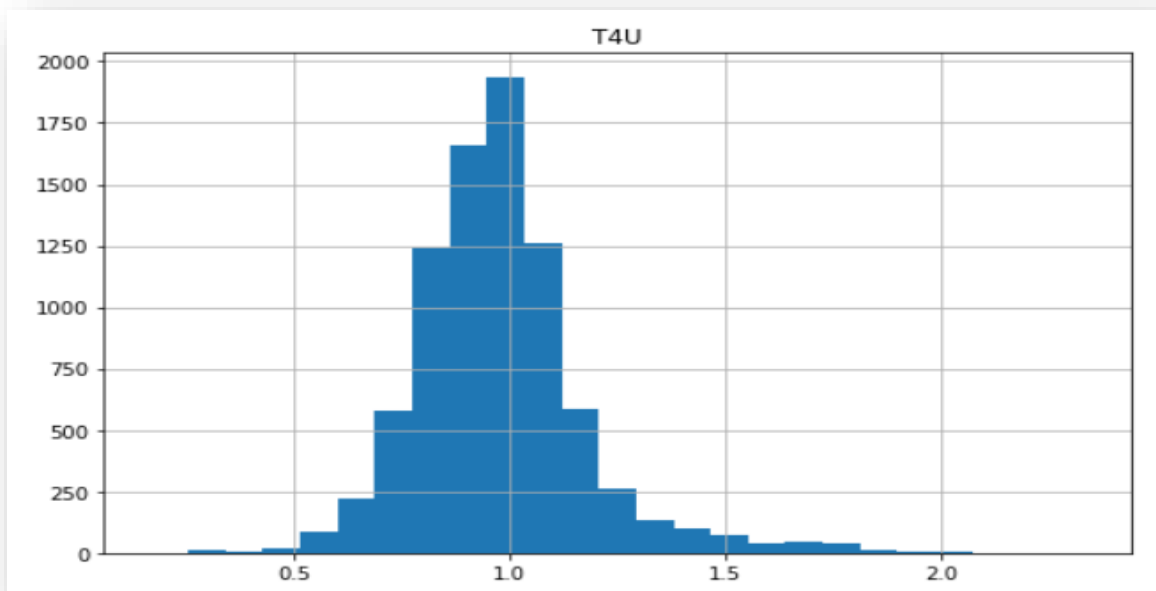


Here we can see there are too many outliers in TSH



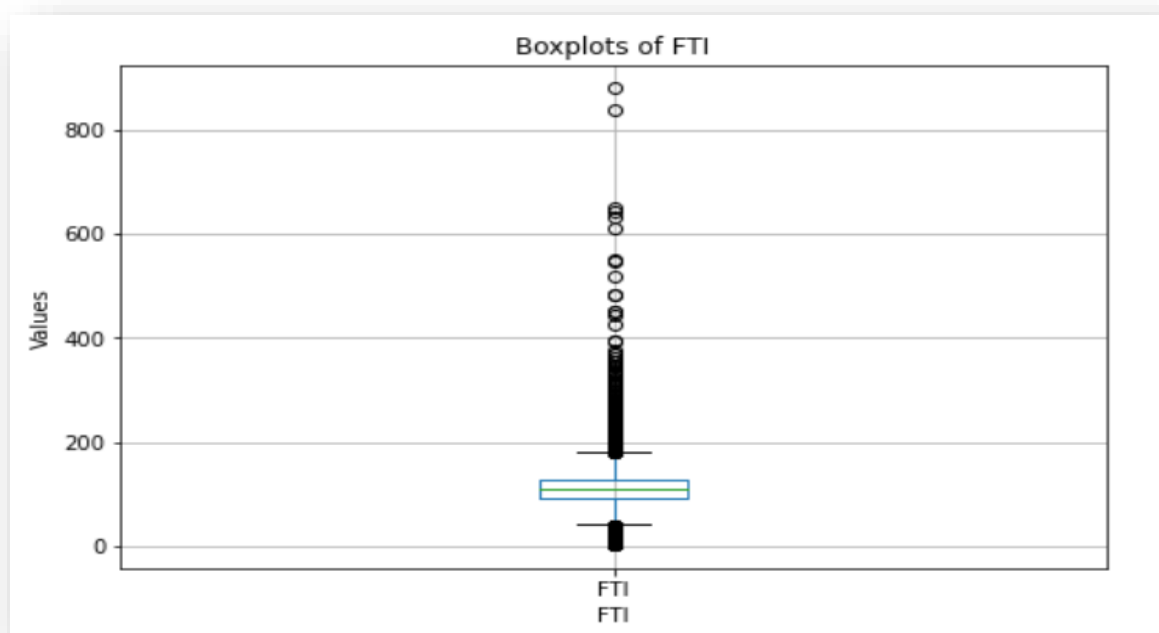
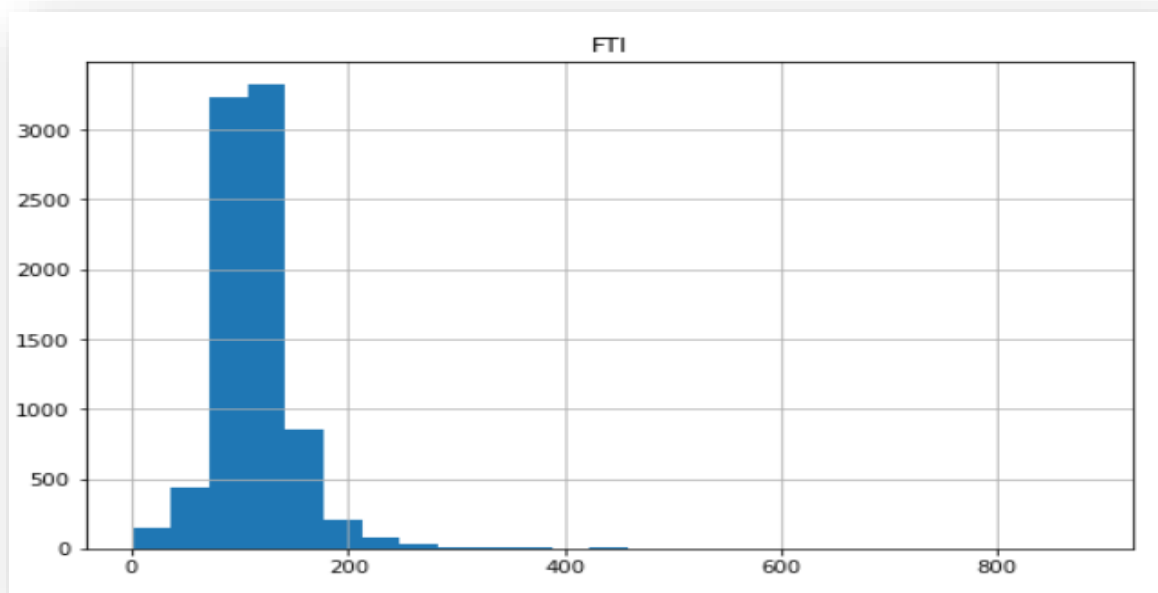
T3 or Triiodothyronine

Here we can see there are too many outliers in T3 or Triiodothyronine



T4U or Thyroxine Uptake

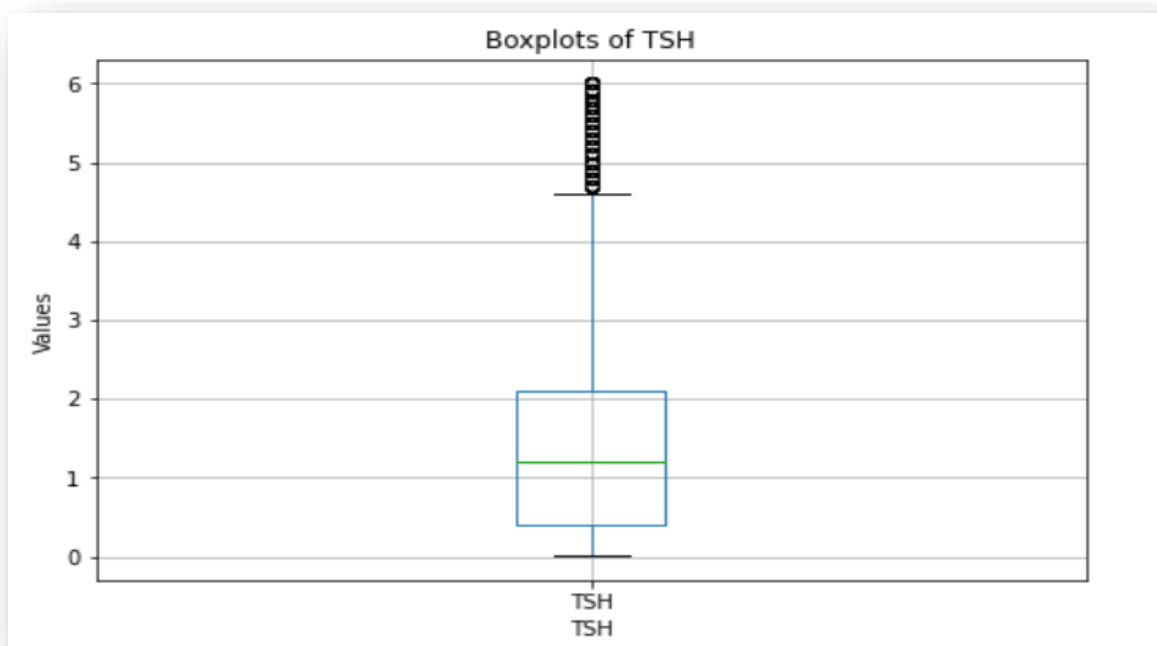
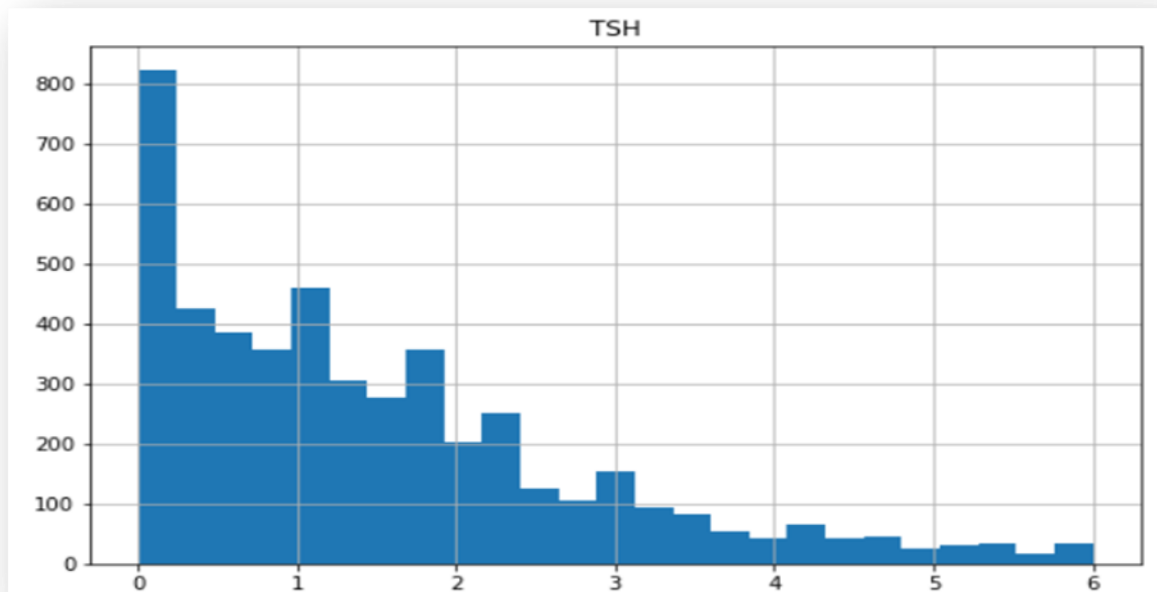
Here we can see there are too many outliers in T4U or Thyroxine Uptake



Free Thyroxine Index

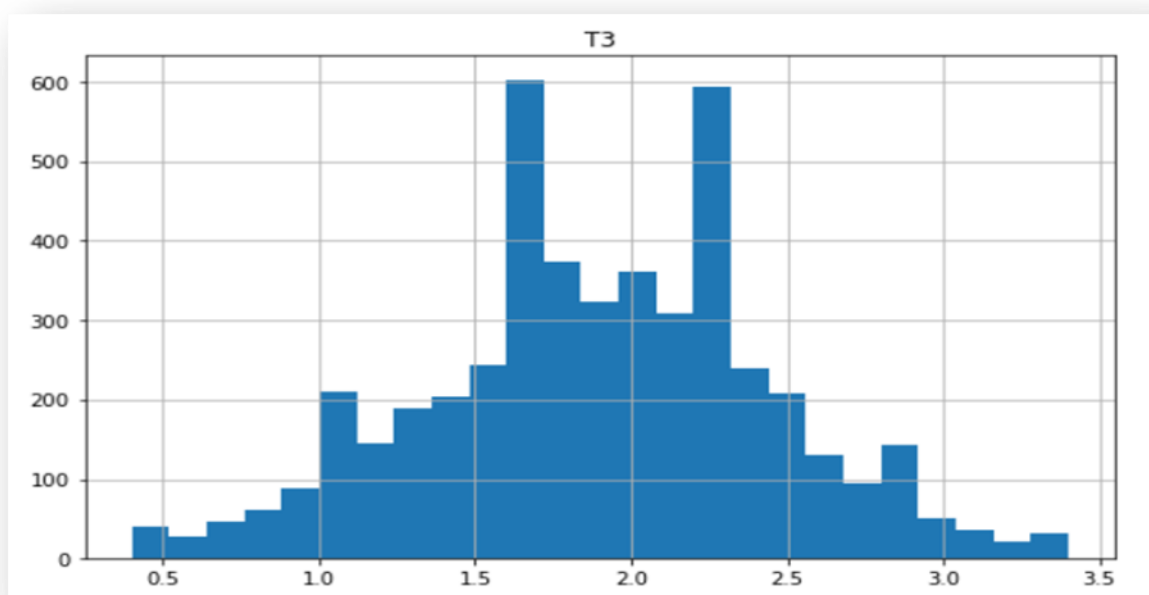
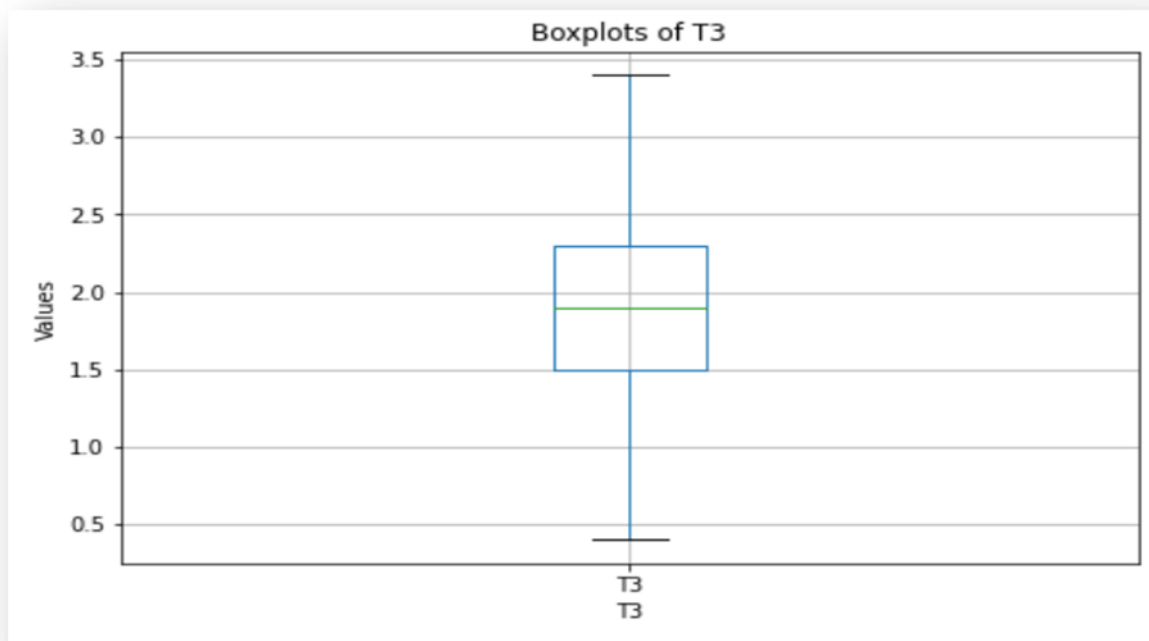
Here we can see there are too many outliers in Free Thyroxine Index

After cleaning:



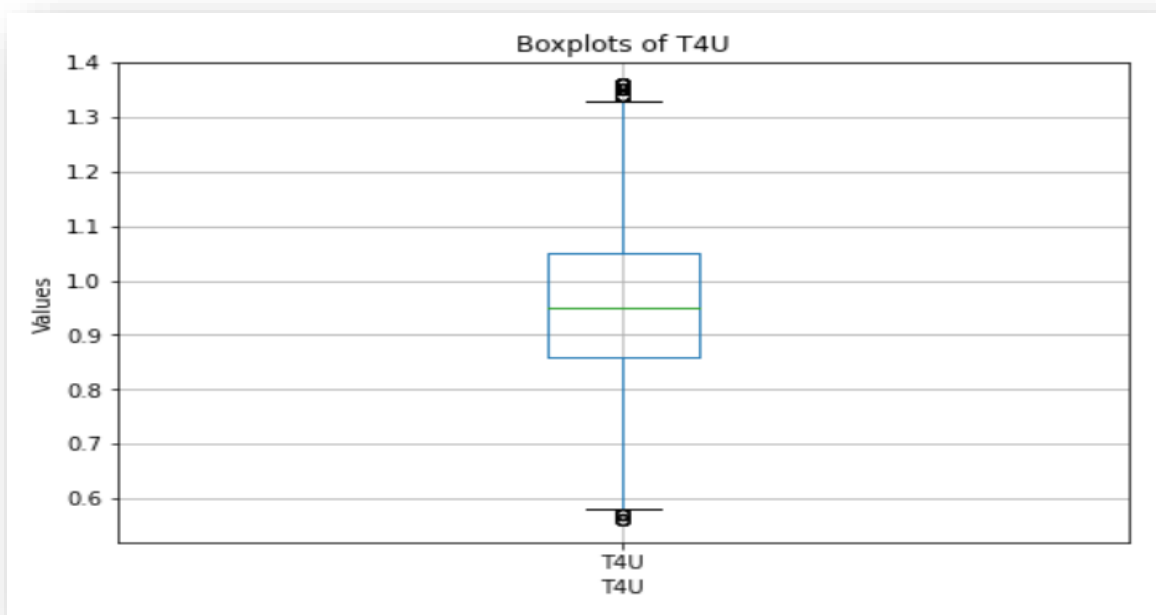
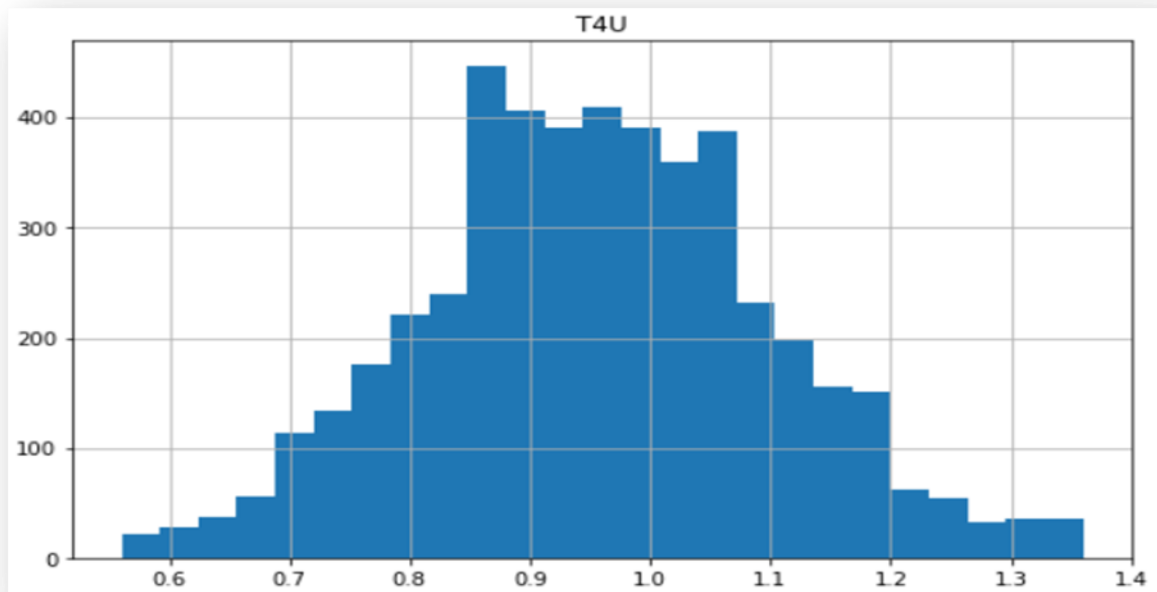
Thyroid Stimulating Hormone

After cleaning we can see there are few outliers in TSH



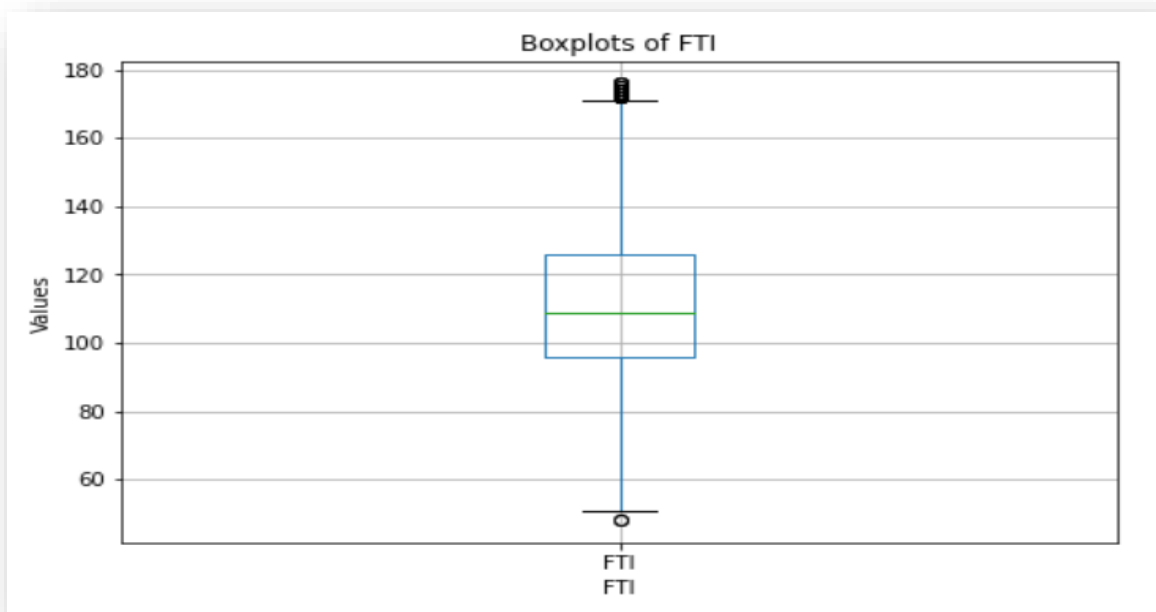
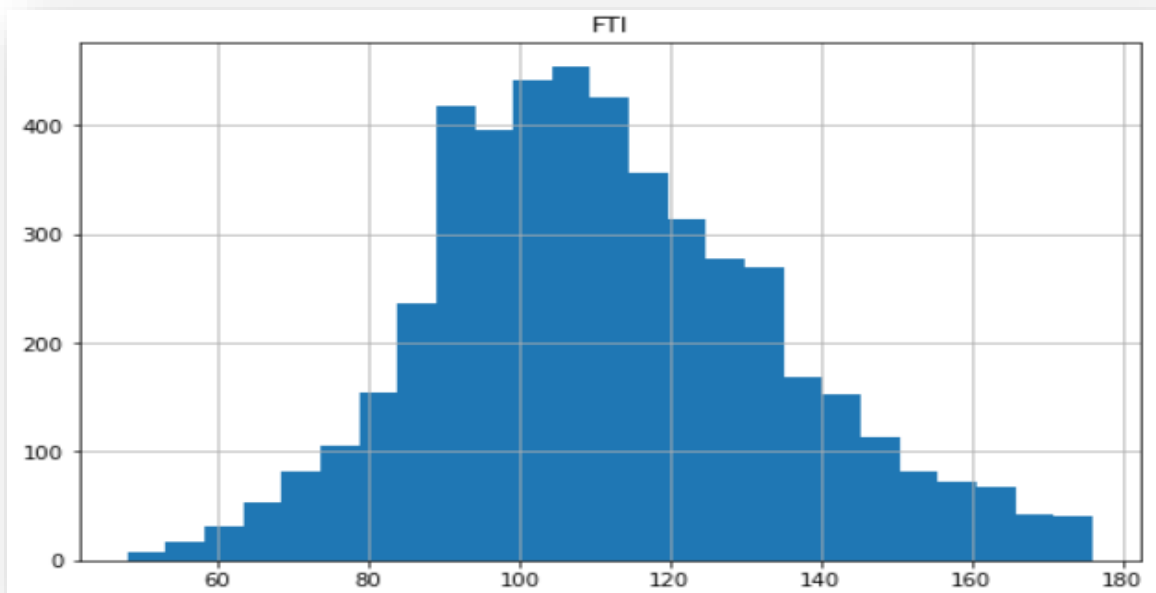
T3 or Triiodothyronine

After cleaning we can see there are No outliers T3 or Triiodothyronine



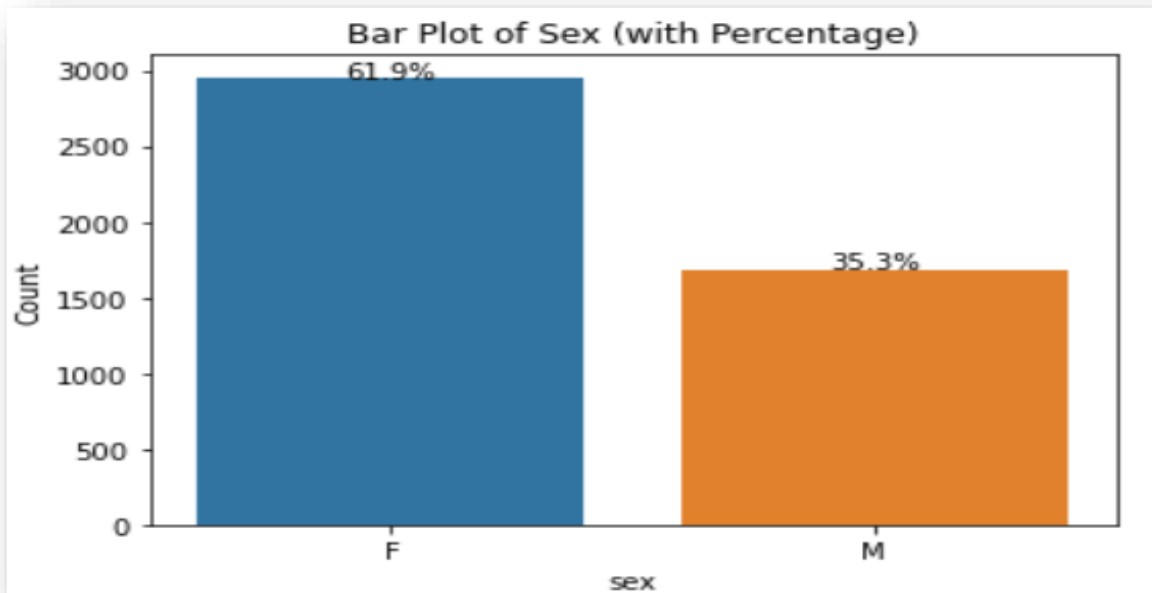
T4U or Thyroxine Uptake

After cleaning we can see there are few outliers in T4U or Thyroxine Uptake

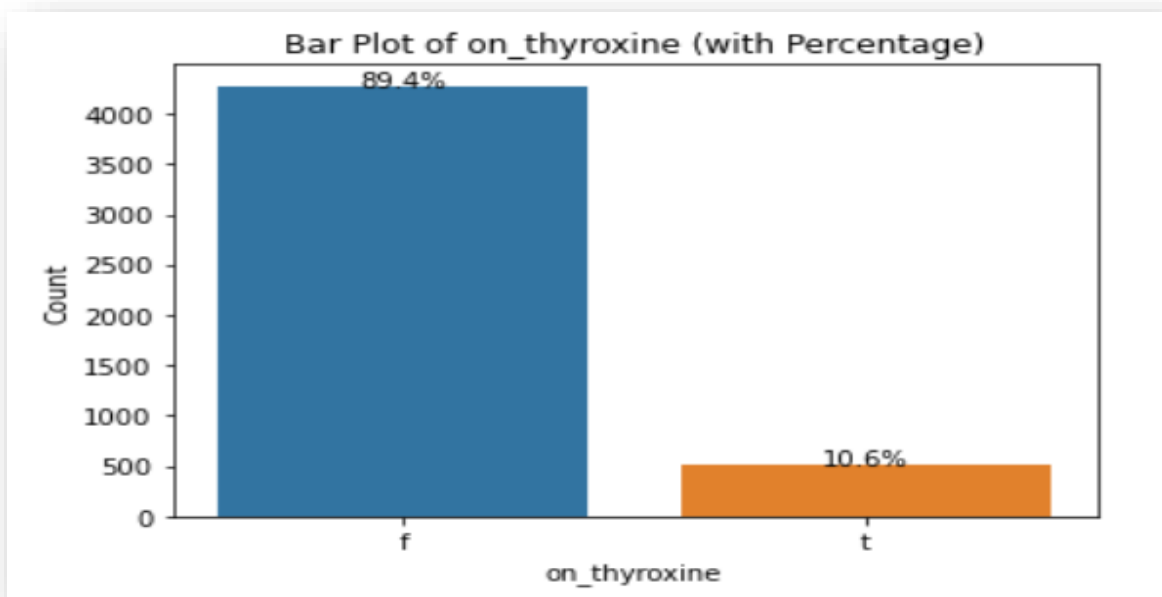


Free Thyroxine Index

After cleaning we can see there are few outliers in Free Thyroxine Index



In Bar plot of Sex We can see in the given dataset we have 35.3% Males and 61.9% Females.



From this Bar plot on Thyroxine we can say approximately 10 out of 1 people have Thyroxine.


Conclusion:

- 1) Machine learning models, specifically Random Forest proved to be effective in detecting thyroid disease. The model achieved an accuracy of 96.44%, indicating its potential for accurate identification.
- 2) The Random Forest model, trained on the given dataset, demonstrated a strong ability to discriminate between thyroid disease and non-thyroid disease cases. This highlights the potential of machine learning in assisting healthcare professionals in accurate disease diagnosis.
- 3) The selected features in the dataset played a crucial role in the accuracy of the model.
- 4) The results obtained from this project suggest that machine learning can be a valuable tool in assisting with the detection of thyroid disease. However, it should not replace clinical diagnosis or medical expertise. The model can be utilized as a supportive tool for healthcare professionals in making accurate diagnoses, leading to improved patient care.


ML-Pipeline:

Age:


Sex:


On Thyroxine:


Query on Thyroxine:


On Antithyroid Medication:


Sick:


Pregnant:


Thyroid Surgery:

I131 Treatment:

Query Hypothyroid:

Query Hyperthyroid:

False



Lithium:

False



Goitre:

False



Tumor:

False



Hypopituitary:

False



Psych:

False



TSH:

T3:

TT4:

T4U:

FTI:



Here Green Button Indicates Submit Option, after putting your all required information you will get an answer you have thyroid or not.

Future Scope:

Future work may involve incorporating additional machine learning techniques, exploring different algorithms, and evaluating the model's performance on larger and more diverse datasets.

Continued research and development in the field of machine learning-based thyroid disease detection could lead to more accurate and efficient diagnostic tools in the future.

Additionally, collaborations with domain experts and healthcare professionals can provide valuable insights for refining the model and making it more applicable in real-world clinical settings.

THANK YOU!



To Access all Research papers, Research paper Presentation PPT, Final PPT, and all Project materials scan this QR code.