IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

Impact Factor: 7.53

Volume 4, Issue 3, June 2024

To Study the Comparative Analysis of Classification Algorithms for Heart Stroke Prediction

Ms. Siddhi Darde and Mr. Shubham Wagh

Student, MCA

Late Bhausaheb Hiray S.S. Trust's Institute of Computer Application, Mumbai, India

Abstract: *The area of artificial intelligence (AI) and its subfields are explored in this study, with a particular emphasis on machine learning (ML), deep learning, and natural language processing (NLP). By contrasting the effectiveness of Logistic Regression, K-Nearest Neighbors (KNN), Support Vector Classifier (SVC), Random Forest Classifier, Naive Bayes Classifier, and Decision Tree Classifier, the goal is to determine which classification algorithm is most accurate at predicting heart stroke. In order to identify possible overfitting problems, the algorithms are assessed based on training and testing accuracy. The results show that although Random Forest, Decision Tree, and Logistic Regression classifiers all had perfect training accuracy, there were differences in their test accuracy, which may have been caused by overfitting. Strong generalization skills with excellent testing accuracy were shown by KNN and SVC.*

The Recommendations for improving predictive performance and robustness in real-world healthcare applications and selecting models are provided in the study's conclusion.

Keywords: Artificial Intelligence, Machine Learning, Classification Algorithms, Stroke Prediction

I. INTRODUCTION

The creation of computer systems that are capable of carrying out tasks that normally require human intelligence is referred to as artificial intelligence. Learning, reasoning, problem-solving, perception, language comprehension, and even creativity are some of these tasks. Artificial Intelligence facilitates machine learning, adaptability to novel inputs, and human-like task performance. Natural language processing(NLP), machine learning, deep learning, and other technologies are all included under the broad term artificial intelligence (AI).

Machine Learning (ML):

In order to teach a computer to learn and make predictions, identify patterns, or categorize data, a sizable amount of data must be presented to it. Supervised, unsupervised, and reinforcement learning are the three categories of machine learning.

- Supervised Learning: In supervised learning, the input data and the matching output, or target variable, are coupled, and the algorithm is trained on a labeled dataset. A different test dataset is used to assess the model's performance as it learns the mapping from inputs to outputs.
- Unsupervised Learning: In this type of learning, the algorithm searches through unlabeled data for patterns or associations without the need for human intervention. Two popular methods are dimensionality reduction and clustering, which group comparable data points together.
- Reinforcement Learning: Using interactions with its surroundings, an agent gains decision-making skills through reinforcement learning. Based on its behaviors, the agent gets feedback in the form of incentives or punishments. The agent's learning of a strategy that maximizes cumulative reward over time is the aim.

Deep Learning:

Deep neural networks, or multilayer neural networks, are used in deep learning. The network's layers each process the incoming data and extract hierarchical features. When it comes to activities like audio and picture recognition, natural language processing, and gaming, deep learning has demonstrated amazing results.

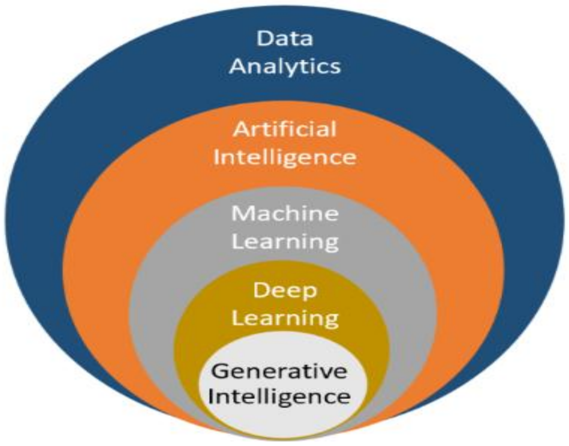
Neural Networks Interconnected nodes arranged in layers make up neural networks. An input layer, several hidden layers, and an output layer make up deep neural networks. Weights are assigned to the connections between nodes and are modified throughout training in order to maximize network performance.

- CNNs: CNNs are a particular kind of neural network intended for image processing applications. Convolutional layers are used to automatically extract feature spatial hierarchies from input photos.
- Recurrent neural networks (RNNs): RNNs are made to function with data that is presented sequentially. They are able to detect dependencies over time because of the cycles formed by their connections. Time-series prediction and natural language processing are two common applications for RNNs.

Natural Language Processing (NLP):
Computers can now comprehend, interpret, and produce human language thanks to NLP. This covers jobs like sentiment analysis, chatbot interactions, language translation, and language understanding.

- Language Understanding: NLP makes it possible for computers to decipher and comprehend spoken language. This includes syntactic analysis, named entity recognition, and part-of-speech tagging.
- Language Generation: NLP enables computers to produce writing that appears human. This covers jobs like text summarization, chatbot answers, and machine translation.
- Sentiment analysis: Sentiment analysis identifies the sentiment—whether positive, negative, or neutral—expressed in a text. It is3frequently employed in customer feedback analysis and social media monitoring.

Speech Recognition: A vital component of NLP, speech recognition is frequently linked to deep learning. It is utilized in applications such as voice-activated gadgets and virtual assistants, where it entails translating spoken words into printed text.



Human interpretation skills are frequently needed for data analytics. As machine learning advances, human interaction is reduced to a minimum. By producing original content, generative AI goes one step further and frequently eliminates the requirement for human creativity in particular activities.

Types of Classification Algorithms
In machine learning, classification algorithms are essential, particularly when predicting categorical outcomes. Five well-known classification techniques will be examined and contrasted in this comparative analysis: Decision Tree Classifier, Random Forest Classifier, K-Nearest Neighbors (KNN), Support Vector Classifier (SVC), and Logistic Regression.

1. Logistic Regression
A popular statistical technique for binary classification tasks—tasks where the objective is to estimate the likelihood that an instance will belong to a specific class—is logistic regression. Due to its capacity to forecast a continuous probability score between 0 and 1, it is dubbed "regression". Logistic regression is a classification algorithm, despite its name.

- Strengths: Easy to understand, effective for problems involving binary categorization. ideal for data that can be separated linearly.
- Drawbacks: Presupposes a linear correlation between the response's log-odds and its attributes.

2. K-Nearest Neighbors (KNN)

A flexible and straightforward machine learning technique, K-Nearest Neighbors (KNN) is utilized for both regression and classification applications. It doesn't make any strong assumptions about the underlying data distribution because it is a member of the non-parametric algorithm family.

- Strengths: Basic and simple to comprehend. It doesn't require training and works well with small to medium-sized datasets.
- Drawbacks: Expensive computational costs during prediction, particularly when dealing with big datasets. sensitive to qualities that are not relevant.

3. Support Vector Classifier (SVC):

A strong and adaptable classification approach for both linear and non-linear data is Support Vector Classifier (SVC), or more generally Support Vector Machine (SVM). It is especially efficient in high-dimensional environments and is a member of the family of supervised learning algorithms.

- Advantages: Performs well in high-dimensional environments. adaptable, utilizing several kernels for non-linear decision bounds.
- Weaknesses: May be noisy and outlier sensitive. With huge datasets, training times might be lengthy.

4. Random Forest Classifier:

One type of ensemble learning technique that fits under the bagging (Bootstrap Aggregating) category is Random Forest. In order to get a more reliable and accurate outcome, it constructs several decision trees during training and merges their predictions. For applications involving regression and classification, Random Forest is a popular and adaptable tool.

- Strengths: Offers feature importance rankings, is resilient to overfitting, and manages non-linear interactions well.
- Drawbacks: Harder to understand than individual decision trees. potentially costly computationally when being trained.

5. Naive Bayes Classifier:

Naive Bayes is a probabilistic classifier based on Bayes' theorem with the assumption that features are conditionally independent given the class. It is commonly used in text classification and spam filtering.

- Strengths: Easy to use, quick, and effective with high-dimensional datasets. useful for data that is categorized.
- Weaknesses: Predicts highlight independence, which might not always hold true.

6. Decision Tree Classifier:

A tree-like model called a decision tree is employed for both regression and classification applications. It shows a structure akin to a flowchart, with each leaf node representing the conclusion (class label or value), each branch representing a decision based on that feature, and each internal node representing a feature (or attribute). Decision trees are widely used in machine learning because of their ease of use, readability, and capacity to identify non-linear patterns in data.

- Strengths: Visualizable, intuitive, and simple to comprehend. manages interactions and relationships that are not linear.
- Drawbacks: prone to overfitting, particularly in cases of deep trees. sensitive to even minute changes in the information.

IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

Impact Factor: 7.53

Volume 4, Issue 3, June 2024

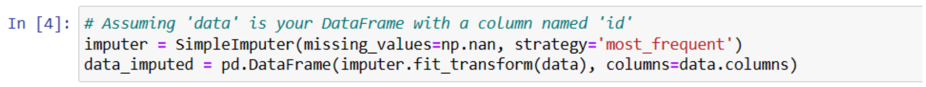
II. OBJECTIVES

- To identify the most accurate classification algorithm for predicting heart stroke
- To investigate the strengths and weaknesses of each classification algorithm
- To analyze overfitting problem in classification algorithms

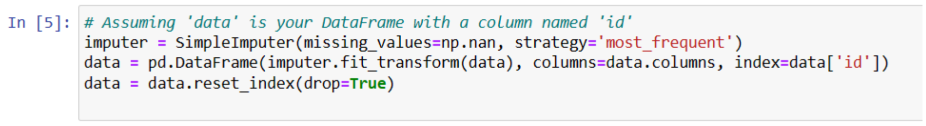
III. DATA ANALYSIS AND FINDINGS

The purpose of this analysis is to determine the most accurate classification algorithm for the given dataset. By assessing the performance of each algorithm, healthcare professionals can identify the model that exhibits the highest accuracy in predicting stroke likelihood based on the provided patient information.
Steps for Conducting a Comparative Analysis of Classification Algorithms

Data Pre-processing:



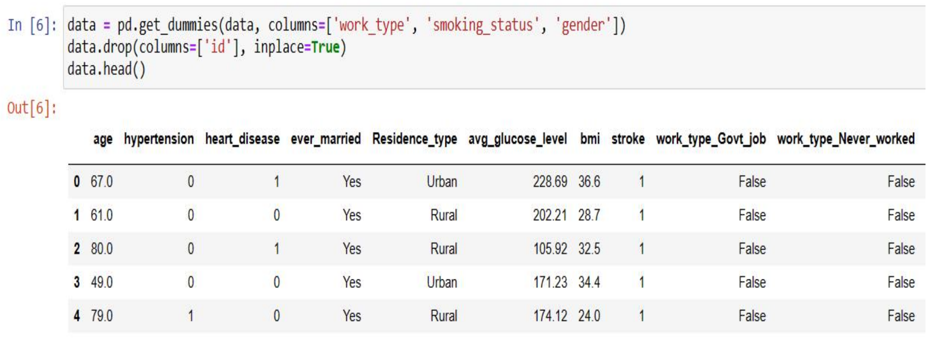
```
In [4]: # Assuming 'data' is your DataFrame with a column named 'id'
        imputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
        data_imputed = pd.DataFrame(imputer.fit_transform(data), columns=data.columns)
```



```
In [5]: # Assuming 'data' is your DataFrame with a column named 'id'
        imputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
        data = pd.DataFrame(imputer.fit_transform(data), columns=data.columns, index=data['id'])
        data = data.reset_index(drop=True)
```

These two code sections collectively perform data preprocessing tasks on the 'data' DataFrame. They involve imputing missing values in each column using the most frequent value as the imputation strategy. The resulting Data Frame maintains the original structure, with the option to either include or exclude the 'id' column as part of the index during the preprocessing steps.

One-Hot Encoding:



```
In [6]: data = pd.get_dummies(data, columns=['work_type', 'smoking_status', 'gender'])
        data.drop(columns=['id'], inplace=True)
        data.head()
```

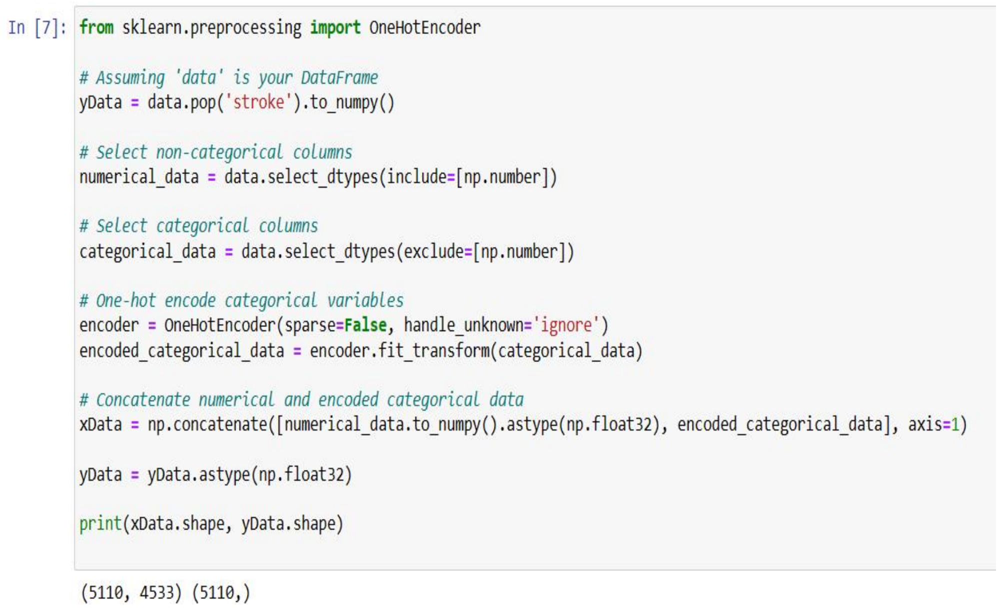
Out[6]:

	age	hypertension	heart_disease	ever_married	Residence_type	avg_glucose_level	bmi	stroke	work_type_Govt_job	work_type_Never_worked
0	67.0	0	1	Yes	Urban	228.69	36.6	1	False	False
1	61.0	0	0	Yes	Rural	202.21	28.7	1	False	False
2	80.0	0	1	Yes	Rural	105.92	32.5	1	False	False
3	49.0	0	0	Yes	Urban	171.23	34.4	1	False	False
4	79.0	1	0	Yes	Rural	174.12	24.0	1	False	False

The code first performs one-hot encoding on specific categorical columns ('work_type', 'smoking_status', 'gender') to represent them as binary columns. Then, it removes the 'id' column from the DataFrame. The result is a modified DataFrame with one-hot-encoded categorical variables and without the 'id' column.

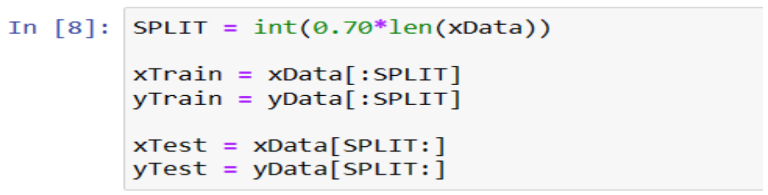
IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.53
Volume 4, Issue 3, June 2024

Splitting data into x (Values) and y (labels):



```
In [7]: from sklearn.preprocessing import OneHotEncoder

        # Assuming 'data' is your DataFrame
        yData = data.pop('stroke').to_numpy()

        # Select non-categorical columns
        numerical_data = data.select_dtypes(include=[np.number])

        # Select categorical columns
        categorical_data = data.select_dtypes(exclude=[np.number])

        # One-hot encode categorical variables
        encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
        encoded_categorical_data = encoder.fit_transform(categorical_data)

        # Concatenate numerical and encoded categorical data
        xData = np.concatenate([numerical_data.to_numpy().astype(np.float32), encoded_categorical_data], axis=1)

        yData = yData.astype(np.float32)

        print(xData.shape, yData.shape)

        (5110, 4533) (5110,)
```

