Stroke Detection

A Project in the Field of AI

Ain Shams University

May 2024

Team’s Biographical Sketch

Team leader	19p6937	Osama Ali Mohamed	[email]
Team member	19p9223	Joseph robert makram	[email]
Team member	18P2797	Omama Mohammed Alnajjar	[email]
Team member	20p6300	Abdelrahman mahmoud fouad	[email]

Table of Contents

Table of Figures

Abstract

Strokes are a major health threat, but what if we could identify people at risk beforehand? This project explores using data mining to analyze patient information like age, health conditions, and habits. By cleaning the data, building a model, and visualizing the results, we aim to discover patterns that might predict stroke risk. This could be a valuable tool for doctors and individuals to take preventative action and potentially save lives.

1. Introduction

An Approach to Stroke Prediction is mitigating risk Through Early Identification Stroke using data mining techniques and methods, the second leading cause of death globally according to the World Health Organization (WHO), poses a significant public health challenge. This project explores the potential of data mining to predict stroke risk, enabling early intervention and improved patient outcomes.

The project leverages a dataset containing patient information, including demographics, health conditions, and lifestyle factors. Through data mining techniques, we aim to uncover hidden patterns and relationships within this data that contribute to stroke risk.

2. Understanding Dataset

In this section we will study the dataset and try to get as much information as possible to better understand what variables we are working on, and their types and preprocessing methods needed for them.

2.1 Variables Overview and their data types:

- **id (string):** This variable uniquely identifies each patient within the dataset. It won't be used in the training process.
- **gender (string):** This variable indicates the patient's gender. It's a categorical variable stored as text ("Male", "Female").
- **age (float\int):** This variable represents the patient's age in years. In csv it has floating point but in section 4. Dataset Preprocessing we will convert it to int.
- **hypertension (int):** This variable indicates whether the patient has hypertension (high blood pressure) , 0 for no hypertension, 1 for hypertension.
- **heart_disease (int):** This variable indicates whether the patient has a pre-existing heart disease. Like hypertension, 0 for no heart disease, 1 for heart disease.
- **ever_married (string):** This variable indicates the patient's marital status. It's a categorical variable stored as text ("Yes", "No").
- **work_type (string):** This variable describes the patient's type of employment. It's a categorical variable stored as text ("Private", "Self-employed"," Govt_job"," children","Never_worked"). self-employed generally works for themselves as a business owner, freelancer, Children is also given for kids who haven't reached age of work.

- **Residence_type (string):** This variable indicates the patient's place of residence (e.g., urban or rural area). It's a categorical variable stored as text (e.g., "Urban", "Rural").
- **avg_glucose_level (float):** This variable represents the patient's average blood glucose level.
- **bmi (string):** This variable denotes the patient's Body Mass Index (BMI), Where Body Mass Index (BMI) is a person's weight in kilograms divided by the square of height in meters.
- **smoking_status (string):** This categorical variable describes the patient's smoking habits (e.g., never smoked, currently smokes, formerly smoked).
- **stroke (int):** This variable is the target variable, indicating whether the patient experienced a stroke (1) or not (0).
-

2.2 Preprocessing Needed on Variables:

This will be discussed in detail in Section ***4. Dataset Preprocessing*****,** For Now we need to put outlines for this Phase, From the abstract look we took at the data we see that:

- **age** attribute needs to be converted to int rather than being float, and `String` in the csv.
- **Hypertension** needs to be treated as **int** to be easier for model to train on.
- **Heart_disease & stroke** are the exact same case for **Hypertension.**
- All **categorical** values are needed to be label encoded:
    - **Gender** ○ **Ever_married.**
    - **Work_type.**
    - **Residence_type**
    - **Smoking_status**
- We will also need to drop the **id** column as it has no benefit in the training process.

3. Studying Variable Relations

In this section we will study the dataset and try to get as much information as possible to better understand what variables we are working on, and their types and preprocessing methods needed for them.

3.1 getting & Visualizing Correlation Matrix of Variables:

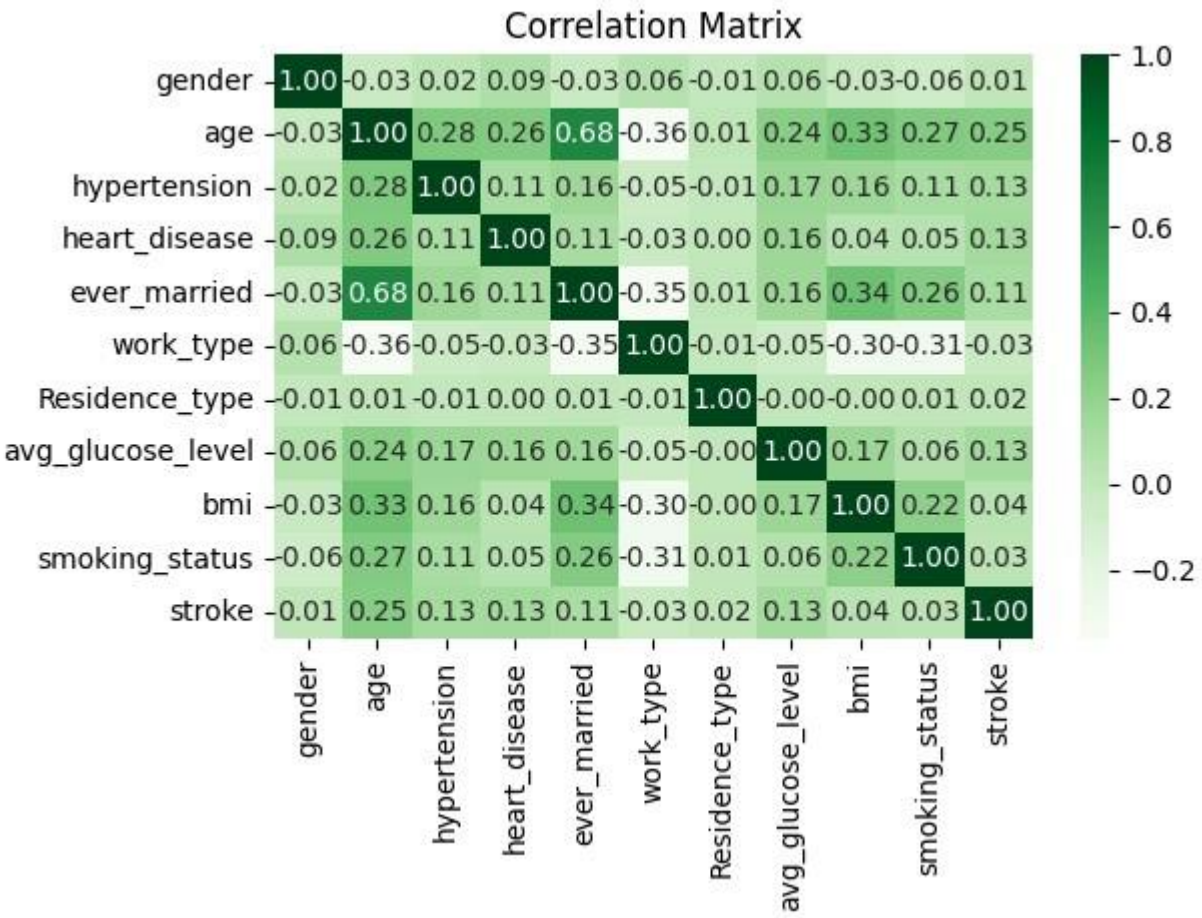


Figure 1: Correlation Matrix

Note #1: It's clear that there are few **Strong** relations between the following that are okay with common sense:

- **Age, ever_married:** This Can be easily explained as younger patients are very unlikely to be married

Note #2: We need to investigate more in the relations between the categorical variables and each other as there might be a relation between two specific categories in different variables Which we will discuss in 3.2 Using Pair Plot to Observe Relations of Important Variables

3.2 Using Pair Plot to Observe Relations of Important Variables:

I will do some of them manually and some others by generating matrices of graphs.

- There of course might be a relation between marriage and strokes, as it opposes more responsibilities on the patients which makes them more viable to strokes.

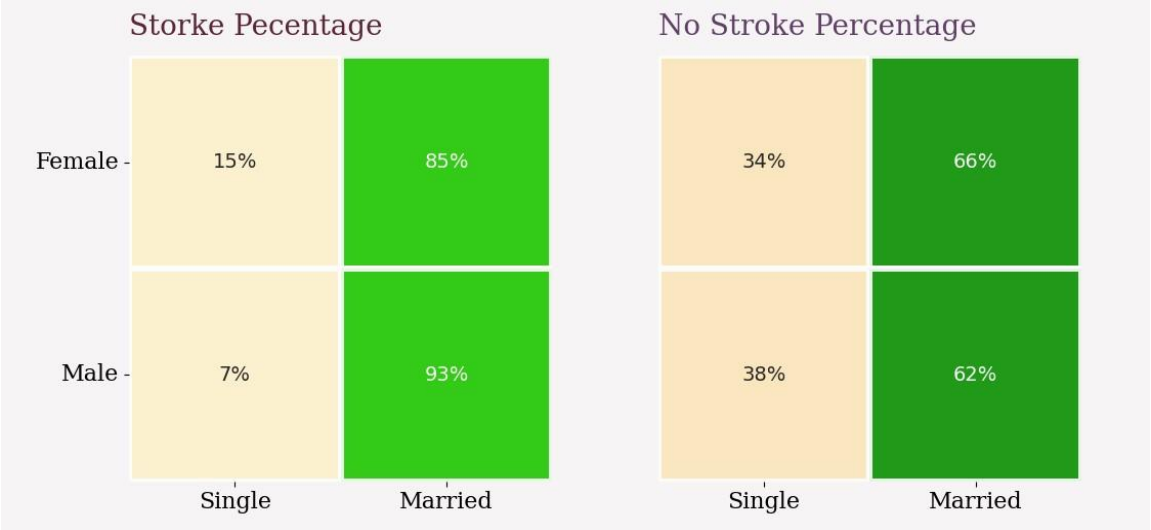


Figure 2: marriage and strokes relationship

- Also, there might be a relation between work and being viable to strokes due to work responsibilities and Exhaustion, and it's clear that business owners and freelancers are less likely to have strokes than an employee who works for nonGovt companies.

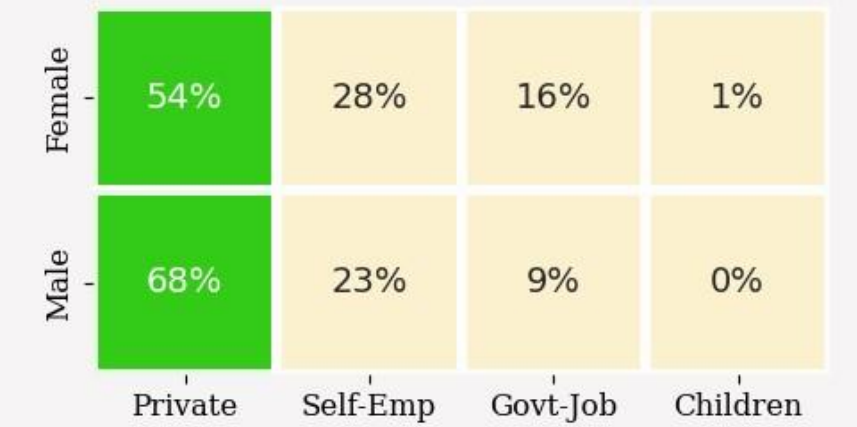


Figure 3: work & stroke relationship

Let's explore if additional relationships may emerge among other variables that WHO deems crucial to strokes:

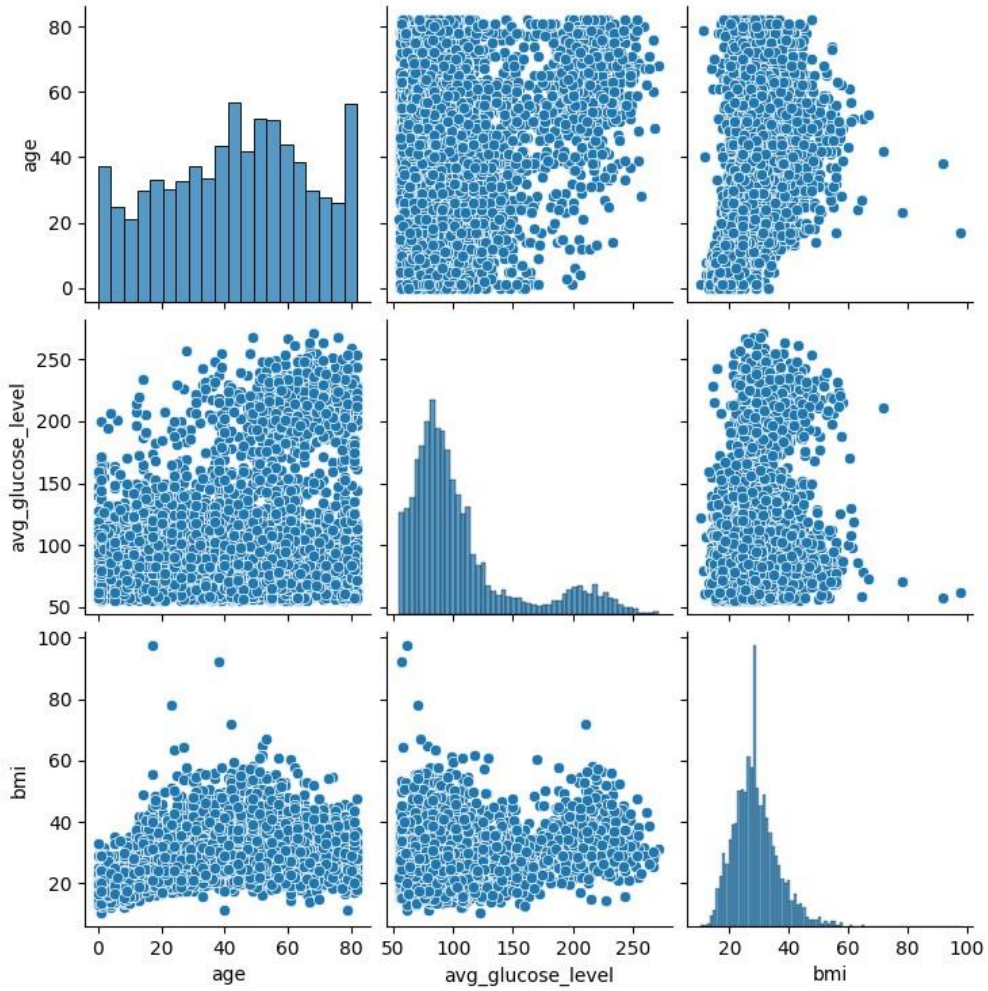


Figure 4: relationships between age, avg_glucose_level, & bmi

Observations:

- The data distribution for age shows prominent values centered around **10, 60, and 80**. Average glucose level exhibits two peaks of varying heights, approximately at values of 100 and 200.
- Regarding age, instances of stroke occurrence are noticeable within the range of **35 to 80**. However, this range does not dominate due to the dataset's inherent imbalance.
- For average glucose level, two distinct groups emerge: **60 to 100** and **180 to 220**. Patients within the first group exhibit a higher susceptibility to strokes compared to those in the second group.
- BMI values ranging from 15 to 40 have shown a higher incidence of stroke cases.
- There is a positive relation between **Age & BMI** which can be further supported from correlation matrix (r = 0.33) , which can be concluded by common sense also.
- There is a weak positive relation between **bmi** and average glucose level.

- The body mass index (BMI) demonstrates a distribution close to normal, albeit with fewer occurrences towards the higher end of the scale.

4. Dataset Preprocessing

In this section we will study the dataset and try to get as much information as possible to better understand what variables we are working on, and their types and preprocessing methods needed for them.

4.1 Filling Missing Values

```
              age  hypertension  ...          bmi       stroke
count  5110.000000   5110.000000  ...  4909.000000  5110.000000
mean     43.226614      0.097456  ...    28.893237     0.048728
std      22.612647      0.296607  ...     7.854067     0.215320
min       0.080000      0.000000  ...    10.300000     0.000000
25%      25.000000      0.000000  ...    23.500000     0.000000
50%      45.000000      0.000000  ...    28.100000     0.000000
75%      61.000000      0.000000  ...    33.100000     0.000000
max      82.000000      1.000000  ...    97.600000     1.000000

[8 rows x 6 columns]
```

Figure 5: filling missing values

BMI variable is the only variable that has missing values, so let's fix this by replacing the missing values with the mean of the BMI column.
Also applying this to categorical columns just in case we missed something, by replacing Nan values with the mode of the column.

```
################################################################ #
Fill Missing Values
################################################################
 for columnName in
dataset.select_dtypes(exclude='object').columns:
    column = dataset[columnName]
column.fillna(round(column.mean()), inplace=True)
 for columnName in
dataset.select_dtypes(include='object').columns:
    column                  =                 dataset[columnName]
column.fillna(column.mode()[0], inplace=True)
```

These two loops fill missing values in numerical and categorical columns respectively. They replace NaNs in numerical columns with the rounded mean of each column, while NaNs in categorical columns are replaced with the mode (most frequent value) of each column.

4.2 Datatype Correction:

```
#########################################################################
# Cast variables to correct type
#########################################################################
dataset['age'] = dataset['age'].astype(int) dataset['hypertension']
= dataset['hypertension'].astype(int) dataset['heart_disease'] =
dataset['heart_disease'].astype(int) dataset['stroke'] =
dataset['stroke'].astype(int)
```

Values other than categorical values and int are of type **<class 'numpy. float64'>**

- **'Heart_disease', 'stroke' , 'hypertension' :** Makes sense to be stored as int as they have only two values '0' & '1'
- **'Age'** was already float, but since it won't matter much for the model if the age is 1.02 and if it's 1.5 so we floored the value and naked it's type as int

4.3 Categorical Data Encoding:

```
#########################################################################
# Encode Categorical Variables
#########################################################################
labelEncoder = LabelEncoder() for column in
dataset.select_dtypes(include='object').columns:
labelEncoder.fit(dataset[column])
    dataset[column] = labelEncoder.transform(dataset[column])
```

Here, a LabelEncoder object is instantiated. LabelEncoder is a utility class in scikit-learn used to encode categorical labels as integer numbers.
This loop iterates over columns in the dataset that have data types of 'objects', which typically indicates categorical variables.

```
for column in dataset.select_dtypes(include='object').columns:
labelEncoder.fit(dataset[column])     dataset[column] =
labelEncoder.transform(dataset[column])
```

For each categorical column, the **fit()** method of the LabelEncoder is called to fit the encoder to the unique values in that column. This essentially assigns a unique integer to each unique category. Finally, the **transform()** method of the LabelEncoder is applied to transform the categorical values in the column to their corresponding integer representations. These integer representations replace the categorical values in the dataset.

4.4 Detecting and Replacing Outliers:

```
######################################################################
# Replace Outliers with Mean
###################################################################### for
column in ['age', 'avg_glucose_level', 'bmi']:
    Q1 = dataset[column].quantile(0.25)
    Q3 = dataset[column].quantile(0.75)     IQR = Q3 - Q1     newMin =
Q1 - 1.5 * IQR     newMax = Q3 + 1.5 * IQR     mean_value =
dataset[column].mean()     dataset[column] =
dataset[column].where((dataset[column] >= newMin) & (dataset[column] <=
newMax), mean_value)
```

This loop iterates over the following columns - 'age', 'avg_glucose_level', and 'bmi' - were outliers are to be replaced with the mean, Where for each column, the first quartile (Q1), third quartile (Q3), and interquartile range (IQR) are calculated. This is a common method for identifying outliers using the IQR method.

The mean value of the column is calculated. This will be used to replace outliers.
Finally, outliers in the column are replaced with the mean value using the **where()** method. Values that fall outside the defined range **(newMin to newMax)** are replaced with the **mean_value**.

4.5 Normalizing Numerical Data:

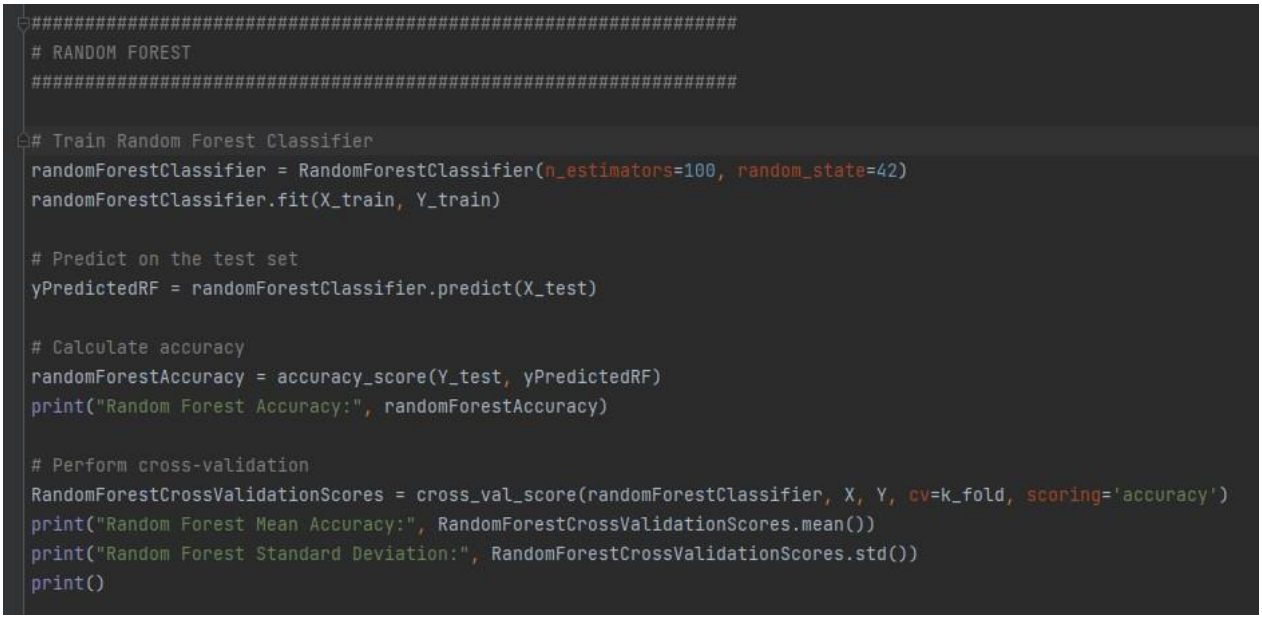
```
######################################################################
# Normalize Data
######################################################################
minMaxNormalizer = MinMaxScaler(feature_range=(0, 1))
NumericalColumns = dataset[['age', 'avg_glucose_level', 'bmi']]
dataset[NumericalColumns.columns] =
minMaxNormalizer.fit_transform(NumericalColumns)
```

A MinMaxScaler object is instantiated. Min-Max scaling is a method used to scale numerical values to a fixed range, for us it's between 0 and 1.
We select the numerical columns 'age', 'avg_glucose_level', and 'bmi', and these selected numerical columns are transformed using the **fit_transform()** method of the **MinMaxScaler**. This replaces the original values in the dataset with the scaled values.

6. Training Models & Test Performance

3 models were used in this project, they were trained using cross validation from the k-folds and the normal test train data split. They were finally tested, their accuracy was calculated and compared to each other.

6.1 Random Forest Model:



```
#################################################################
# RANDOM FOREST
#################################################################

# Train Random Forest Classifier
randomForestClassifier = RandomForestClassifier(n_estimators=100, random_state=42)
randomForestClassifier.fit(X_train, Y_train)

# Predict on the test set
yPredictedRF = randomForestClassifier.predict(X_test)

# Calculate accuracy
randomForestAccuracy = accuracy_score(Y_test, yPredictedRF)
print("Random Forest Accuracy:", randomForestAccuracy)

# Perform cross-validation
RandomForestCrossValidationScores = cross_val_score(randomForestClassifier, X, Y, cv=k_fold, scoring='accuracy')
print("Random Forest Mean Accuracy:", RandomForestCrossValidationScores.mean())
print("Random Forest Standard Deviation:", RandomForestCrossValidationScores.std())
print()
```

Breaking down the random forest model:

- **Initialization: RandomForestClassifier(n_estimators=100, random_state=42)**: This initializes a Random Forest classifier with 100 decision trees **(n_estimators=100)**. The **random_state=42** parameter ensures reproducibility by setting the random seed.

- **Training: randomForestClassifier.fit(X_train, Y_train)**:
  This trains the Random Forest classifier on the training data **(X_train, Y_train)**. During training, each decision tree in the forest is built on a random subset of the training data **(bootstrapping)** and a random subset of features.

- **Prediction: yPredictedRF = randomForestClassifier.predict(X_test)**:
  This predicts the target variable for the test data **(X_test)** using the trained Random Forest classifier.

- **Evaluation: randomForestAccuracy = accuracy_score(Y_test, yPredictedRF)**: This calculates the accuracy of the Random Forest model by comparing the predicted labels **(yPredictedRF)** with the actual labels for the test set **(Y_test)**.

- **Output: print("Random Forest Accuracy:", randomForestAccuracy)**: This prints out the accuracy of the Random Forest model on the test set.

6.2 Logistic Regression Model:

```
#####################################################################
# Logistic Regression
#####################################################################

# Train Logistic Regression classifier
logRegClassifier = make_pipeline(StandardScaler(), LogisticRegression())
logRegClassifier.fit(X_train, Y_train)

# Predict on the test set
yPredictedLogReg = logRegClassifier.predict(X_test)

# Calculate accuracy
logRegAccuracy = accuracy_score(Y_test, yPredictedLogReg)
print("Logistic Regression Accuracy:", logRegAccuracy)

# Perform cross-validation
logRegCrossValidationScores = cross_val_score(logRegClassifier, X, Y, cv=k_fold, scoring='accuracy')
print("Logistic Regression Mean Accuracy:", logRegCrossValidationScores.mean())
print("Logistic Regression Standard Deviation:", logRegCrossValidationScores.std())
print()
```

Breaking down the logistic regression model:

- **Initialization: make_pipeline(StandardScaler(), LogisticRegression())**:
  It creates a pipeline that first scales the input features using **StandardScaler()** and then applies logistic regression using **LogisticRegression()**. Scaling is crucial for logistic regression as it helps in improving convergence and regularization.

- **Training: logRegClassifier.fit(X_train, Y_train)**:
  trains the logistic regression model on the scaled training data **(X_train, Y_train)**. The model learns the relationship between the input features and the target variable.

- **Prediction: yPredictedLogReg = logRegClassifier.predict(X_test)**:
  It predicts the target variable for the test data **(X_test)** using the trained logistic regression model.

- **Evaluation: logRegAccuracy = accuracy_score(Y_test, yPredictedLogReg)**: Itcalculates the accuracy of the logistic regression model by comparing the predicted labels **(yPredictedLogReg)** with the actual labels for the test set **(Y_test)**.

- **Output: print("Logistic Regression Accuracy:", logRegAccuracy)**:
  prints out the accuracy of the logistic regression model on the test set.

6.3 XGBoost Model:

```
######################################################################
# XGBoost
######################################################################

# Train XGBoost classifier
xgbClassifier = xgb.XGBClassifier()
xgbClassifier.fit(X_train, Y_train)

# Predict on the test set
yPredictedXGB = xgbClassifier.predict(X_test)

# Calculate accuracy
xgbAccuracy = accuracy_score(Y_test, yPredictedXGB)
print("XGBoost Accuracy:", xgbAccuracy)

# Perform cross-validation
xgbCrossValidationScores = cross_val_score(xgbClassifier, X, Y, cv=k_fold, scoring='accuracy')
print("XGBoost Mean Accuracy:", xgbCrossValidationScores.mean())
print("XGBoost Standard Deviation:", xgbCrossValidationScores.std())
print()
```

Breaking down the XGBoost model (Extreme Gradient Boost):

- **Initialization: xgb.XGBClassifier()**:
  This initializes an XGBoost classifier with default hyperparameters. XGBoost is a gradient boosting framework that uses an ensemble of decision trees for classification tasks.

- **Training: xgbClassifier.fit(X_train, Y_train):**
  This trains the XGBoost classifier on the training data **(X_train, Y_train)**. During training, XGBoost sequentially builds decision trees minimizing the loss function, which is defined as the negative log-likelihood for classification tasks.

- **Prediction: yPredictedXGB = xgbClassifier.predict(X_test):**
  predicts the target variable for the test data **(X_test)** using the trained XGBoost classifier.

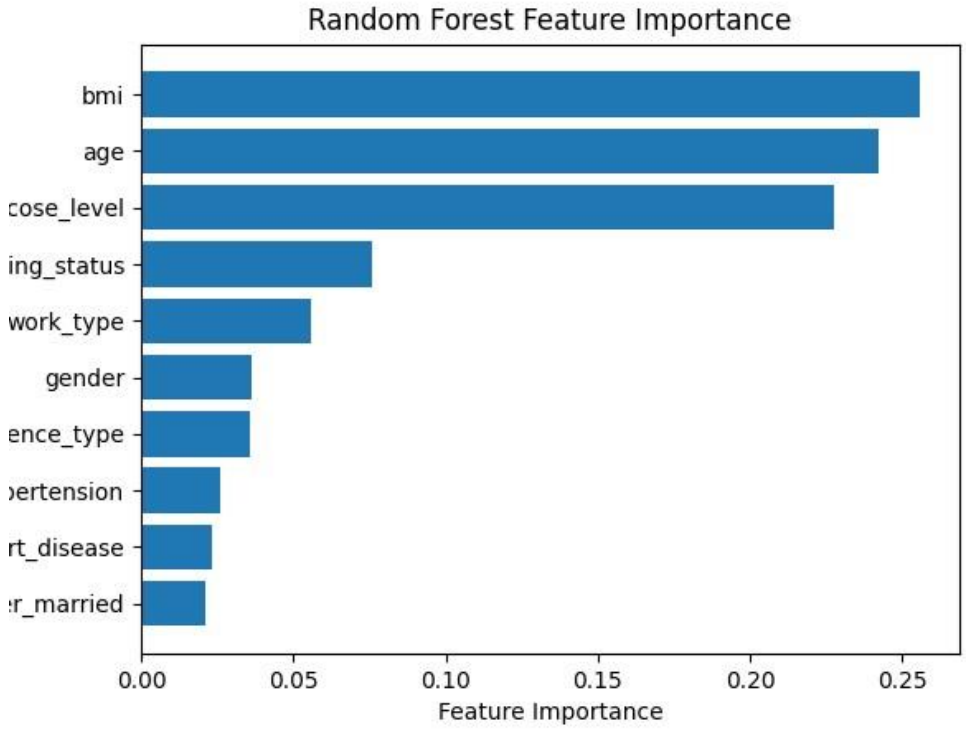
- **Evaluation: xgbAccuracy = accuracy_score(Y_test, yPredictedXGB)**:
  calculates the accuracy of the XGBoost model by comparing the predicted labels **(yPredictedXGB)** with the actual labels for the test set **(Y_test)**.

- **Output: print("XGBoost Accuracy:", xgbAccuracy)**:
  prints out the accuracy of the XGBoost model on the test set.

7. Evaluating Important Features

Now we get into the depth of the dataset and study the features in detail to find out which features contribute more to our decision. This determines the most influential features in the dataset that helps predict the target variable.

7.1 Random Forest Model:

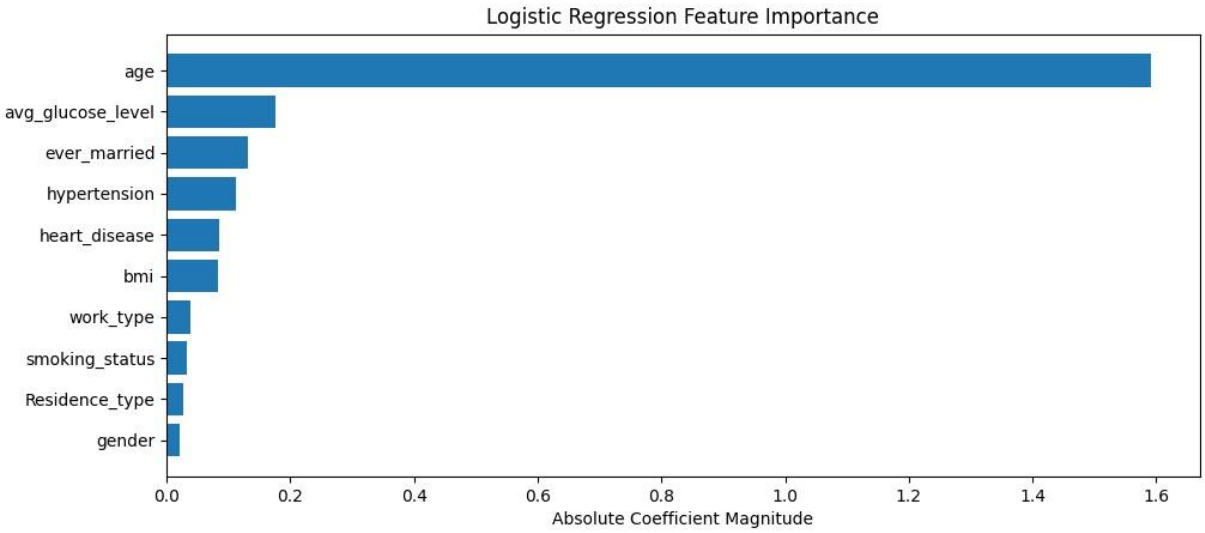



```
randomForestFeatureImportance = randomForestClassifier.feature_importances_
sortedIndexes = np.argsort(randomForestFeatureImportance)
plt.barh(range(X.shape[1]), randomForestFeatureImportance[sortedIndexes], align='center')
plt.yticks(range(X.shape[1]), np.array(dataset.columns.tolist())[sortedIndexes])
plt.xlabel('Feature Importance')
plt.ylabel('Feature')
plt.title('Random Forest Feature Importance')
plt.show()
```

Figure 6: Random Forest Feature Importance

- This part calculates the feature importance scores from the trained Random Forest Classifier **(randomForestClassifier.feature_importances_).**
- It sorts the feature importance scores and plots them in a horizontal bar chart, showing the importance of each feature in predicting the target variable.

7.2 Logistic Regression Model:



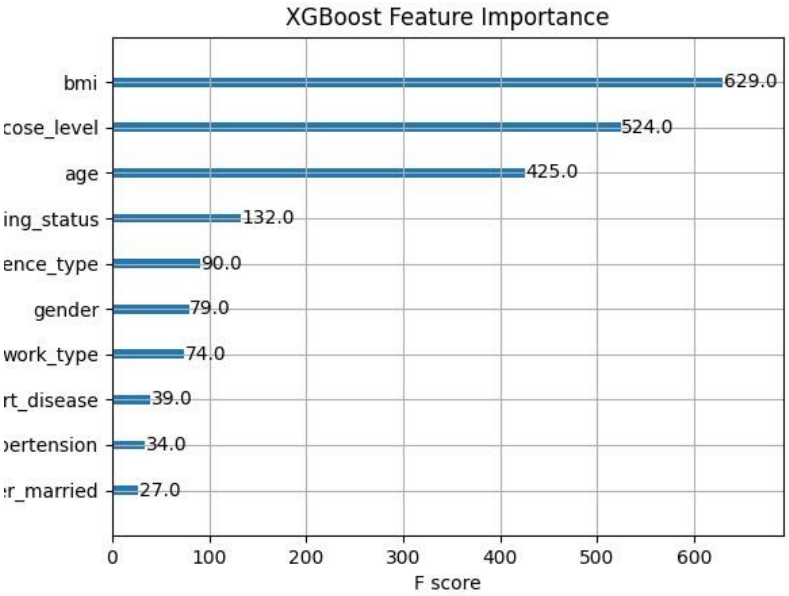

```
plt.show()

logRegModel = logRegClassifier.named_steps['logisticregression']
logRegCoefficients = np.abs(logRegModel.coef_[0])
sortedIndexes = np.argsort(logRegCoefficients)
plt.barh(np.array(dataset.columns.tolist()[:-1])[sortedIndexes], logRegCoefficients[sortedIndexes])
plt.xlabel('Absolute Coefficient Magnitude')
plt.ylabel('Feature')
plt.title('Logistic Regression Feature Importance')
plt.show()
```

Figure 7: Logistic Regression Feature Importance

- This part retrieves the coefficients of the Logistic Regression model **(logRegModel.coef_)** and takes the absolute values.

- It sorts the coefficients and plots them in a horizontal bar chart, indicating the importance of each feature in the Logistic Regression model.

7.3 XGBoost Model:




```
xgb.plot_importance(xgbClassifier)
plt.title('XGBoost Feature Importance')
plt.show()
```

Figure 8: XGBoost Feature Importance

- This part uses **XGBoost's** built-in function **plot_importance** to visualize feature importance.
- It directly plots the feature importance scores calculated by **XGBoost.**

8. Predicting Results and Comparing Accuracy

In this section we will study the dataset and try to get as much information as possible to better understand what variables we are working on, and their types and preprocessing methods needed for them.

8.1 Random Forest Model:

```
# Calculate accuracy
randomForestAccuracy = accuracy_score(Y_test, yPredictedRF)
print("Random Forest Accuracy:", randomForestAccuracy)

# Perform cross-validation
RandomForestCrossValidationScores = cross_val_score(randomForestClassifier, X, Y, cv=k_fold, scoring='accuracy')
print("Random Forest Mean Accuracy:", RandomForestCrossValidationScores.mean())
print("Random Forest Standard Deviation:", RandomForestCrossValidationScores.std())
print()
```

Figure 9: Random Forest code snippert

- After training the Random Forest model **(randomForestClassifier.fit(X_train, Y_train))**, predictions are made on the test data using **randomForestClassifier.predict(X_test)**.
- The predicted labels **(yPredictedRF)** are compared with the actual labels for the test set **(Y_test)** using accuracy_score**(Y_test, yPredictedRF).**
- The accuracy of the Random Forest model on the test set is calculated and printed out using **print("Random Forest Accuracy:", randomForestAccuracy).**

The accuracy score represents the percentage of correctly predicted labels out of all the labels in the test set.

8.2 Logistic Regression Model:

```
# Calculate accuracy
logRegAccuracy = accuracy_score(Y_test, yPredictedLogReg)
print("Logistic Regression Accuracy:", logRegAccuracy)

# Perform cross-validation
logRegCrossValidationScores = cross_val_score(logRegClassifier, X, Y, cv=k_fold, scoring='accuracy')
print("Logistic Regression Mean Accuracy:", logRegCrossValidationScores.mean())
print("Logistic Regression Standard Deviation:", logRegCrossValidationScores.std())
print()
```

Figure 10: Logistic Regression code snippet

- After training the Logistic Regression model **(logRegClassifier.fit(X_train, Y_train))**, predictions are made on the test data using **logRegClassifier.predict(X_test).**
- The predicted labels **(yPredictedLogReg)** are compared with the actual labels for the test set **(Y_test)** using accuracy_score**(Y_test, yPredictedLogReg).**
- The accuracy of the Logistic Regression model on the test set is calculated and printed out using **print("Logistic Regression Accuracy:", logRegAccuracy).**

8.3 XGBoost Model:

```
# Calculate accuracy
xgbAccuracy = accuracy_score(Y_test, yPredictedXGB)
print("XGBoost Accuracy:", xgbAccuracy)

# Perform cross-validation
xgbCrossValidationScores = cross_val_score(xgbClassifier, X, Y, cv=k_fold, scoring='accuracy')
print("XGBoost Mean Accuracy:", xgbCrossValidationScores.mean())
print("XGBoost Standard Deviation:", xgbCrossValidationScores.std())
print()
```

Figure 11: XGBoost code snippert

- After training the XGBoost model **(xgbClassifier.fit(X_train, Y_train))**, predictions are made on the test data using **xgbClassifier.predict(X_test)**.
- The predicted labels **(yPredictedXGB)** are compared with the actual labels for the test set **(Y_test)** using **accuracy_score (Y_test, yPredictedXGB)**.
- The accuracy of the XGBoost model on the test set is calculated and printed out using **print("XGBoost Accuracy:", xgbAccuracy).**

8.4 Comparison:

Output:

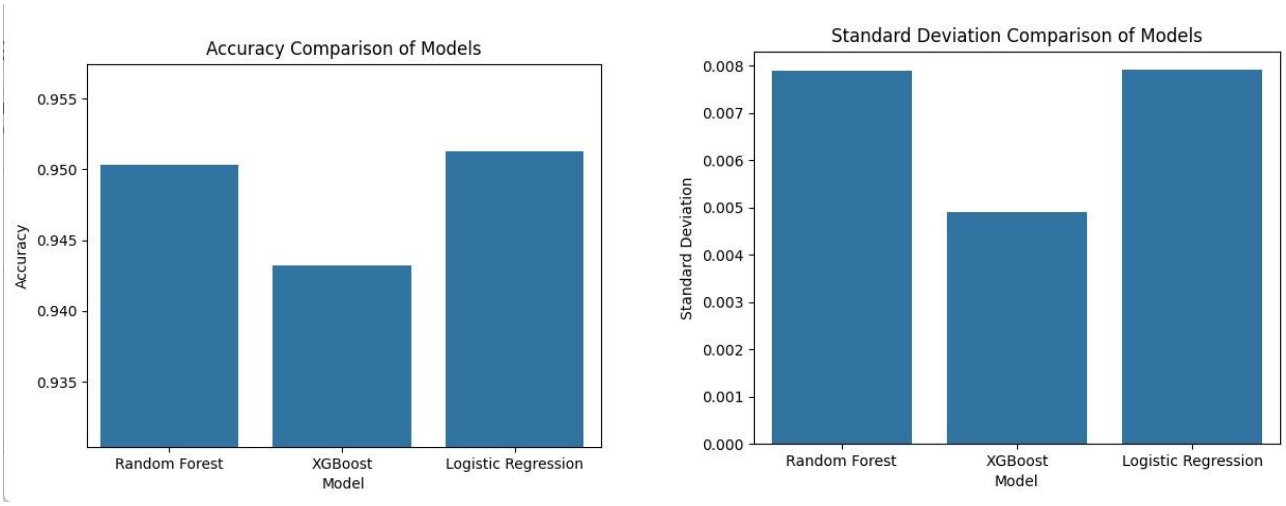


Figure 12: Models' Comparison output

Random Forest Accuracy: 0.9403131115459883
Random Forest Mean Accuracy: 0.950293542074364
Random Forest Standard Deviation: 0.007886278543878637

XGBoost Accuracy: 0.9373776908023483
XGBoost Mean Accuracy: 0.9432485322896282
XGBoost Standard Deviation: 0.004911898394524894

Logistic Regression Accuracy: 0.9393346379647749
Logistic Regression Mean Accuracy: 0.9512720156555773
Logistic Regression Standard Deviation: 0.007910521695267219

Breaking down the output:

1. **Random Forest:**
    - **Accuracy** on the test set: 94.03%
    - **Mean accuracy** from cross-validation: 95.03%
    - **Standard deviation** of accuracy from cross-validation: 0.79%
    - **Interpretation**: The Random Forest model achieved a high accuracy of 94.03% on the test set. The cross-validation results show that, on average, the model maintains a high accuracy of 95.03%, with a low variance of 0.79%, indicating robustness and consistency in performance across different folds of the data.
2. **XGBoost:**
    - **Accuracy** on the test set: 93.74%
    - **Mean accuracy** from cross-validation: 94.32%
    - **Standard deviation** of accuracy from cross-validation: 0.49%
    - **Interpretation:** The XGBoost model achieved an accuracy of 93.74% on the test set, which is slightly lower than the Random Forest model. However, the cross-validation results indicate that the model still performs well on average, with a mean accuracy of 94.32% and a low standard deviation of 0.49%.
3. **Logistic Regression:**
    - **Accuracy** on the test set: 93.93%
    - **Mean accuracy** from cross-validation: 95.13%
    - **Standard deviation** of accuracy from cross-validation: 0.79%
    - **Interpretation:** The Logistic Regression model achieved an accuracy of 93.93% on the test set, which is comparable to the Random Forest and XGBoost models. The crossvalidation results show that the model has a mean accuracy of 95.13%, similar to the Random Forest model, with a standard deviation of 0.79%.

Overall, all three models (Random Forest, XGBoost, and Logistic Regression) demonstrate strong performance on the test set, with accuracies ranging from 93.74% to 94.03%. The cross-validation results indicate that the models generalize well to unseen data, with mean accuracies consistently above 94% and low standard deviations, showing stable performance across different folds of the data.

9. Used Tools and Libraries

Project Dependencies

This project utilized the diverse capabilities of the Python ecosystem to manage data, generate insightful visualizations, develop machine learning models, and evaluate their effectiveness.

Data Handling:

- NumPy: Facilitated efficient array-based computations for numerical operations.
- Pandas: Provided a comprehensive set of tools for loading, cleansing, manipulating, and organizing data into structured DataFrames for analysis.

Data Preprocessing:

- Scikit-learn: Offered versatile functionalities for data preprocessing, including:
- Label Encoding: Converted categorical variables into numerical representations suitable for machine learning algorithms.
- Scaling: Standardized numerical features to a common scale using methods like MinMaxScaler or StandardScaler, ensuring uniform contribution of all features to model training.

Exploratory Data Analysis (EDA):

- Seaborn: Leveraging Matplotlib's foundation, Seaborn furnished an intuitive interface for crafting compelling visualizations that enhanced understanding of the data's characteristics and interrelations.

Machine Learning Algorithms:

- Scikit-learn: Serving as the cornerstone, this library encompassed a wide array of machine learning algorithms, such as:
- Random Forests: Employed an ensemble of decision trees to deliver robust classification performance.
- Logistic Regression: Implemented a linear model for classification tasks.
- XGBoost: Offering a potent implementation of gradient boosting trees, often yielding superior model performance compared to traditional methods.

Model Evaluation:

- Scikit-learn: Equipped with tools for model assessment, including:
- Train-Test Split: Segregated the data into training and testing subsets for model training and unbiased evaluation.
- Accuracy Scoring: Quantified the proportion of accurate predictions made by the trained model.
- Cross-Validation: Utilized techniques like K-fold cross-validation to furnish a more resilient appraisal of model generalizability on unseen data.

By harnessing these robust tools, the project proficiently analyzed data, constructed and assessed diverse machine learning models, ultimately attaining its objectives.

Conclusion

The goal of this project was to use data mining techniques to forecast the risk of stroke, enabling early detection and intervention. We achieved great accuracy (>93%) in stroke prediction by utilizing machine learning models such as Random Forest, Logistic Regression, and XGBoost, and evaluating patient data. The promise of data-driven techniques in healthcare was highlighted by the identification of key factors that influence forecasts. Utilizing Python libraries for data processing, model training, and evaluation, such as NumPy, Pandas, and Scikit-learn, was essential. In summary, the initiative demonstrates how data mining may be used to improve healthcare outcomes by proactively assessing risks and intervening when necessary.

References

https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset