HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Project Report

Stroke Risk Prediction

Course: Machine Learning
Supervisor: Assoc. Prof. Than Quang Khoat

Authors:	*Student ID:*
Nguyen Thanh An	20225541
Pham Tuan Anh	20225542
Le Tien Dat	20225543
Nguyen Quang Hung	20225545
Pham Doan Phuc Lam	20225546
Le Hai Nhat	20225583

Hanoi, May 2024

Abstract

Stroke is a medical condition in which blood vessels in the brain rupture, causing brain damage. According to the World Health Organization (WHO), stroke is the leading cause of death and disability worldwide. Early recognition of stroke warning signs can help reduce the severity of stroke.

Many machine learning (ML) models have been used to predict the likelihood of stroke occurrence. In this study, a dataset containing clinical and personal variables was used and several machine learning models were implemented to predict stroke and then to validate the results. The results highlight the significant importance of oversampling for imbalanced datasets like ours.

Table of contents

1. Introduction

A stroke is defined as an acute neurological disorder of the blood vessels in the brain that occurs when the blood supply to an area of the brain stops and the brain cells are deprived of the necessary oxygen. This can lead to brain cells dying within minutes, making stroke a medical emergency that requires immediate attention. Strokes can lead to severe disability or death, making an early detection and intervention crucial.

Factors that elevate the risk of having a stroke include a previous stroke, transient ischemic attacks, myocardial infarction, and other heart conditions such as heart failure and atrial fibrillation. Age is a significant factor; individuals over 55 are more likely to be affected, although strokes can occur at any age, including in children. Other risk factors are hypertension, carotid stenosis due to atherosclerosis, smoking, high cholesterol levels, diabetes, obesity, sedentary lifestyle, alcohol consumption, blood clotting disorders, estrogen therapy, and the use of substances like cocaine and amphetamines.

This project introduces a methodology for creating effective binary classification machine learning models for predicting stroke occurrence. Given the importance of class balancing for effective stroke prediction methods, the synthetic minority over-sampling technique (SMOTE) was utilized. Various models were then developed, configured, and evaluated on the balanced dataset. The models assessed included Naive Bayes, K-NN, decision trees, random forests, support vector machine (SVM), neural network and XG-boost. Additionally, majority voting and stacking methods were employed, with stacking being the main contribution of this study. The experiments demonstrated the superiority of the stacking method over single models and voting, achieving high precision, recall, F-measure, and accuracy.

2. Data Set

2.1. Dataset description

Our research based on a dataset from Kaggle **dataset**. The original dataset has total 5110 participants, but here we take out randomly the data of 6 people for later testing. So our dataset now only has 5104 participants, and all of the attributes (10 as input to ML models and 1 for target class) are described as follows:

- **age**: Refers to participant's age.
- **gender**: Refers to participant's gender. The number of *Male* is 2111. whilst the number of *Female* is 2992 and 1 *Other*.
- **hypertension**: Refers to whether participant has hypertension or not. *1* for yes and *0* for no.
- **heart_disease**: Refers to whether participant has to suffer with heart disease. *1* for yes and *0* for no.
- **ever_married**: Represents the marital status of the participants. *Yes* or *No*.
- **work_type**: Represents the working status of participants. We have 5 categories (*children*, *Govt_job*, *Never_worked*, *Private* or *Self-employed*).
- **Residence_type**: Represents where a person live, *Rural* or *Urban*.
- **avg_glucose_level**: Captures the participant's average glucose level.
- **bmi**: Represent the bmi of a person.
- **smoking_status**: Captures the smoking status. (*formerly smoked*, *never smoked*, *smokes* or *Unknown*).
- **stroke**: Represents if the participant previously had a stroke or not.

Except age, average_glucose, bmi are numerical, most attributes are nominal.

2.2. Data Analysis and Preprocessing

The following analysis shows a clear picture of how our dataset is.

Figure 1 represents the distribution in attribute **stroke** (*1* means the participant had a stroke in the past, while *0* for not). With 4.8% participants had stroke in the past, we can conclude that our data is highly unbalanced.

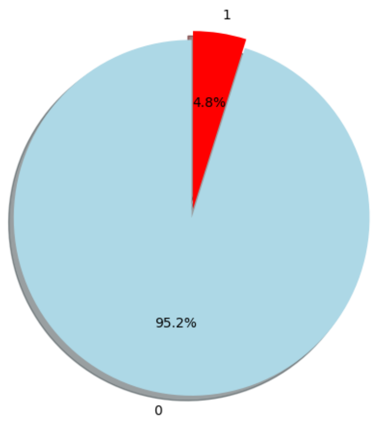


Figure 1. The distribution of stroke class

Figure 2 illustrates the distribution among work type and stroke class. We can see that people who work private and self-employed have higher chance of getting stroke, while people from government are more likely to not have a stroke. Moreover children and never worked participants are not very likely to get stroke. Maybe this could be explain due to the degree if pressure felt by workers.

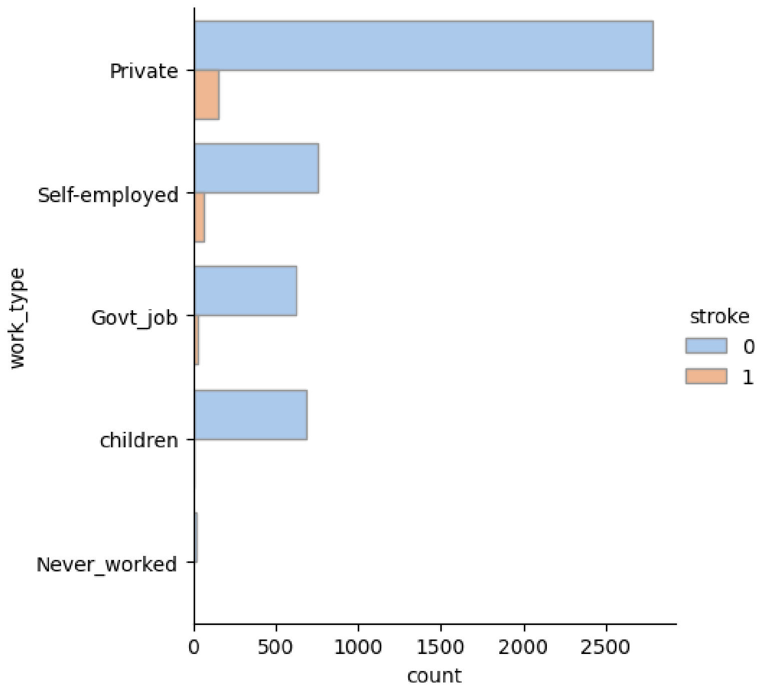


Figure 2. The distribution among work type and stroke class

Figure 3 illustrates the distribution among smoking status and stroke class. The stroke is not highly related to smokers, since the proportion of person having stroke is fairly the same among different status.

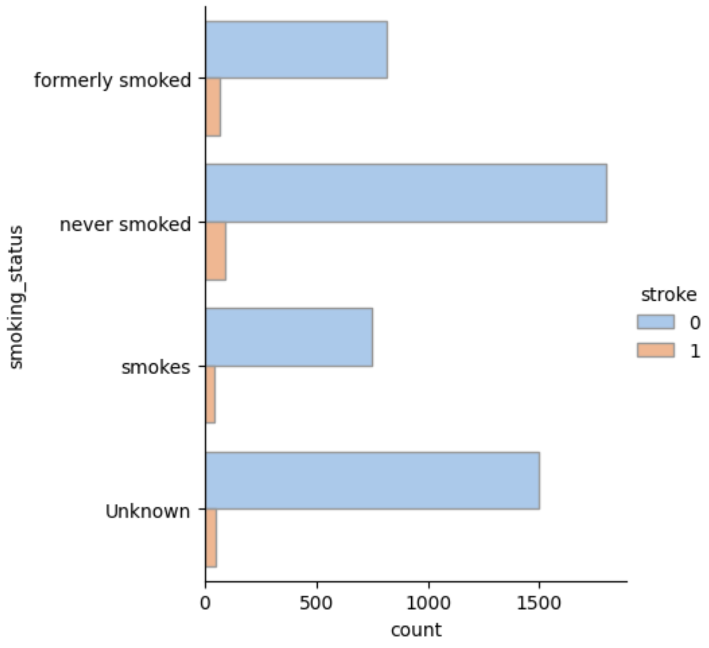
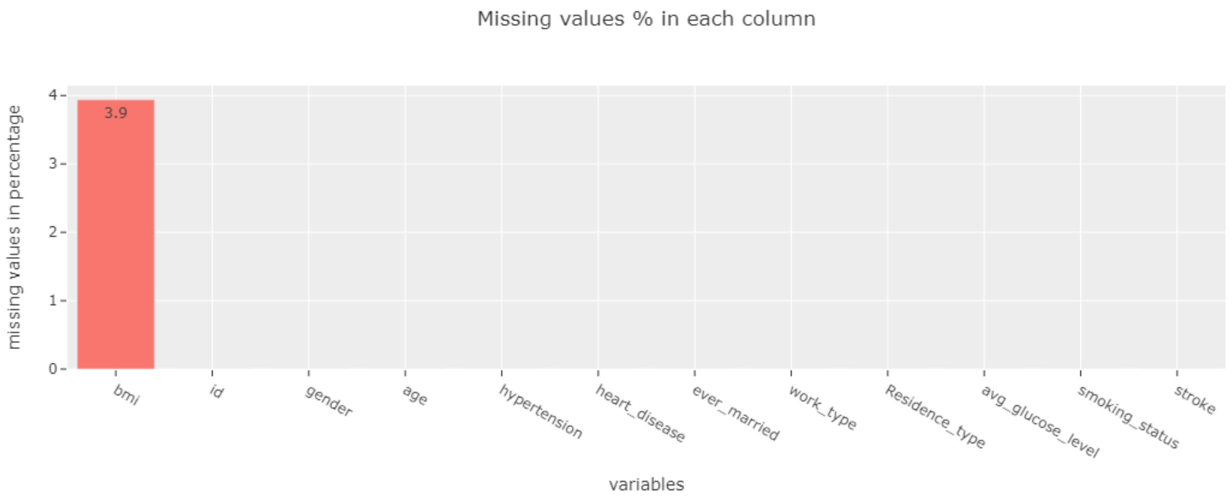


Figure 3. The distribution among smoking status and stroke class

From the figure missing values % in each columns, we can see that **bmi** is missing some data, at about 201 data.



To deal with missing data problem, we prefer to assign mean of bmi to these missing values.

As can be seen from figure 1, our data is unbalanced. In our proposed framework, we employed a so-called SMOTE **smote** to address the imbalanced distribution of participants among the stroke and non-stroke classes.

More specifically, the minority class, in this case, the "stroke", was oversampled, such that the participants were equally distributed. In addition, after fixing the missing data problem, there were no more missing or null values, so neither dropping nor data imputation was further applied.

3. Proposed models

We present the models that will be utilized in the classification framework for stroke occurrence. For this purpose, several types of classifier are employed.

3.1. Bernoulli Naive Bayes

The Naive Bayes algorithm is a probabilistic classifier based on Bayes' Theorem. It assumes independence among features, making it simple yet effective for various tasks. Commonly used in text classification, spam detection, and sentiment analysis, it calculates the probability of each class given the input features. Despite its "naive" assumption of feature independence, it often performs well in practice, especially with large datasets. The model is fast, scalable, and easy to implement, making it a popular choice for many machine learning applications.

Algorithm Overview

The probability calculations in Naive Bayes are influenced by the likelihood of the features. Therefore, there are three commonly used variants of Naive Bayes: Gaussian, Multinomial, and Bernoulli. The nature of the problem at hand will determine which variant is most suitable.

For our problem of predicting stroke, we use Bernoulli Naive Bayes, suitable for continuous data, assuming that the features follow a Bernoulli distribution. Let c denotes the hypothesis that there is a stroke, and $\bar{c}$ denotes the hypothesis that there is no stroke. Each X is composed of features $x_1, x_2, x_3, \ldots, x_n$, which are the attributes or characteristics used to make the prediction, such as age, glucose levels, and body mass index (BMI).

According to Bayes' theorem:

$$P(c|X) = \frac{P(X|c)P(c)}{P(X)}$$

$$P(\bar{c}|X) = \frac{P(X|\bar{c})P(\bar{c})}{P(X)}$$

where $P(c|X)$ and $P(\overline{c}|X)$ are posterior probabilities, $P(X|c)$ and $P(X|\overline{c})$ are likelihoods, $P(c)$ and $P(\overline{c})$ are class prior probabilities, and $P(X)$ is predictor prior probabilities.

To confirm whether we have a stroke or not, we compare $P(c|X)$ and $P(\overline{c}|X)$. More simply, we can compare $P(X|c)P(c)$ and $p(X|\overline{c})P(\overline{c})$. We have:

$$P(X|c)P(c) = P(x_1|c) \times P(x_2|c) \times \ldots \times P(x_n|c) \times P(c)$$

$$P(X|\overline{c})P(\overline{c}) = P(x_1|\overline{c}) \times P(x_2|\overline{c}) \times \ldots \times P(x_n|\overline{c}) \times P(\overline{c})$$

$P(c)$ is the proportion of stroke in training dataset, and $P(\overline{c}) = 1 - P(c)$.

To determine $P(x_i|c)$, we consider all the stroke in the training dataset, count the number of occurrences of x_i and divide it by the total number of patients.

$$P(x_i|c) = \frac{N_c x_i}{N_c}$$

$P(x_i|\overline{c})$ is determined in the same way.

Parameters

- **alpha**: It is a smoothing parameter used in the Naive Bayes algorithm. This parameter is to handle the problem of zero probabilities in the model.
- **binarize**: This parameter is used to threshold the feature values to binary values (0 or 1).

Choosing the suitable parameters

With default parameters on Bernoulli Naive Bayes classifier, we have following result:

Accuracy	**75.78%**
Precision	**72.14%**
Recall	**83.81%**
F1 score	**77.54%**

Since default parameters show a temporary stable performance on metric scores, we perform a grid search over a range of hyperparameters to identify optimal values, using GridSearchCV class in Python's Scikit-learn library to automate this process.

The parameter grid is for this is:

```
{"alpha" : [1.0, 1.25, 1.5, 1.75, 2.0], "binarize" = [0.0, 0.25, 0.5,
```

After performing the search, we found:

```
{"alpha" = 1.0, "binarize" = 0.5}
```

We have following result:

Accuracy	**76.04%**
Precision	**72.26%**
Recall	**84.33%**
F1 score	**77.83%**

However, after using the parameter, which is founded by GridSearchCV, the score of the model have no significant change. This problem maybe due to the lack of experiments involving other hyperparameters such as `"class_prior"`, `"force_alpha"`, `etc.`

3.2. Support Vector Machine

A support vector machine (SVM) is a supervised machine learning algorithm that classifies data by finding an optimal line or hyperplane that maximizes the distance between each class in an N-dimensional space. It is one of the most popular and strong methods for classification since it can work well with very high dimensional problems.

Types of SVM

- **Linear SVM**:

  Linear SVM is used for classification when the dataset is linear separable. The task is finding a hyperplane with max margin. There are two approaches to find that hyperplane.

  - *Hard-margin classification*: If the dataset is perfectly linear separable, hard-margin classification will provide accurate results. However, it is sensitive to noise.
  - *Soft-margin classification*: If the dataset is nearly linear separable, soft-margin classification will be applied. This method relaxes the constraint about the margin by using some slack variables, allowing for some misclassification to find reasonable results.

- **Non-linear SVM**:

Much of the data in real-world scenarios are not linearly separable, and that's where nonlinear SVM come into play. In order to make the data linearly separable, preprocessing methods are applied to the training data to transform it into a higher-dimensional feature space. That said, higher dimensional spaces can create more complexity by increasing the risk of overfitting the data and by becoming computationally taxing. The "kernel trick" helps to reduce some of that complexity, making the computation more efficient, and it does this by replacing dot product calculations with an equivalent kernel function.

Each kernel can be applied to different problems. Some of common kernel include:

- Linear kernel

$$k(x, z) = x^T z$$

- Polynomial kernel

$$k(x, z) = (r + \gamma x^T z)^d$$

- Gaussian kernel

$$k(x, z) = \exp(-\gamma ||x - z||_2^2) \ , \quad \gamma > 0$$

- Sigmoid kernel

$$k(x, z) = \tanh(r + \gamma x^T z)$$

Models

In this project, we use 2 common kernels: *linear* and *Gaussian*

- For linear kernel:
  - C: Regularization parameter. The strength of the regularization is inversely proportional to C. We trained model with $C \in [0.1, 20]$.
- For Gaussian kernel:
  - $C \in [0.1, 20]$
  - $\gamma \in [0.01, 1]$

Linear Kernel

Increasing value of C parameter resulted in long execution time. However, the accuracy score always around 78%. The table below gives the result of linear kernel with $C = 1.0$.

Accuracy Score	78.20%
Precision Score	75.62%
Recall Score	83.79%
F1 Score	79.50%

Result of linear kernel with $C = 1.0$

Gaussian Kernel

After training, the most feasible values are $C = 3.0$ and $\gamma = 0.2$. The table below gives the result.

Accuracy Score	95.78%
Precision Score	96.19%
Recall Score	95.40%
F1 Score	95.79%

Result of Gaussian kernel with $C = 3.0$ and $\gamma = 0.2$

3.3. Decision Tree and Random Forest

3.3.1. Decision Tree

A tree-like model of decision and their possible consequences, including chance event outcomes, resource cost, and utility.

Parameters

- **criterion**: Here we specify which method we will choose when performing split operations. Partition is the most important concept in decision trees. It is very crucial to determine how to split and when to split.
- **splitter**: It is the strategy of how to split node.
- **max_feature**: The number of features to consider when splitting, We can determine the maximum number of features to be used by giving integers and proportionally by giving float number.

Choosing the suitable parameters

With default hyperparameters on Decision Tree classifier, we have following result:

Accuracy score	**91.20%**
Precision score	**90.57%**
Recall score	**92.14%**
F1 score	**91.35%**

Default parameters show a good performance on metric scores, but we still perform a grid search over a range of hyperparameters to identify optimal values, using GridSearchCV class in Python's scikit-learn library to automate this process. The parameter grid is for this is:

```
{"criterion" : ["gini" , "entropy", "log_loss"],
 "splitter" : ["best", "random"],
 "max_features" : ["auto", "sqrt", "log2"]}
```

After performing the search, we found:

```
"criterion": "log_loss", "max_features": "log2", "splitter": "best"
```

With following table result:

Accuracy score	**89.81%**
Precision score	**88.94%**
Recall score	**91.12%**
F1 score	**90.02%**

However, after using founded parameter, the score of the model is lower than the score of default parameter. This may be due to the lack of experiments involving other hyperparameters such as "max_depth", "max_leaf_node", .etc...

3.3.2. Random Forest Classifier

Random Forest (RF) **RF** ensembles many independent decision trees and, by resampling, creates different subsets of instances to perform classification and regression tasks. Each decision tree exports its own classification outcome, and then the final class is derived through majority voting.

Parameters

- **criterion**: Here we specify which method we will choose when performing split operations. Partition is the most important concept in decision trees. It is very crucial to determine how to split and when to split.
- **max_feature**: The number of features to consider when splitting, We can determine the maximum number of features to be used by giving integers and proportionally by giving float number.

Choosing the suitable parameters

With default hyperparameters on Random Forest classifier, we have following result:

Accuracy score	**95.06%**
Precision score	**94.07%**
Recall score	**96.20%**
F1 score	**95.12%**

Default parameters show a good performance on metric scores, but we still perform a grid search over a range of hyperparameters to identify optimal values, using GridSearchCV class in Python's scikit-learn library to automate this process. The parameter grid is for this is:

```
{'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
 'criterion' :[ 'gini' , 'entropy','log_loss'],
 'max_features': ['auto', 'sqrt', 'log2']}
```

After performing the search, we found:

```
{"criterion"="log_loss", "max_features"="sqrt", "max_depth"=100}
```

Accuracy	**93.98%**
Precision	**92.22%**
Recall	**96.39%**
F1 score	**94.26%**

Similar to decision tree, we experienced a worse result. However the recall is higher is higher, but it is negligible.

3.4. K-nearest Neighbors

K-Nearest Neighbors (KNN) is a popular versatile machine learning algorithm used for both classification and regression tasks. It's known for its simplicity, ease of implementation, and interpretability. KNN works by classifying (or predicting) data points based on the labels of their nearest neighbors in the training data. The number of neighbors considered is a crucial parameter, denoted by "k".

Classifying with KNN

Given with a dataset with labeled instances, KNN classifies new instances by analyzing the labels of its closest neighbors in the training data. To *Analyze* the labels of its k nearest neighbors, model determine distances from current instance to others. Then analyze labels of each instances.

Parameters

- **K (Number of Neighbors)**: The value of K significantly impacts KNN's performance. Experimenting with different k values using techniques like cross-validation is crucial to find the optimal k for your dataset.
- **Distance Metric**: is a function used to determine the similarity or "closeness" between two data points. KNN relies on this distance metric to identify the k nearest neighbors of a new data point for classification or regression tasks. Here are some common distance metrics used in KNN:
  - **Euclidean Distance**: This is the most widely used metric. It calculates the straight-line distance between two points in n-dimensional space. Imagine points plotted on a graph - Euclidean distance is the familiar distance formula used to find the length of that line.

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

  - **Manhattan Distance**: Also known as the L1 norm or city block distance, it calculates the total distance traveled along each dimension to get from one point to another.

$$\sum_{i=1}^{n}|x_i - y_i|$$

  - **Cosine Similarity**: This metric is often used for text data or high-dimensional data. It reflects the cosine of the angle between two data points. Intuitively, a higher cosine similarity indicates a more similar direction between the points.

$$\frac{\sum_{i=1}^{n} x_i \times y_i}{\sqrt{\sum_{i=1}^{n} {x_i}^2} \times \sqrt{\sum_{i=1}^{n} {y_i}^2}}$$

- **Weighting Neighbors**: it refers to assigning different importance to the k nearest neighbors identified for a new data point. By default, KNN treats all k neighbors equally. However, weighting allows you to prioritize closer neighbors and give them more influence in the final prediction.

Choosing suitable parameters

The value of K is crucial because it determines how many neighbors influence the prediction for a new data point. With a too small k, The model becomes oversensitive to noise in the training data and may not capture the underlying patterns well. Imagine a noisy neighborhood - a small k might focus on the closest neighbors, even if they're noisy outliers. Besides, a too large k can lead model be underfitting and the model might lose sight of the local patterns and become overly generalized. After iterating k from 1 to 30 and examining the performance of model, I sum up this below graph.

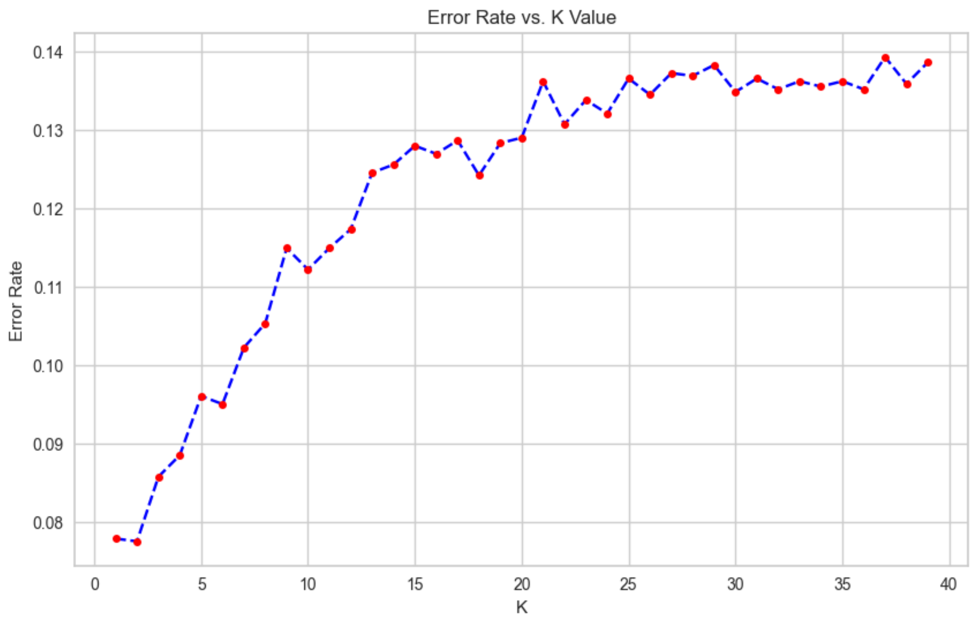


Figure 1: Experimental error rate with each k neighbors

After analyzing above graph, larger k is equivalent to larger error rate. And we can see that when k = 2 the error rate is smallest. Here is model's score:

Accuracy score	92.21%
Precision score	91.56%
Recall score	92.84%
F1 score	92.20%

Test result for optimal value k = 2

After finding optimal value for k, I perform a grid search over a range of hyperparameters to identify other optimal values, using the GridSearchCV class that has been installed within Python's scikit-learn library to automate this process. The parameter grid is :

```
"neighbors" : [1, 30],
"metric" : ["euclidean", "manhattan", "cosine"],
"weights": ["uniform", "distance"]
```

Grid search return the following parameters

```
"neighbors"=1, "metric"="manhattan", "weights": "uniform"
```

with result:

Accuracy score	92.38%
Precision score	89.33%
Recall score	96.04%
F1 score	92.56%

But we have an optimal k (neighbors value) = 2, while grid search return optimal k = 1, so we test our model again with:

```
"neighbors"=2, "metric"="manhattan", "weights": "uniform"
```

and we received the result:

Accuracy score	92.86%
Precision score	92.01%
Recall score	93.67%
F1 score	92.83%

Overall, this is better than the above two cases.

3.5. Logistic Regression

Logistic regression accomplishes binary classification task by predicting the probability of an outcome, event, or observation. In context of stroke risk prediction, it attempts to determine whether a participant has higher chance to get stroke or not.

Parameters

L1 and L2 regularization are techniques used to prevent overfitting in machine learning models. We continue to use the GridSearchCV to find the suitable parameters among:

```
"solver": ["lbfgs", "liblinear"],
"max_iter": [100, 200, 300, 400, 500],
"multi_class": ["auto", "ovr", "multinomial"],
"penalty": ["l1", "l2", "elasticnet", "none"],
```

The optimal parameters we found are

```
"solver"="lbfgs", "max_iter"=200,
"multi_class"="auto", penalty"="l2"
```

with the following result:

Accuracy score	75.76%
Precision score	75.58%
Recall score	76.88%
F1 score	76.22%

Test result for tuning Logistic Regression

3.6. XGBoost

XGBoost **XGBoost**, which stand for Extreme Gradient Boosting, based on gradient boosting. However, it comes with great improvements in algorithm optimization and the perfect combination of software and hardware power, helping to achieve outstanding results in both training time and memory used.

Parameters

- **n_estimators**: Determines the number of boosting iterations and controls the overall complexity of the model.
- **learning_rate**: An optimal parameter where step size shrinkage prevents overfitting.
- **booster**:
- **gamma**: L1 regularization term on weights
- **reg_lambda**: L2 regularization term on weights.

Choosing suitable parameters

Similar to other methods, we first run the model with default parameters:

Accuracy score	**94.49%**
Precision score	**92.75%**
Recall scorer	**96.63%**
F1 score	**94.65%**

We keep using GridSearchCV to automate identify the optimal values for the hyperparameters. The grid for this is:

```
{"n_estimators": [100, 500, 1000], "learning_rate"': [0.01, 0.1, 0.5],
 "booster": ["gbtree", "gblinear"], "gamma": [0, 0.5, 1],
 "reg_lambda": [0, 0.5, 1]}
```

The optimal parameters we found is

```
{"booster"="gbtree", "gamma"=0, "learning_rate"=0.1,
 "n_estimator"=1000, "reg_lambda"=0}
```

With following table result:

Accuracy score	**95.57%**
Precision score	**95.00%**
Recall score	**96.18%**
F1 score	**95.59%**

4. Results

4.1. Evaluation metrics

Under the evaluation process of the considered ML models, several performance metrics were recorded. In this project, we have used some commonly metrics.

Denote:

- TP: the number of true positives
- TN: the number of true negatives
- FP: the number of false positives
- FN: the number of false negatives

We evaluate the performance of our system based of the following criteria

- $Accuracy = \dfrac{TP + TN}{TP + TN + FP + FN}$
- $Precision = \dfrac{TP}{TP + FP}$
- $Recall = \dfrac{TP}{TP + FN}$
- $F1\ score = 2 \times \dfrac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$

4.2. Compare results

Performance among different models

	Accuracy	**Precision**	**Recall**	**F1**
Naive Bayes	76.04%	72.26%	84.33%	77.83%
SVM (Linear)	78.20%	75.62%	83.79 %	79.50%
SVM (Gaussian)	95.78%	96.19%	95.40%	95.79%
Decision Tree	91.20%	90.57%	92.14%	91.35%
Random Forest	95.06%	94.07%	96.20%	95.12%
KNN Classifier	92.86%	92.01%	93.67%	92.83%
Logistic Regression	75.76%	75.58%	76.88%	76.22%
XGBoost	95.57%	95.00%	96.18%	95.59%

Performance of different models

Except Naive Bayes, SVM (linear kernel) and Logistic regression show poor performance, the other models show amazing performance (over 91%).

Random Forest, SVM(Gaussian kernel) and XGBoost illustrate the best performance, at around 95%. There are a few differences in their recall score and precision score. SVM (Gaussian kernel) is the finest since its accuracy score and f1-score was the highest, at **95.78%** and **95.79%**, respectively.

5. Conclusions

A stroke constitutes a threat to a human's life that should be prevented and/or treated to avoid unexpected complications. Nowadays, with the rapid evolution of AI/ML, the clinical providers, medical experts and decision-makers can exploit the established models to discover the most relevant features (or risk factors) for the stroke occurrence, and can assess the respective probability or risk.

In this direction, machine learning can aid in the early prediction of stroke and mitigate the severe consequences. This study investigates the effectiveness of various ML algorithms to identify the most accurate algorithm for predicting stroke based on several features that capture the participants' profiles.

The performance evaluation of the classifiers using F-measure (which summarizes precision and recall) and accuracy is essentially suitable for the models' interpretation, demonstrating their classification performance. In addition, they reveal the models' validity and predictive ability in terms of the stroke class. Support Vector Machine with Gaussian kernel classification outperforms the other methods, with an F1-score of 95.79% and an accuracy of 95.78%. Hence, a SVM (Gaussian) is an efficient approach for identifying those at high risk of experiencing a stroke in the long term.

REFERENCES