Machine Learning Report for Midterm

By: Harshikaa Thiagu

Abstract

The aim of this project was to investigate the performance of three classification algorithms (Logistic Regression, Decision Tree, and Naive Bayes) on an imbalanced healthcare dataset for cerebral stroke prediction. I implemented the Naive Bayes classification algorithm from scratch. To address the issue of class imbalance, each classifier was evaluated using three resampling techniques: Random Oversampling, Random Undersampling, and Synthetic Minority Oversampling Technique (SMOTE). A 5-fold cross-validation method was used to ensure robust evaluation. Classifier performance was assessed using key evaluation metrics such as accuracy, precision, recall, and F1-score. The best-performing combination was determined based on the F1-score for the minority class. The combination of Logistic Regression and SMOTE demonstrated the most balanced performance across all evaluation metrics. It achieved the highest F1-score proving to be the most efficient at identifying stroke cases in the imbalanced dataset.

Link to Kaggle Dataset Used:
https://www.kaggle.com/datasets/shashwatwork/cerebral-stroke-predictionimbalaced-dataset

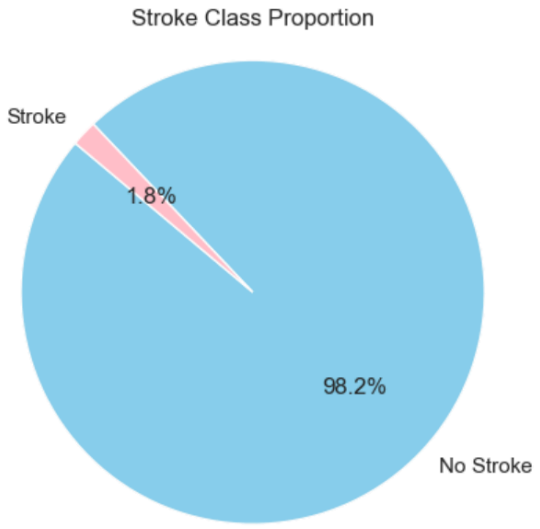
Introduction

In recent years, machine learning has become increasingly influential in healthcare. This is noted by Flam (2025), who mentioned: "Machine learning for healthcare has seen exponential growth, offering groundbreaking capabilities that range from improving diagnostic accuracy to personalizing patient treatment plans" (Flam, 2025).

Historically, diagnosing illness depended heavily on the "experience of the healthcare professional, and to some extent their intuition, along with available tests" (EIT Health, 2024). However, machine learning is reshaping the landscape of diagnostics by quickly analysing large amounts of medical data and spotting patterns that may go unnoticed by humans. This is particularly helpful for detecting serious illnesses like cancer or heart disease at an early stage, when treatment is more likely to succeed.

This project builds on that shift by applying machine learning to the prediction of cerebral strokes, a life-threatening condition that benefits significantly from early detection. However, effective stroke prediction poses a unique challenge. Real-world medical datasets, like the one used in this project, are often highly imbalanced, as positive cases (patients who suffered a stroke) are significantly outnumbered by negative cases.

For example, in the dataset I used, stroke-positive cases accounted for only 1.8% of the total records. This severe imbalance makes it challenging for standard classifiers to learn meaningful patterns related to the minority class without introducing bias.



Addressing this imbalance is therefore essential to improving predictive performance, especially in a medical context where false negatives (failing to identify actual stroke cases) and false positives (incorrectly flagging healthy individuals as at risk) can have serious consequences. In this project, I particularly focused on how different combinations of classifiers and resampling techniques influence the ability to detect stroke cases effectively and fairly.

Background

This project explores the performance of three widely used classification algorithms (Logistic Regression, Decision Tree, and Naive Bayes) in the context of cerebral stroke prediction using an imbalanced dataset. Each algorithm offers a unique classification strategy and was used in combination with three resampling techniques (Random Oversampling, Random Undersampling, and SMOTE) to handle the imbalance in the dataset.

1.1 Logistic Regression

Logistic Regression is a supervised learning algorithm commonly used for binary classification problems, such as the one addressed in this project. It is a linear model that estimates the probability of a given input belonging to a specific class by applying a sigmoid function to a weighted combination of feature inputs.

In this project, I implemented Logistic Regression using the 'scikit-learn' library and evaluated its performance under different resampling conditions to address class imbalance. To ensure fair and robust evaluation, I applied 5-fold stratified cross-validation. This helped preserve the original class distribution within each fold, which is essential given the skewed nature of the dataset. It also helped reduce the risk of overfitting and provided more reliable average performance estimates.

The implementation was structured using a pipeline to integrate preprocessing and model training, allowing it to be applied iteratively across different resampling techniques. For the baseline model, which involved no resampling, I used a 'SklearnPipeline' that included feature scaling with 'StandardScaler' and classification using 'LogisticRegression'. For the resampling approaches (Random Oversampling, Random Undersampling, and SMOTE), I used an 'ImblearnPipeline' that included feature scaling with 'StandardScaler', resampling using 'RandomOverSampler()', 'RandomUnderSampler()' and 'SMOTE()', followed by classification using 'LogisticRegression'.

These pipelines were applied to 'X_train' and 'y_train', which were obtained from a stratified train-test split to maintain the original class proportions in both sets.

Each pipeline was evaluated using the following classifier evaluation metrics: accuracy, precision, recall, and F1-score. These were chosen to provide a comprehensive picture of model performance. This is particularly important when working with imbalanced data where accuracy alone can be misleading. The F1-score, which balances precision and recall, was used as the primary metric for identifying the most effective classifier and resampling technique combination.

1.2 Decision Tree

Decision Tree is a supervised learning algorithm that constructs a tree-like model. It employs a "divide and conquer strategy by conducting a greedy search to identify the optimal split points within a tree. This process of splitting is then repeated in a top-down, recursive manner until all, or the majority of records have been classified under specific class labels" (IBM, n.d.).

In this project, I used 'DecisionTreeClassifier' from 'scikit-learn' library, applying the same resampling techniques and cross-validation approach as with Logistic Regression. However, unlike Logistic Regression, feature scaling was not necessary. This is because Decision Trees are not sensitive to the range of feature values.

Pipelines were constructed in a similar manner, with the only difference being the use of 'DecisionTreeClassifier' instead of 'LogisticRegression', and the removal of the 'StandardScaler' step. Evaluation was again performed using stratified 5-fold cross-validation and the same four classifier evaluation metrics, with the F1-score used as the primary metric for comparison.

1.3 Naive Bayes (Implemented from Scratch)

According to Scikit-learn, "Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable" (Scikit-learn, n.d.). I implemented Naive Bayes from scratch using standard Python and NumPy, without relying on any built-in machine learning libraries.

The training process involved computing the mean, variance, and prior probability for each class. To prevent numerical underflow from multiplying very small values, all probability calculations were done in logarithmic form. For prediction, I implemented a Gaussian probability density function to estimate the likelihood of each feature value under each class. The model determines the log-posterior probability for each class by adding the log of the prior probability to the total log-likelihoods across all features. The class with the highest log-posterior is selected as the predicted label for each instance.

Since this classifier was implemented manually, I could not integrate it into a 'scikit-learn' or 'imblearn' pipeline. Instead, I handled preprocessing, resampling, training, and evaluation through a custom cross-validation loop. The evaluation process is similar to Logistic Regression and Decision Tree. I applied 5-fold stratified cross-validation and tested three resampling strategies (Random Oversampling, Random Undersampling, and SMOTE). Resampling was applied only to the training fold in each split to prevent data leakage.

For every fold and resampling technique, the Naive Bayes algorithm was trained on the resampled data and tested on the validation set. Once again, the same four classifier evaluation metrics were used, with the F1-score used as the primary metric for comparison.

1.4 Resampling Techniques

To address class imbalance (stroke-positive cases constituted only 1.8% of the dataset), three resampling techniques were applied:

1. **Random Oversampling:** "Random oversampling selects existing original samples from the minority class randomly and duplicates them to balance out the dataset" (Galli, 2023). It can lead to overfitting since identical minority class samples are reused.
2. **Random Undersampling:** Reduces the majority class by randomly removing samples to achieve class balance. However, it may result in the loss of valuable information from the majority class (Brownlee, 2021).
3. **SMOTE:** Improves upon oversampling by creating 'new data artificially' (Galli, 2023). Galli (2023) noted that the "main idea behind the SMOTE algorithm is to generate synthetic data points of the minority class by interpolating between the minority class instances" (Galli, 2023).

Methodology

1.1 Exploratory Data Analysis (EDA)

I began with EDA to understand the structure and quality of the dataset. The dataset contained various demographics and health-related features, with the target variable indicating whether a patient had suffered a stroke.

I used descriptive statistics, histograms, bar plots, a correlation matrix, and a pie chart to:

- Explore how each feature may influence stroke occurrence.
- Assess class imbalance.
- Identify necessary preprocessing steps.

The exploration revealed:

- Missing values in certain columns (e.g., 'bmi', 'smoking_status').
- Several numerical and categorical features such as age, BMI, gender, smoking status, and hypertension.
- A highly imbalanced class distribution, with stroke-positive cases constituting only 1.8% of the total dataset.

1.2 Preparing the Dataset

Following EDA, I cleaned and prepared the dataset through the following steps:

- Dropped the 'id' column, as it did not contribute meaningful information to the prediction task.
- Handling missing values by applying mean imputation to the 'bmi' column and replacing missing entries in the 'smoking_status' column with the label 'Unknown'.
- Encoded categorical columns such as 'gender' and 'smoking_status' using label encoding. This was necessary because columns like 'smoking_status' contained multiple categories (e.g., 'Unknown', 'never smoked', 'formerly smoked', 'smoked').
- Split the dataset into training and testing sets using 'train_test_split', with stratification applied to maintain the original class distribution across both sets.
- Applied featured scaling using 'StandardScaler' for models that required it, such as Logistic Regression.

1.3 Model Pipelines

For models using 'scikit-learn' (Logistic Regression and Decision Tree), I created pipelines to structure the training and testing process:

- 'SklearnPipeline' was used for baseline models, combining 'StandardScaler' where necessary with the classifier.
- 'ImblearnPipeline' was used for models that included resampling, combining the resampling step with feature scaling (wh)en required) and the classifier in a single, streamlined workflow.

1.4 Cross-Validation

To obtain a fair and robust estimate of model performance, I used 5-fold stratified cross-validation. This method divides the dataset into 5 subsets (folds), ensuring each fold retains the original class distribution as the complete dataset. I used stratified cross-validation as random splits without stratification could lead to validation sets containing very few or no positive stroke cases. Stratified cross-validation was also useful in reducing overfitting. It ensured the model was evaluated on multiple separate subsets of data, allowing for a more reliable performance score.

1.5 Resampling Techniques

Resampling techniques were a key part of this project due to the dataset's severe class imbalance. The implementation and justification for each technique (Random Oversampling, Random Undersampling, and SMOTE) have already been discussed in earlier sections. Each resampling technique was evaluated across all three classifiers using consistent cross-validation and performance metrics.

1.6 Classifier Evaluation Metrics

The performance for each classifier under different resampling techniques was assessed using four key metrics:

- **Accuracy:** Indicates the overall proportion of correct predictions. It can be misleading in imbalanced datasets.
- **Precision:** Measures the proportion of correctly predicted positive cases.
- **Recall:** Reflects model's ability to identify actual positive instances.
- **F1-Score:** Balances precision and recall. Ensures both false positives and false negatives are taken into account when evaluating performance.

1.7 Code for Custom Naive Bayes Algorithm (Implemented from Scratch)

As previously described in **Section 1.3 of Background**, I implemented a Gaussian Naive Bayes classifier from scratch using standard Python and NumPy.

Algorithm:

```
#train the classifier by computing the mean, variance, and prior for each class
class CustomNaiveBayes:
    #X has input features
    #y has class labels (e.g., 0 or 1)
    def fit(self, X, y):

        #find all unique class labels (0 or 1)
        self.unique_classes = np.unique(y)
        #store mean of each feature for each class
        self.class_means = {}
        #store variance of each feature for each class
        self.class_vars = {}
        #store prior probability for each class
        self.class_prior_probs = {}

        for cls in self.unique_classes:
            #filter rows where label is cls
            samples = X[y == cls]
            #compute and store statistics for this class
            self.class_means[cls] = np.mean(samples, axis=0)
            #added 1e-9 to avoid dividing by 0
            self.class_vars[cls] = np.var(samples, axis=0) + 1e-9
            #proportion of this class
            self.class_prior_probs[cls] = len(samples) / len(y)

    #predict class labels for input
    def predict_class(self, X):
        predictions = [self.predict_class_single(x) for x in X]
        return np.array(predictions)

    #predict class label for a single row in the input
    def predict_class_single(self, x):

        #store posterior probabilities (log) for each class
        probabilities = []

        for cls in self.unique_classes:
            #compute log of prior probability of cls to prevent very small mumbers
            prior_prob_log = np.log(self.class_prior_probs[cls])
            #compute sum of probabilities (log) for all features
            likelihood_log = self.gaussian_log(x, cls).sum()
```

```
            #total probability (log) for this class
            total_prob_log = prior_prob_log + likelihood_log
            probabilities.append(total_prob_log)

        #return the class with the highest probability
        return self.unique_classes[np.argmax(probabilities)]

    #compute the Gaussian density function (log) for each feature
    def gaussian_log(self, x, cls):
        #list of variances, one for each feature
        feature_var = self.class_vars[cls]
        #list of means, one for each feature
        feature_mean = self.class_means[cls]
        #measures how far the value is from the average for that class
        deviation = -((x-feature_mean)**2) / (2*feature_var)
        #shapes the bell curve based on feature_variance (one curve for each feature)
        bell_shape_log = -0.5*np.log(2*np.pi*feature_var)
        return deviation + bell_shape_log
```

Training and Testing the Algorithm:

```
#use StratifiedKFold to keep class distribution similar in each fold
cross_validation = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

#dictionary of resampling strategies including baseline
resamplers = {
    "Baseline": None,
    "Random Oversampling": RandomOverSampler(random_state=42),
    "Random Undersampling": RandomUnderSampler(random_state=42),
    "SMOTE": SMOTE(random_state=42)
}

results = []

#loop through each resampling strategy
for name, resampler in resamplers.items():
    accuracies, precisions, recalls, f1s = [], [], [], []

    #split data into folds
    #each iteration gives train_index, validation_index
    for train_index, validation_index in cross_validation.split(X_train, y_train):
        X_train_fold, X_validation_fold = X_train.iloc[train_index], X_train.iloc[validation_index]
        y_train_fold, y_validation_fold = y_train.iloc[train_index], y_train.iloc[validation_index]

        #apply resampling to training data if needed
        if resampler is not None:
            X_train_fold, y_train_fold = resampler.fit_resample(X_train_fold, y_train_fold)
```

```
        #train classifier and predict
        custom_classifier = CustomNaiveBayes()
        custom_classifier.fit(X_train_fold.values, y_train_fold.values)
        y_predictions = custom_classifier.predict_class(X_validation_fold.values)

        #calculate metrics
        a_s = accuracy_score(y_validation_fold, y_predictions)
        ps = precision_score(y_validation_fold, y_predictions, zero_division=0)
        rs = recall_score(y_validation_fold, y_predictions)
        f1_s = f1_score(y_validation_fold, y_predictions)

        #store results
        accuracies.append(a_s)
        precisions.append(ps)
        recalls.append(rs)
        f1s.append(f1_s)

    #save mean results
    results.append({
        "Resampling Method": name,
        "Accuracy": np.mean(accuracies),
        "Precision": np.mean(precisions),
        "Recall": np.mean(recalls),
        "F1 Score": np.mean(f1s)
    })

df_results = pd.DataFrame(results)
print(df_results)
```

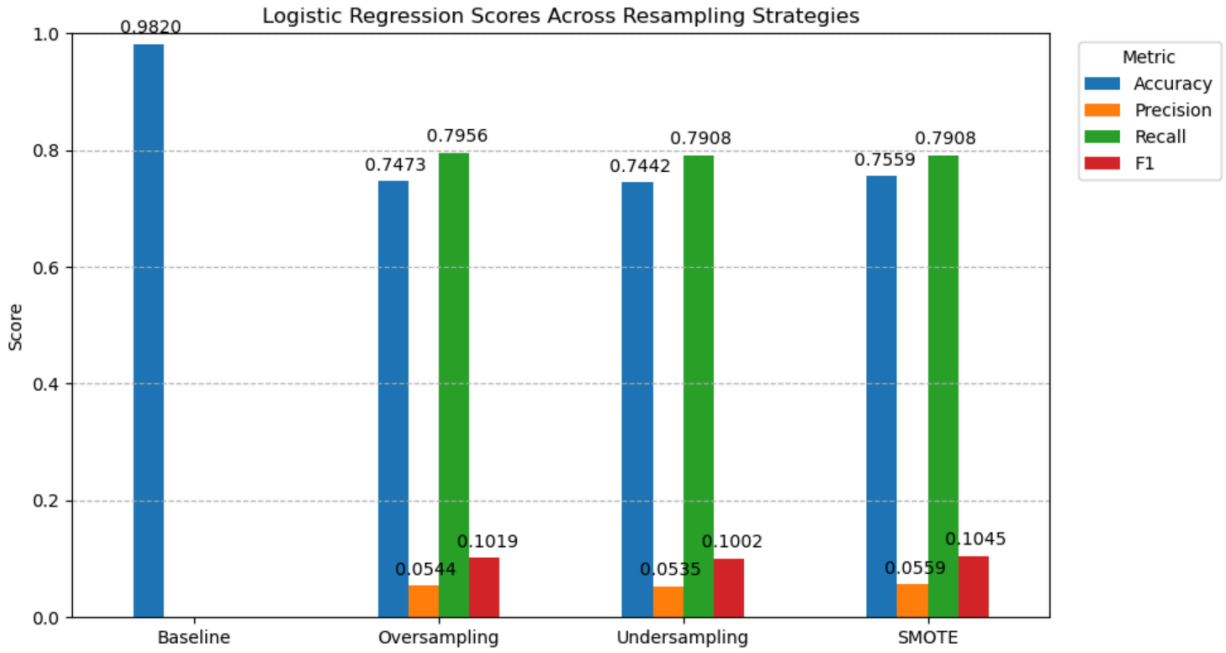
Results

1.1 Results for Logistic Regression and 3 Resampling Techniques

1.1.1 Table

Technique	Accuracy	Precision	Recall	F1-Score
Baseline (No Resampling)	0.9820	0.0000	0.0000	0.0000
Random Oversampling	0.7473	0.0544	0.7956	0.1019
Random Undersampling	0.7442	0.0535	0.7908	0.1002
SMOTE	0.7559	0.0559	0.7908	0.1045 (Best)

1.1.2 Graph



1.1.3 Interpretation

Baseline:

- The baseline model achieved high accuracy but failed to correctly identify minority class cases (stroke = 1).
- This is due to the imbalanced dataset.
- The model predicts mostly the majority class to achieve high accuracy, which is misleading.

Oversampling:

- Accuracy: Dropped to 0.7473 which is expected as the model no longer predicts only the majority class.
- Precision: Low, indicating many false positives when predicting the minority class.
- Recall: High, showing the model is now able to detect most of the actual minority class cases (stroke = 1).
- F1-Score: Improved (0.1019) compared to baseline, a better balance between precision and recall.

Undersampling:

- Accuracy: Slightly lower (0.7442) due to training on a reduced dataset.
- Precision: Low, indicating many false positives when predicting the minority class.
- Recall: High, showing the model is now able to detect most of the actual minority class cases (stroke = 1).
- F1-Score: Lower (0.1002) than oversampling.

SMOTE:

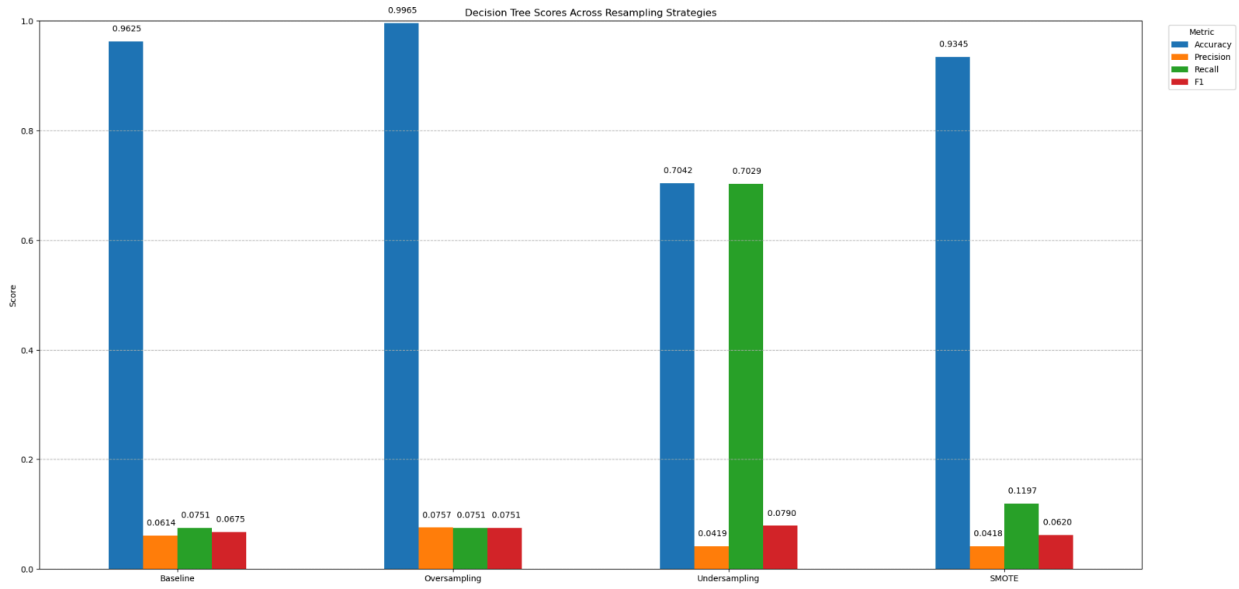
- Accuracy: Slightly better (0.7559) than oversampling and undersampling.
- Precision: Slightly improved (0.0559), indicating fewer false positives than other resampling strategies.
- Recall: High, showing the model is now able to detect most of the actual minority class cases (stroke = 1).
- F1-Score: Highest among all (0.1045), suggesting SMOTE achieved the most balanced performance overall.

2.1 Results for Decision Tree and 3 Resampling Techniques

2.1.1 Table

Technique	Accuracy	Precision	Recall	F1-Score
Baseline (No Resampling)	0.9625	0.0614	0.0751	0.0675
Random Oversampling	0.9665	0.0757	0.0751	0.0751
Random Undersampling	0.7042	0.0419	0.7029	0.0790 (Best)
SMOTE	0.9345	0.0418	0.1197	0.0620

2.1.2 Graph



2.1.3 Interpretation

Baseline:

- The baseline model achieved high accuracy but failed to correctly identify minority class cases (stroke = 1).
- This is due to the imbalanced dataset.
- The model predicts mostly the majority class to achieve high accuracy, which is misleading.
- Although slightly better than zero, precision, recall, and f1-score remain very low.

Oversampling:

- Accuracy: Slightly higher than baseline (0.9965), but still misleading.
- Precision: Low, indicating many false positives when predicting the minority class.
- Recall: Low, suggesting the model struggles to detect minority cases (stroke = 1) despite oversampling.
- F1-Score: Slightly higher (0.0751) compared to baseline, showing some gain in performance.

Undersampling:

- Accuracy: Dropped significantly (0.7042) due to training on a smaller, balanced dataset.
- Precision: Very low, indicating many false positives when predicting the minority class.
- Recall: Much higher, showing the model is now able to detect most of the actual minority class cases (stroke = 1).
- F1-Score: Highest among all (0.0790), suggesting SMOTE achieved the most balanced performance overall.

SMOTE:

- Accuracy: (0.9345) Slightly lower than baseline and oversampling but better than undersampling.
- Precision: Similar (0.0418) to undersampling.
- Recall: (0.1197) Higher than oversampling but lower compared to undersampling.
- F1-Score: Lowest among all (0.0620).

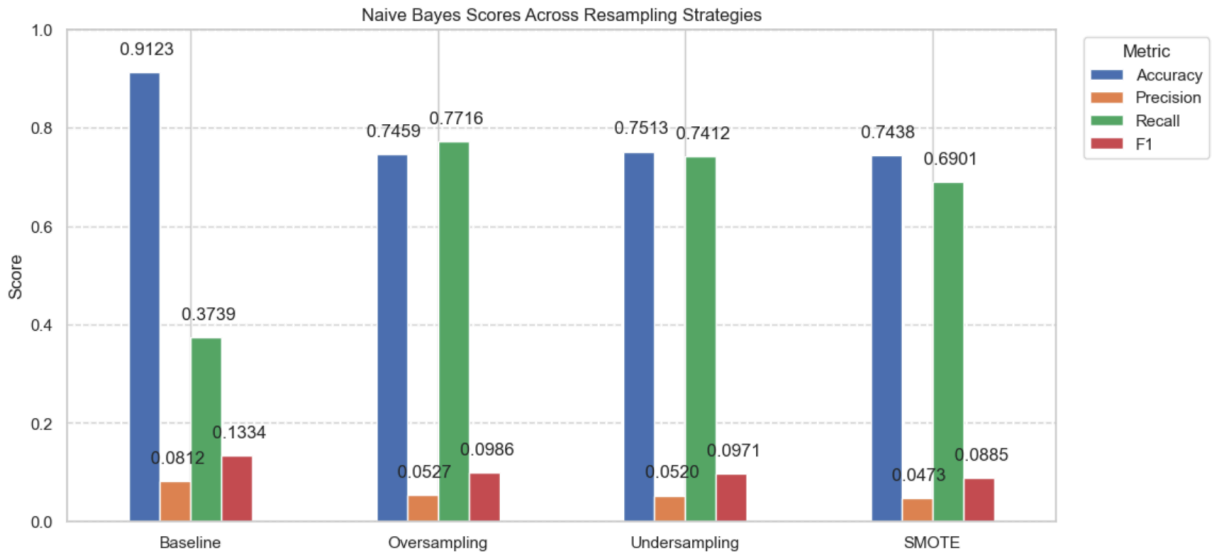
3.1 Results for Naive Bayes and 3 Resampling Techniques

3.1.1 Table

	Resampling Method	Accuracy	Precision	Recall	F1 Score
0	Baseline	0.912298	0.081226	0.373905	0.133400
1	Random Oversampling	0.745881	0.052684	0.771594	0.098630
2	Random Undersampling	0.751267	0.051963	0.741206	0.097110
3	SMOTE	0.743779	0.047298	0.690108	0.088526

Best: Naive Bayes + Random Oversampling (0.098630)

3.1.2 Graph



3.1.3 Interpretation

Baseline:

- The baseline model achieved high accuracy but failed to correctly identify minority class cases (stroke = 1).
- This is due to the imbalanced dataset.
- The model predicts mostly the majority class to achieve high accuracy, which is misleading.
- Although slightly better than zero, precision, recall, and f1-score remain very low.

Oversampling:

- Accuracy: Dropped (0.7459), which is expected as the model no longer predicts only the majority class.
- Precision: Very low, indicating many false positives when predicting the minority class.
- Recall: Improved significantly (0.7716), showing the model is better at identifying actual stroke cases.
- F1-Score: Improved slightly (0.0986) compared to baseline, a better balance between precision and recall.

Undersampling:

- Accuracy: Slightly higher than oversampling (0.7513), but still lower than baseline.
- Precision: Low, indicating many false positives when predicting the minority class.
- Recall: High, showing the model is now able to detect most of the actual minority class cases (stroke = 1).
- F1-Score: Similar (0.0971) to oversampling.

SMOTE:

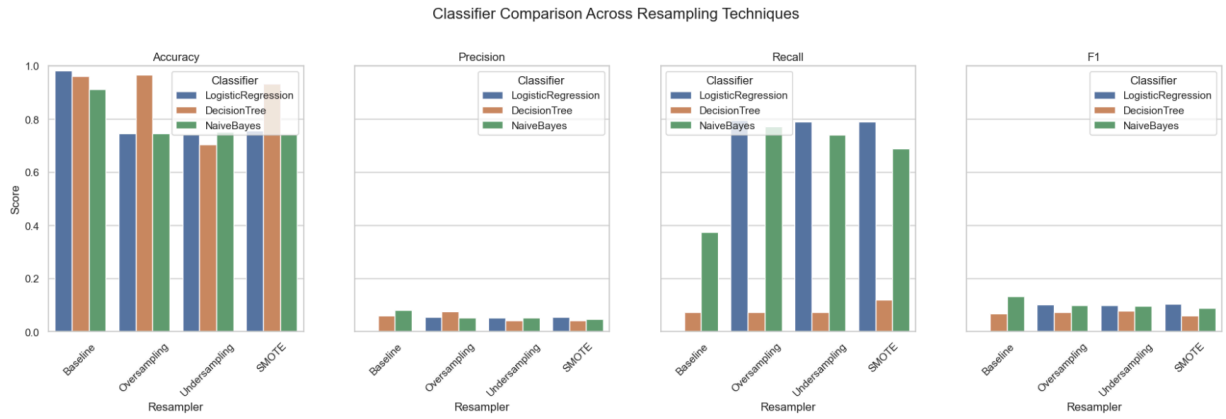
- Accuracy: (0.7439) Slightly lower oversampling and undersampling.
- Precision: Lowest (0.0473), indicating many false positives when predicting the minority class.
- Recall: Reasonably high, showing stroke cases are being detected.
- F1-Score: Lowest among all (0.0885), suggesting SMOTE struggled to improve the overall balance.

4.1 Overall Results for the 3 Classifiers and 3 Resampling Techniques

4.1.1 Table

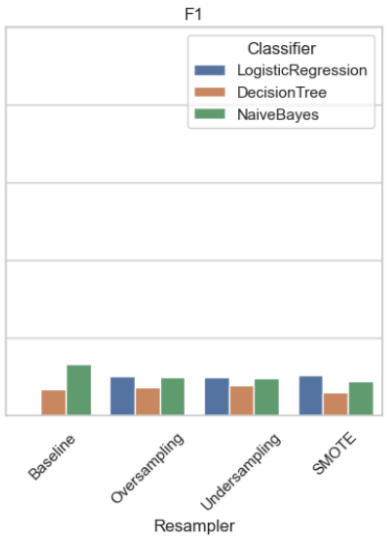
	Classifier	Resampler	Accuracy	Precision	Recall	F1
0	LogisticRegression	Baseline	0.9820	0.0000	0.0000	0.0000
1	LogisticRegression	Oversampling	0.7473	0.0544	0.7956	0.1019
2	LogisticRegression	Undersampling	0.7442	0.0535	0.7908	0.1002
3	LogisticRegression	SMOTE	0.7559	0.0559	0.7908	0.1045
4	DecisionTree	Baseline	0.9625	0.0614	0.0751	0.0675
5	DecisionTree	Oversampling	0.9665	0.0757	0.0751	0.0751
6	DecisionTree	Undersampling	0.7042	0.0419	0.0729	0.0790
7	DecisionTree	SMOTE	0.9345	0.0419	0.1197	0.0620
8	NaiveBayes	Baseline	0.9123	0.0812	0.3739	0.1334
9	NaiveBayes	Oversampling	0.7459	0.0527	0.7716	0.0986
10	NaiveBayes	Undersampling	0.7513	0.0520	0.7412	0.0971
11	NaiveBayes	SMOTE	0.7438	0.0473	0.6901	0.0885

4.1.2 Graph



5.1 Conclusion

Based on the comparison above, Logistic Regression with SMOTE emerges as the best-performing combination. It offers the most balanced performance in terms of F1-Score.



Evaluation

The aim of this project was achieved through a methodical approach that explored a variety of model and resampling combinations.

Strengths

A key strength of this project was the systematic exploration of multiple classifier-resampling combinations to identify the most effective setup for predicting stroke. In a medical context, where early and accurate detection of high-risk patients can be critical, it is essential to evaluate models beyond surface-level accuracy. By integrating three resampling techniques (Random Oversampling, Random Undersampling, and SMOTE), I was able to directly address the class imbalance and observe their effect on performance. This strategy significantly improved model sensitivity to the minority class. For example, Logistic Regression without any resampling yielded an F1-score of 0.0000. It completely failed to identify stroke-positive cases. However, when combined with SMOTE, the F1-score improved to 0.1045. This demonstrates the impact of appropriate resampling.

Another key strength was the clear and modular methodology. I used loops to streamline evaluation logic across all resampling techniques. The use of stratified 5-fold cross-validation helped reduce overfitting and ensure fair model evaluation. Model performance was assessed using four key metrics (Accuracy, Precision, Recall, F1-score), to provide a more complete understanding beyond a simple accuracy.

The successful custom implementation of Naive Bayes using only Python and NumPy is also another strength of this project.

Limitations

A limitation of this project was the consistently low precision across classifiers and resampling techniques, indicating a high number of false positive predictions. This highlights the need for better-calibrated models or alternative techniques that can improve precision without severely lowering recall.

The Decision Tree model also showed signs of overfitting, particularly when used with Random Undersampling. This is due to the reduction of training data from the majority class, which limits the model's ability to learn general patterns.

Furthermore, each resampling technique has its limitations. Random Oversampling can lead to overfitting by duplicating existing minority class samples. Random Undersampling might discard valuable majority class samples. SMOTE creates synthetic examples, but they might not always reflect real-world patterns, especially when classes overlap.

Although the Naive Bayes model worked as expected, it depends on strong assumptions (that all features are independent and normally distributed) which may not be true in this dataset. Thus, limiting how well it performs.

Conclusion

Though the project met its aim, there are several ways it could be improved:

- Hyperparameter tuning (e.g., applying regularisation in Logistic Regression or adjusting tree depth) to improve generalisation.
- Feature selection to reduce noise.
- Exploring ensemble methods like Random Forest, which may offer better performance.
- Using more advanced techniques for handling imbalance, such as class weighting or cost-sensitive learning.

Conclusion

The aim of this project was to identify the most effective combination of classifier and resampling technique for stroke prediction on an imbalance healthcare dataset. This was achieved through the implementation of three classifiers (Logistic Regression, Decision Tree, and a custom Naive Bayes), each tested with baseline, Random Oversampling, Random Undersampling, and SMOTE techniques.

The results demonstrated that resampling techniques, particularly SMOTE, improved model performance on the minority class. For instance, the F1-score for Logistic Regression improved from 0.0000 (baseline) to 0.1045 when SMOTE was applied.

Among all combinations, Logistic Regression with SMOTE provided the most balanced and reliable performance in predicting strokes.

References

Barth, S. (n.d.). *Machine Learning in Healthcare: Guide to Applications & Benefits*. ForeSee Medical. Retrieved June 30, 2025, from https://www.foreseemed.com/blog/machine-learning-in-healthcare

Brownlee, J. (2021, January 5). *Random Oversampling and Undersampling for Imbalanced Classification - MachineLearningMastery.com*. Machine Learning Mastery. Retrieved June 30, 2025, from https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/

Galli, S. (2023, March 20). *Exploring Oversampling Techniques for Imbalanced Datasets*. Train in Data's Blog. Retrieved June 30, 2025, from https://www.blog.trainindata.com/oversampling-techniques-for-imbalanced-data/

Machine learning in healthcare: Uses, benefits and pioneers in the field. (2024, September 18). EIT Health. Retrieved June 30, 2025, from https://eithealth.eu/news-article/machine-learning-in-healthcare-uses-benefits-and-pioneers-in-the-field/

1.9. Naive Bayes — scikit-learn 1.7.0 documentation. (n.d.). Scikit-learn. Retrieved June 30, 2025, from https://scikit-learn.org/stable/modules/naive_bayes.html

Tiwari, S. (2021). *Cerebral Stroke Prediction-Imbalanced Dataset*. Kaggle. https://www.kaggle.com/datasets/shashwatwork/cerebral-stroke-predictionimbalaced-dataset

What is a Decision Tree? (n.d.). IBM. Retrieved June 30, 2025, from https://www.ibm.com/think/topics/decision-trees