Pattern Recognition Assignment: Heart Failure Classification Problem

Saifullah Mousaad 21010651
Ahmed Ashraf 21010040
Ahmed Osama 21010037

March 10, 2025

Contents

1 Methodology

1.1 Dataset

The Heart Failure dataset is used, consisting of 968 samples with 11 features and 2 classes as an indication whether the case is diagnosed with heart failure or not. The dataset is split into 70% training, 10% validation set and 20% Test sets using a random seed of 42 for reproducibility. All models were trained on the training set and evaluated on the test set. Accuracy was computed as the fraction of correctly predicted labels, with F1-Score. The tree structures were visualized and compared.

1.2 Data Preprocessing

In order to use the data properly with our models, a preprocessing step was required to ensure that the dataset is fine and won't affect the performance, actions takes:

- The random seed is fixed along the notebook, the randomness we encountered was due to the usage of random function from TensorFlow, Numpy and normal built-in method. Hence, before any progress the seeds were fixed to 42.
- Distribution of data to be trained, validated and tested was set to 70%, 10% and 20% respectively. Stratifying was necessary to hold the original distribution model for each set.
- Numerate the categorical features as most of the classifiers can't operate but for numerical features.
- Normalizing the features, some classifiers (especially KNN and Logistic Regression) showed huge improvement when features are normalized. The followed approach was the Z-Score normalization.

2 Decision Tree

The custom `DecisionTreeClassifier` is implemented in Python with the following key components:

- **Splitting Criterion**: Entropy $H(y) = -\sum p_i \log_2(p_i)$ and information gain $IG = H(\text{parent}) - \sum_{child} \frac{|child|}{|parent|} H(\text{child})$.
- **Hyperparameters**: Maximum depth (`max_depth=7`) and minimum samples to split (`min_samples_split=10`).
- **Tree Construction**: Recursive splitting based on the best feature and threshold.

A snippet of the splitting logic is shown below:

```
def get_best_split(self, X, y):
    best_split = {'info_gain': float('-inf'), 'threshold': None,
       'feature_index': None}
    for feature_index in range(X.shape[1]):
        thresholds = np.unique(X[:, feature_index])
        for threshold in thresholds:
            left_indices = X[:, feature_index] <= threshold
            right_indices = X[:, feature_index] > threshold
            split_info_gain = self.compute_information_gain(y,
               y[left_indices], y[right_indices])
            if split_info_gain > best_split['info_gain']:
                best_split['info_gain'] = split_info_gain
                best_split['feature_index'] = feature_index
                best_split['threshold'] = threshold
    return best_split
```

3 Bagging (Random Forest)

Bootstrap Aggregation (aka Bagging) is based on sampling subsets say **M** from the same original dataset each subset with the same number of samples as the original with samples replacement and construct **M** models that vote when predict a new sample. Random forest is implemented to represent the Bagging Ensemble with a number of decision trees **T** as subclassifiers that will vote to predict. At this point, the presence of bagging method appears as it reduces the varaince by a factor can reach **1/T** (Not always, depending on the covariance between samples) implying that **T** must be tuned to obtain the model's optimal performance.

3.1 Bootstrapping

As mentioned, we construct **M** sets from the original one each with the same size with samples replacement. The extra randomness is that we choose a only number of features **D** to include in the bootstrapped dataset. Hence, **D** is a hyperparameter that will be tuned to optimize the model performance.

3.2 Forest Structure

The **M** bootstrapped sets are fitted using the normal decision tree classifier (reminder: decision trees usually suffer from overfitting) outputing **M** decision trees forming the "Forest"

3.3 Prediction

Completing our forest construction, when it's time for predicting new samples, each decision tree in our forest contribute to decide whether the sample case is diagnosed with heart failure or not. The Decision scheme is using the majority vote, if **M/2** or more decision trees agreed on a prediction, it's then chosen.

3.4 Optimal Hyperparameters

Tuning with the validation set i.e testing with a subset of values for **D** (3, 5, 7, 11, 13, 15) and another one for **T** (10, 20, 50, 100, 150, 250, 500), the random forest model was optimized the most with 11 features per bootstrapped set and 20 distinct classifiers forming the forest with an accuracy rate of 87% for the test set.

3.5 Observations

- Although Theoretically it's better to set **D** to the square root of the total features, **D = 3** showed a not very good performance indicating that the bias was high and caused some underfitting where it wasn't able to capture the structure of the data.
- Leveling up with 7, The model showed really good performance especially with **T** set to 100.
- Reaching 11, the model operated with its full capability achieving the best performance for the model.

- Putting all the features hadn't seem to be best with a higher error than 7 and 11 indicating the possibility of overfitting.
- Regarding **T** values, in general, smaller and larger values fall in the problem of underfitting and overfitting except for 20 with **D** set to 11. The intermediate values (50 : 250) were much stable.

4 Boosting (AdaBoost)

Adaptive Boosting (aka AdaBoost) is an ensemble learning method that builds a strong classifier by combining multiple weak classifiers iteratively. Unlike Bagging, where models are trained independently, AdaBoost assigns weights to the training samples to focus on misclassified cases and improve prediction accuracy in subsequent iterations.

4.1 Weighting and Weak Learners

Initially, all training samples are assigned equal weights. A weak classifier is trained, and its weighted error is computed. Misclassified samples are given higher weights, making them more influential in the next iteration. This process continues for a specified number of iterations **T**, gradually improving model performance.

4.2 Updating Weights

After training each weak classifier, the sample weights are updated based on the classifier's accuracy. Misclassified samples receive higher weights, while correctly classified ones get lower weights. The model assigns a weight α_t to each classifier, which determines its influence on the final decision.

$$w_t^{(n+1)} = w_t^{(n)} \cdot \exp\left(2\alpha_t \cdot \mathbb{I}(y_i \neq h_t(x_i))\right) \tag{1}$$

where:

- $w_i^{(t)}$ is the weight of sample i at iteration t.
- α_t is the weight assigned to the weak classifier at iteration t, computed as:

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) \tag{2}$$

- ϵ_t is the weighted error of the weak classifier:

$$\epsilon_t = \sum_{i=1}^{N} w_i^{(t)} \mathbb{I}(y_i \neq h_t(x_i)) \tag{3}$$

- $\mathbb{I}(y_i \neq h_t(x_i))$ is an indicator function that is 1 if the weak classifier h_t misclassifies x_i, and 0 otherwise.
- The weights are then normalized to ensure they sum to 1:

$$w_i^{(t+1)} = \frac{w_i^{(t+1)}}{\sum_{j=1}^{N} w_j^{(t+1)}} \tag{4}$$

4.3 Final Prediction

When predicting a new sample, each weak classifier contributes to the final decision based on its assigned weight α_t. The weighted sum of all weak classifiers' predictions is computed, and the class with the highest aggregated score is chosen.

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right) \tag{5}$$

where:

- $H(x)$ is the final strong classifier.
- $h_t(x)$ is the prediction of the t^{th} weak classifier.
- α_t is the weight of the t^{th} weak classifier.
- The function sign$(\cdot)$ determines the final class label based on the weighted sum.

4.4 Optimal Hyperparameters

Using validation tuning, AdaBoost was optimized by selecting a subset of values for the number of weak classifiers **T** (10, 20, 50, 100, 200, 500). The best performance based on validation set data was achieved with $\mathbf{T} = 10$ weak classifiers, resulting in an accuracy of 84% on the test set.

4.5 Observations

- When **T** was too low (3, 10), the model underperformed, showing that too few weak classifiers were insufficient to capture the patterns in the data.
- Increasing **T** improved the model's performance, but very large values (200, 500) led to overfitting and at some point the accuracy will remain the same.
- The best balance was found with $\mathbf{T} = 10$, where the model learned effectively without overfitting.

5 Evaluation

5.1 Comparison of Models

Model	Accuracy	F1 Score	Confusion Matrix
Decision Tree	75.54%	0.79	$\begin{bmatrix} 55 & 27 \\ 18 & 84 \end{bmatrix}$
Bagging (Random Forest)	87.0%	0.88	$\begin{bmatrix} 71 & 11 \\ 13 & 89 \end{bmatrix}$
Boosting (AdaBoost)	83.7%	0.85	$\begin{bmatrix} 72 & 10 \\ 20 & 82 \end{bmatrix}$

Table 1: Comparison of Decision Tree, Bagging, and Boosting Performance Metrics

6 Analysis and Comparison

6.1 Best Model

The final ranking was as follows:

- **Bagging**
  Performed very well with an average accuracy = 87% for the training set and oscillating between 85% to 87% for the test set. The reason behind this is its focus to reduce the variance which drastically limited overfitting from happening.

- Boosting
  With an average training accuracy of 85% and testing accuracy of 84%, boosting did almost well as expected with some limitations due to the weak learners being decision stumps, i.e decision trees with depth = 1. Although, a noticeable improvement from the decision tree due to the idea of limiting the bias through taking the averaged weight of the predictions unlike the normal average of the random forest.

- Decision Trees
  As expected due to its high variance as known achieving a bad accuracy of 75% for the test set reflecting the drastic effect of overfitting.

6.2 Decision Tree Vs. Ensemble

6.2.1 Advantages of Decision Trees Vs. Ensemble

- Simplicity, Decision trees are much simpler than ensemble methods.
- Speed, Decision trees are fast to be trained and for classification.
- Interpretability, Decision trees are much easier for human to understand.

6.2.2 Advantages of Ensemble Vs. Decision Trees

- Performance, ensemble methods have the upper-hand regarding the accuracy from the bias and variance aspects.
- Undependability, ensemble methods clearly care about eliminating or at least limiting the dependence between samples using the randomization techniques.
- Scalability, ensemble methods can handle large datasets obviously better than decision trees.

6.3 Confusion Matrices

- **Decision Tree**
  Although the samples with true target = 0 is less than the set with true target = 1, The Decision Tree Classifier made most of its mistakes in the samples with target = 0 and predicted it with 1 indicating that the limitation of the depth impacted nodes that were supposed to split further and considered it a leaf node with value = target of the majority samples (samples with target 1 as mentioned) dominating the 0 target samples.
- **Random Forest**
  Bagging resulted a very balanced confusion matrix reflecting one of its advantages over the decision trees which is the sampling with replacement, the early dominance of the 1-true target samples is reduced via allowing replacement allowing 0-true target samples to be the majority in some bootstrapped datasets.
- **AdaBoost**
  In contrast with decision trees, with decision stumps as our weak learners, the true class for the majority samples would suffer more from inaccuracy. Decision Stumps split the dataset on a single feature and a single value, hence, the class with more samples is more probable to loose some of his samples due to the oversimplicity in splitting the dataset.

7 Bonus Models

7.1 K-NN

- We manually optimized the number of neighbors **K** and the weight function (uniform vs. distance-based weighting). The values tested for **K** were (3, 5, 7, 11, 15, 21), and we tested both uniform and distance weighting.
- The best-performing configuration was found to be **K** = 3 with distance-based weighting, achieving an accuracy of approximately 89%.

7.2 Logistic Regression

7.2.1 Automatically Tuned SKlearn Class

Scikit-learn Library has a class Containing automatic tuning for C hyperparameter (The inverse of the regularization factor), We Tried with both l1 and l2 regularizer.// l1 achieved better accuracy rate of almost 89%-90% on test data, slightly better than l2 which achieved 88%-89%.

7.2.2 Manually Tuned SKlearn Class

- The other Scikit-learn library class is the one without automatic tuning for C hyperparameter. This time, we tried the model that combine l1 and l2 regularizer each having a participation ratio. Using the validation values, we tried to tune C across the values (0.001, 0.01, 0.05, 0.1, 0.25, 0.5, 0.7, 0.8, 0.9, 1, 5, 10, 100, 1000) and tune the l1 ratio in the regularizer across the values (0, 0.01, 0.05, 0.1, 0.3, 0.4, 0.5, 0.7, 0.9, 1).
- The result was somehow surprising, the best hyperparameters combination among these values were (c, l1 ratio) = (0.001, 0) indicating that the regularizer is only l2.
- To Solve the conflict in comparing the result with that we get from the automatically tuned class (l1 got 89%-90% accuracy rate), it turned out that the sklearn class a complex tuning system dividing the data into subsets and tune every subset k times having the average results as its outputs to be more resilient to overfitting unlike the way our validation set-based tuning works. Hence, it was reasonable to get such a result due to the distinction between the evaluation techniques especially when our tuning achieved 87% accuracy which is also good enough.

7.3 Neural Networks

7.3.1 Simple Feedforward Neural Network (FNN)

We implemented a simple Feedforward Neural Network (FNN) with the following configurations:

- A single hidden layer.
- Binary Cross Entropy (BCE) as the loss function.
- Stochastic Gradient Descent (SGD) for weight optimization.
- The hidden layer utilizes the ReLU activation function.
- The output layer uses the Sigmoid activation function.
- Early stopping was applied to prevent overfitting.
- Fixed initial weights for reproducibility.