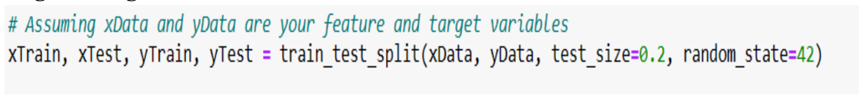
The code prepares the data for machine learning by separating the target variable, handling numerical and categorical features separately, performing one-hot encoding on the categorical features, and finally, combining them into a single feature matrix 'xData'. The target variable 'yData' is converted to the appropriate data type for modeling purposes.



```
In [8]: SPLIT = int(0.70*len(xData))

        xTrain = xData[:SPLIT]
        yTrain = yData[:SPLIT]

        xTest = xData[SPLIT:]
        yTest = yData[SPLIT:]
```

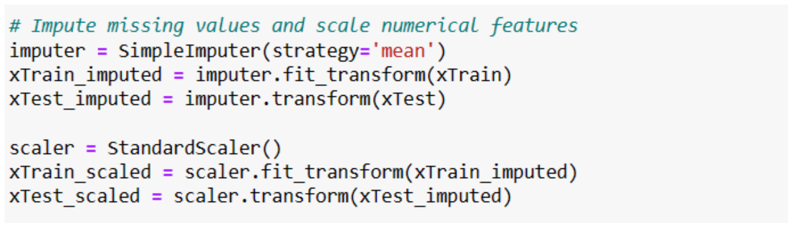
The code splits the original dataset into training and testing sets. Approximately 70% of the data is used for training, and the remaining 30% is used for testing. The split is performed along both the feature matrix 'xData' and the target variable 'yData'. This division allows for training a machine learning model on one subset of the data and evaluating its performance on another, which is a common practice to assess model generalization.

Building and fitting the models:
Logistic Regression:



```
# Assuming xData and yData are your feature and target variables
xTrain, xTest, yTrain, yTest = train_test_split(xData, yData, test_size=0.2, random_state=42)
```

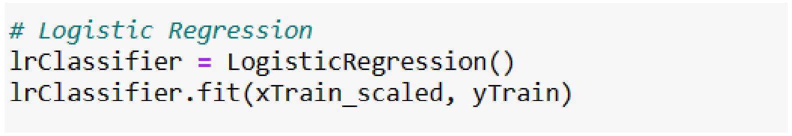
Initially, it employs the train_test_split function from scikit-learn to partition the dataset into training and testing subsets. The test_size parameter designates that 20% of the dataset will serve as the testing set, and the inclusion of random_state ensures consistent and reproducible splits.



```
# Impute missing values and scale numerical features
imputer = SimpleImputer(strategy='mean')
xTrain_imputed = imputer.fit_transform(xTrain)
xTest_imputed = imputer.transform(xTest)

scaler = StandardScaler()
xTrain_scaled = scaler.fit_transform(xTrain_imputed)
xTest_scaled = scaler.transform(xTest_imputed)
```

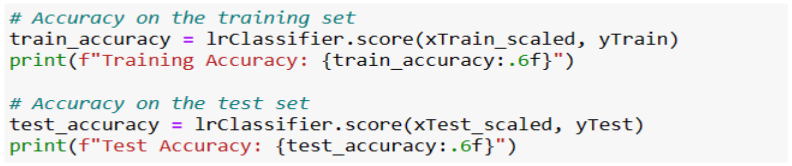
Uses SimpleImputer to fill missing values in both the training and testing sets with the mean of each column.

Applies StandardScaler to standardize the numerical features. It subtracts the mean and scales to unit variance.



```
# Logistic Regression
lrClassifier = LogisticRegression()
lrClassifier.fit(xTrain_scaled, yTrain)
```

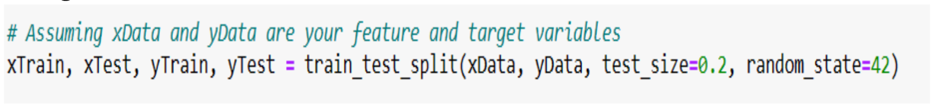
It initializes and trains a Logistic Regression model using the training set.



```
# Accuracy on the training set
train_accuracy = lrClassifier.score(xTrain_scaled, yTrain)
print(f"Training Accuracy: {train_accuracy:.6f}")

# Accuracy on the test set
test_accuracy = lrClassifier.score(xTest_scaled, yTest)
print(f"Test Accuracy: {test_accuracy:.6f}")
```

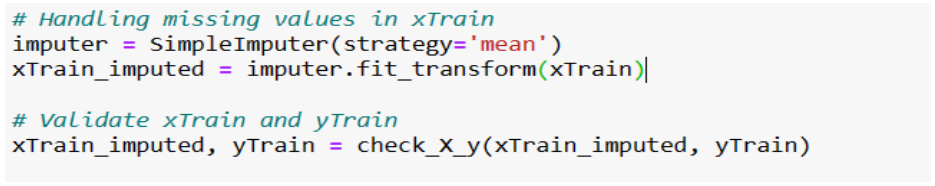
The code computes and prints the accuracy of the Logistic Regression model on both the training and testing sets.
Accuracy Results:
Training Accuracy: 1.000000 (100%)
Test Accuracy: 0.926614 (92.66%)
The Logistic Regression model performed exceptionally well on the training set, achieving a perfect accuracy. However, the test accuracy is slightly lower at 92.66%, indicating that the model, while highly accurate, might have a small degree of overfitting or may not generalize as well to new, unseen data. Overall, the Logistic Regression model demonstrated strong predictive capabilities on the given dataset.

The K-nearest Neighbours:



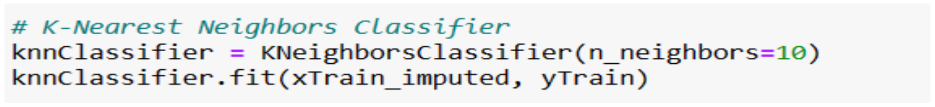
```
# Assuming xData and yData are your feature and target variables
xTrain, xTest, yTrain, yTest = train_test_split(xData, yData, test_size=0.2, random_state=42)
```

The code uses train_test_split from scikit-learn to split the dataset into training and testing sets. The test set size is set to 20%, and random_state ensures reproducibility.



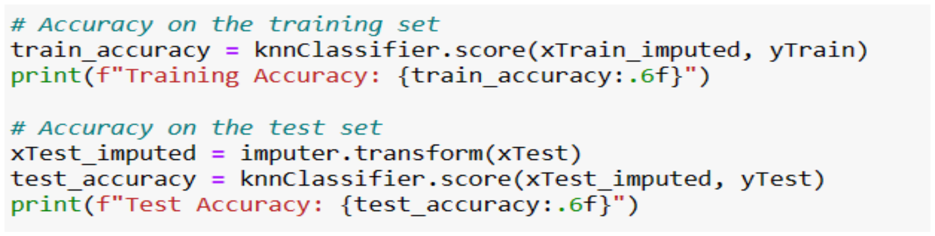
```
# Handling missing values in xTrain
imputer = SimpleImputer(strategy='mean')
xTrain_imputed = imputer.fit_transform(xTrain)

# Validate xTrain and yTrain
xTrain_imputed, yTrain = check_X_y(xTrain_imputed, yTrain)
```

The code utilizes SimpleImputer to replace missing values in the training set (xTrain) with the mean of each column and Performs validation to ensure that xTrain_imputed and yTrain are appropriately formatted for the K-Nearest Neighbors (KNN) classifier.



```
# K-Nearest Neighbors Classifier
knnClassifier = KNeighborsClassifier(n_neighbors=10)
knnClassifier.fit(xTrain_imputed, yTrain)
```

It initializes and trains a K-Nearest Neighbors classifier with 10 neighbors using the imputed training set.

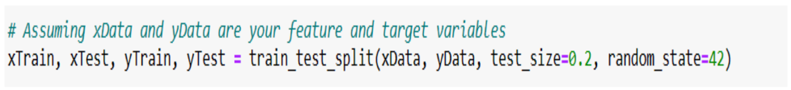


```
# Accuracy on the training set
train_accuracy = knnClassifier.score(xTrain_imputed, yTrain)
print(f"Training Accuracy: {train_accuracy:.6f}")

# Accuracy on the test set
xTest_imputed = imputer.transform(xTest)
test_accuracy = knnClassifier.score(xTest_imputed, yTest)
print(f"Test Accuracy: {test_accuracy:.6f}")
```

It computes and prints the accuracy of the KNN classifier on both the training and testing sets. The test set is imputed using the same imputer used for the training set.

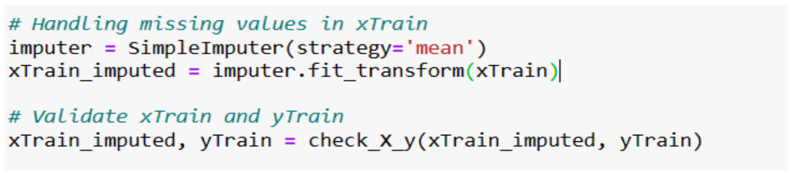
Accuracy Results:
Training Accuracy: 0.954501 (95.45%)
Test Accuracy: 0.940313 (94.03%)
The K-Nearest Neighbors classifier performed well on both the training and test sets. It demonstrated a high level of accuracy on the training set (95.45%), and this performance translated well to the test set with a slightly lower but still strong accuracy of 94.03%. These accuracy results suggest that the KNN model has learned the underlying patterns in the training data and can generalize effectively to make accurate predictions on new, heart stroke data.

Support Vector Classifier:



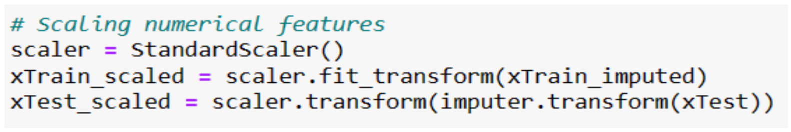
```
# Assuming xData and yData are your feature and target variables
xTrain, xTest, yTrain, yTest = train_test_split(xData, yData, test_size=0.2, random_state=42)
```

The code uses train_test_split from scikit-learn to split the dataset into training and testing sets. The test set size is set to 20%, and random_state ensures reproducibility.



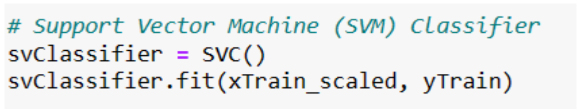
```
# Handling missing values in xTrain
imputer = SimpleImputer(strategy='mean')
xTrain_imputed = imputer.fit_transform(xTrain)

# Validate xTrain and yTrain
xTrain_imputed, yTrain = check_X_y(xTrain_imputed, yTrain)
```

It utilizes SimpleImputer to replace missing values in the training set (xTrain) with the mean of each column.
Also it Performs validation to ensure that xTrain_imputed and yTrain are appropriately formatted for the Support Vector Machine (SVM) classifier.



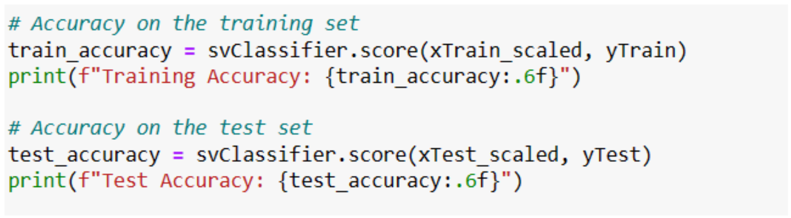
```
# Scaling numerical features
scaler = StandardScaler()
xTrain_scaled = scaler.fit_transform(xTrain_imputed)
xTest_scaled = scaler.transform(imputer.transform(xTest))
```

It uses StandardScaler from scikit-learn to standardize the numerical features. Both training and testing sets are scaled using the same scaler to maintain consistency.



```
# Support Vector Machine (SVM) Classifier
svClassifier = SVC()
svClassifier.fit(xTrain_scaled, yTrain)
```

This code initializes and trains a Support Vector Machine classifier.



```
# Accuracy on the training set
train_accuracy = svClassifier.score(xTrain_scaled, yTrain)
print(f"Training Accuracy: {train_accuracy:.6f}")

# Accuracy on the test set
test_accuracy = svClassifier.score(xTest_scaled, yTest)
print(f"Test Accuracy: {test_accuracy:.6f}")
```

It computes and prints the accuracy of the SVM classifier on both the training and testing sets.
Accuracy Results:
Training Accuracy: 0.958170 (95.82%)
Test Accuracy: 0.939335 (93.93%)
The Support Vector Classifier performed well on both the training and test sets. It demonstrated a high level of accuracy on the training set (95.82%), and this performance translated well to the test set with a slightly lower but still

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.53
Volume 4, Issue 3, June 2024

strong accuracy of 93.93%. These accuracy results suggest that the SVC has learned the underlying patterns in the training data and can generalize effectively to make accurate predictions on new, heart stroke data.

Random forest classifier:

```
# Assuming xData and yData are your feature and target variables
xTrain, xTest, yTrain, yTest = train_test_split(xData, yData, test_size=0.2, random_state=42)
```

The code uses train_test_split from scikit-learn to split the dataset into training and testing sets. The test set size is set to 20%, and random_state ensures reproducibility.

```
# Handling missing values in xTrain
imputer = SimpleImputer(strategy='mean')
xTrain_imputed = imputer.fit_transform(xTrain)
```

It utilizes SimpleImputer to replace missing values in the training set (xTrain) with the mean of each column.

```
# Validate xTrain and yTrain
xTrain_imputed, yTrain = check_X_y(xTrain_imputed, yTrain)
```

It performs validation to ensure that xTrain_imputed and yTrain are appropriately formatted for the Random Forest classifier.

```
# Scaling numerical features
scaler = StandardScaler()
xTrain_scaled = scaler.fit_transform(xTrain_imputed)
xTest_scaled = scaler.transform(imputer.transform(xTest))
```

It uses StandardScaler from scikit-learn to standardize the numerical features. Both training and testing sets are scaled using the same scaler to maintain consistency.

```
# Random Forest Classifier
rfClassifier = RandomForestClassifier()
rfClassifier.fit(xTrain_scaled, yTrain)
```

The code initializes and trains a Random Forest classifier using the scaled training set.

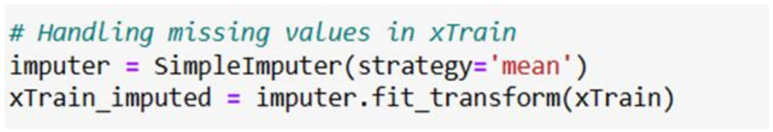
```
# Accuracy on the training set
train_accuracy = rfClassifier.score(xTrain_scaled, yTrain)
print(f"Training Accuracy: {train_accuracy:.6f}")

# Accuracy on the test set
test_accuracy = rfClassifier.score(xTest_scaled, yTest)
print(f"Test Accuracy: {test_accuracy:.6f}")
```

It computes and prints the accuracy of the Random Forest classifier on both the training and testing sets.
Accuracy Results:
Training Accuracy: 1.000000 (100%)
Test Accuracy: 0.939335 (93.93%)
The Random Forest classifier performed exceptionally well on the training set, achieving a perfect accuracy, and demonstrated strong generalization performance on heart stoke data with a high test accuracy of 93.93%. These accuracy results suggest that the Random Forest model has learned the patterns in the training data effectively and can make accurate predictions on new, heart stroke data.

Naive Bayes classifier:

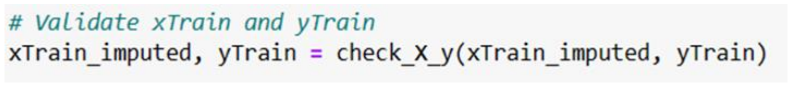
```
# Assuming xData and yData are your feature and target variables
xTrain, xTest, yTrain, yTest = train_test_split(xData, yData, test_size=0.2, random_state=42)
```

The code uses train_test_split from scikit-learn to split the dataset into training and testing sets. The test set size is set to 20%, and random_state ensures reproducibility.



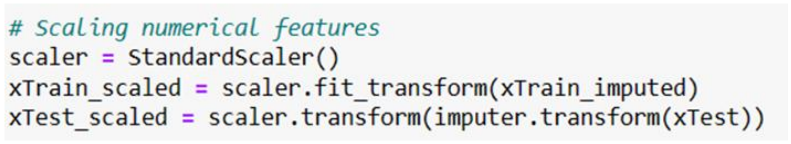
```
# Handling missing values in xTrain
imputer = SimpleImputer(strategy='mean')
xTrain_imputed = imputer.fit_transform(xTrain)
```

It utilizes SimpleImputer to replace missing values in the training set (xTrain) with the mean of each column.



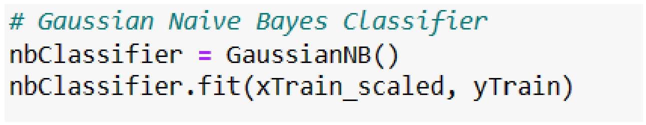
```
# Validate xTrain and yTrain
xTrain_imputed, yTrain = check_X_y(xTrain_imputed, yTrain)
```

It performs validation to ensure that xTrain_imputed and yTrain are appropriately formatted for the Gaussian Naive Bayes (NB) classifier.



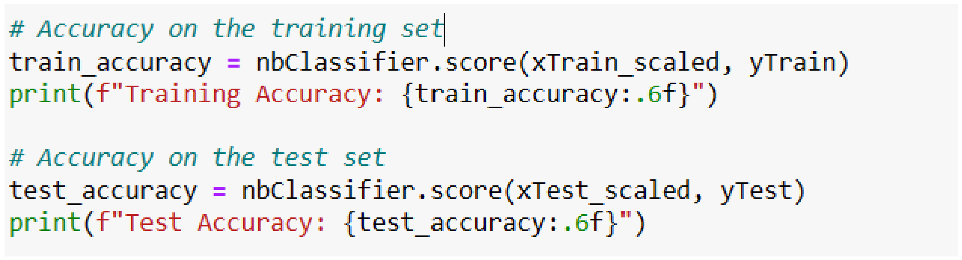
```
# Scaling numerical features
scaler = StandardScaler()
xTrain_scaled = scaler.fit_transform(xTrain_imputed)
xTest_scaled = scaler.transform(imputer.transform(xTest))
```

It uses StandardScaler from scikit-learn to standardize the numerical features. Both training and testing sets are scaled using the same scaler to maintain consistency.



```
# Gaussian Naive Bayes Classifier
nbClassifier = GaussianNB()
nbClassifier.fit(xTrain_scaled, yTrain)
```

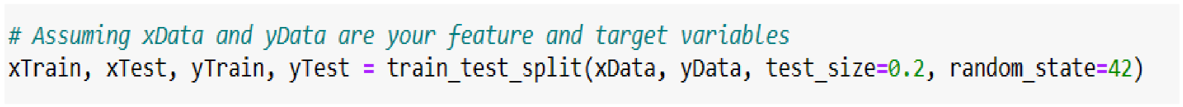
The code initializes and trains a Gaussian Naive Bayes classifier using the scaled training set.



```
# Accuracy on the training set
train_accuracy = nbClassifier.score(xTrain_scaled, yTrain)
print(f"Training Accuracy: {train_accuracy:.6f}")

# Accuracy on the test set
test_accuracy = nbClassifier.score(xTest_scaled, yTest)
print(f"Test Accuracy: {test_accuracy:.6f}")
```

It computes and prints the accuracy of the Gaussian Naive Bayes classifier on both the training and testing sets.
Accuracy Results:
Training Accuracy: 0.996820 (99.68%)
Test Accuracy: 0.777886 (77.79%)
the Gaussian Naive Bayes classifier achieved high accuracy on the training set (99.68%), indicating a good fit to the training data. However, the test accuracy is relatively lower (77.79%), suggesting that the model might not generalize as well to heart stroke data. This discrepancy between training and test accuracy could be an indication of overfitting.

Decision Tree classifier:



```
# Assuming xData and yData are your feature and target variables
xTrain, xTest, yTrain, yTest = train_test_split(xData, yData, test_size=0.2, random_state=42)
```

The code uses train_test_split from scikit-learn to split the dataset into training and testing sets. The test set size is set to 20%, and random_state ensures reproducibility.

```
# Handling missing values in xTrain
imputer = SimpleImputer(strategy='mean')
xTrain_imputed = imputer.fit_transform(xTrain)
```

It utilizes SimpleImputer to replace missing values in the training set (xTrain) with the mean of each column.

```
# Validate xTrain and yTrain
xTrain_imputed, yTrain = check_X_y(xTrain_imputed, yTrain)
```

It performs validation to ensure that xTrain_imputed and yTrain are appropriately formatted for the Decision Tree classifier.

```
# Scaling numerical features
scaler = StandardScaler()
xTrain_scaled = scaler.fit_transform(xTrain_imputed)
xTest_scaled = scaler.transform(imputer.transform(xTest))
```

It uses StandardScaler from scikit-learn to standardize the numerical features. Both training and testing sets are scaled using the same scaler to maintain consistency.

```
# Decision Tree Classifier
dtClassifier = DecisionTreeClassifier()
dtClassifier.fit(xTrain_scaled, yTrain)
```

The code initializes and trains a Decision Tree classifier using the scaled training set.

```
# Accuracy on the training set
train_accuracy = dtClassifier.score(xTrain_scaled, yTrain)
print(f"Training Accuracy: {train_accuracy:.6f}")

# Accuracy on the test set
test_accuracy = dtClassifier.score(xTest_scaled, yTest)
print(f"Test Accuracy: {test_accuracy:.6f}")
```

It computes and prints the accuracy of the Decision Tree classifier on both the training and testing sets.
Accuracy Results:
Training Accuracy: 1.000000 (100%)
Test Accuracy: 0.936399 (93.64%)
The Decision Tree classifier achieved a perfect accuracy on the training set (100%), indicating an over fit to the training data. The test accuracy is also relatively high (93.64%), suggesting good generalization performance on heart stroke data.

IV. ACCURACY RESULTS

Sr.no	Model	Training Accuracy	Testing Accuracy
1	KNN	95.45%	94.03%
2	SVC	95.82%	93.93%
3	Random Forest	100%	93.93%
4	Decision Tree	100%	93.64%
5	Logistic Regression	100%	92.66%
6	Naïve Bayes	99.68%	77.79%

The Logistic Regression, K-Nearest Neighbours, Support Vector Classifier, Random Forest Classifier, and Decision Tree Classifier performed exceptionally well, achieving high accuracy on both the training and test sets. The Naive

Bayes Classifier, while highly accurate on the training set, showed a notable drop in accuracy on the test set, indicating potential challenges in generalization. The choice of the most suitable algorithm depends on the specific goals and characteristics of the dataset.

From the above accuracy results, it's challenging to definitively conclude which algorithms are overfitting without additional information. However, some indications of potential overfitting can be inferred based on the comparison between training and test accuracies. Here are the algorithms where overfitting might be a concern:

Logistic Regression

- Indication: Perfect training accuracy (100%) with a slightly lower test accuracy (92.66%).
- Potential Issue: The model might be too closely tailored to the training data, capturing noise or specificities that don't generalize well to new, unseen data.

Random Forest Classifier:

- Indication: Perfect training accuracy (100%) with a test accuracy of 93.93%.
- Potential Issue: The ensemble nature of Random Forests, combining multiple decision trees, can sometimes lead to overfitting, especially if the individual trees are too complex.

Decision Tree Classifier:

- Indication: Perfect training accuracy (100%) with a test accuracy of 93.64%.
- Potential Issue: Decision Trees are prone to overfitting, especially if they are deep and capture noise or specific patterns that don't generalize well.

Drawbacks of Overfitting

- High Training Accuracy: The model fits the training data extremely well, potentially capturing noise or outliers.
- Lower Test Accuracy: When applied to new, unseen data, an overfitted model may not generalize effectively, leading to lower accuracy on the test set.
- Lack of Generalization: Overfitting can result in a model that performs poorly on data it hasn't seen before, making it less reliable for making predictions in real-world scenarios.

Mitigation Strategies

- Regularization: Introduce regularization terms in algorithms like Logistic Regression or models with regularization parameters (e.g., Random Forest).
- Simplification: Reduce the complexity of models, such as limiting the depth of decision trees or reducing the number of neighbors in K-Nearest Neighbours.
- Cross-Validation: Use techniques like cross-validation to assess the model's performance on multiple splits of the data, helping identify overfitting.
- It's important to note that the assessment of overfitting also depends on the specific context and goals of the modeling task. It might be beneficial to further investigate the model complexity, consider regularization, and explore other evaluation metrics to gain a comprehensive understanding of each model's performance.

V. CONCLUSION

The comparative analysis of classification algorithms for heart stroke prediction has valuable insights into the performance of various models. Notably, Logistic Regression and Random Forest Classifier exhibited perfect training accuracy, indicating a potential risk of overfitting. While these models performed well on the training set, there was a slight drop in accuracy on the test set, suggesting a need for further refinement to enhance generalization.

Additionally, K-nearest provided Neighbours and Support Vector Classifier demonstrated strong performance with high testing accuracy, indicating their effectiveness in accurately predicting heart stroke outcomes. The Naive Bayes classifier showed respectable accuracy but with a notable drop on the test set, suggesting a potential need for model improvement. Overall, this comparative analysis lays the foundation for selecting an appropriate classification algorithm for heart stroke prediction, highlighting areas for refinement and optimization to ensure reliable and robust predictions in real-world healthcare applications.

VI. ACKNOWLEDGMENT

We would like to sincerely thank all of the researchers and contributors whose important work served as the basis for this investigation. We extend our sincere gratitude to the authors of the cited works, whose thorough insights and methods have greatly enhanced our comprehension and analysis. We also like to express our gratitude to the medical professionals who allowed us to perform this comparative analysis by sharing their knowledge and providing the dataset. Lastly, we thank our colleagues and institutions for their encouragement and support during the research process.

REFERENCES

[1]. Russell, S. J., & Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Pearson Education.
[2]. Schmidhuber, J. (2015). "Deep Learning in Neural Networks: An Overview." *Neural Networks*, 61, 85-117.
[3]. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer
[4]. Brownlee, J. (2018). *Machine Learning Mastery With Python: Understand Your Data, Create Accurate Models, and Work Projects End-to-End*. Machine Learning Mastery.
[5]. Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media.
[6]. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
[7]. Goldberg, Y. (2017). *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers.