AIML231

Assignment One

300601546

Shemaiah Rangitaawa
3-25-2024

Part One | Summary Tables and Classification using SKLearn

Below are the two tables generated using the create_summary_tables function written in the Assignment 1 Jupyter Notebook.

Lowest Mean Errors	Ionosphere	Steelplates	Banknotes
DecisionTreeClassifier	0.116364	0	0.023382
KNeighborsClassifier	0.151818	0.017734	0.001516
LogisticRegression	0.125341	0.000536	0.014869
MLPClassifier	0.099773	0.001133	0.000787
RandomForestClassifier	0.068523	0.014851	0.00828
SVC	0.065682	0	0.001487

Corresponding Hyperparameters	Ionosphere	Steelplates	Banknotes
DecisionTreeClassifier	5	6	7
KNeighborsClassifier	1	1	2
LogisticRegression	2	5	5
MLPClassifier	-5	1	-3
RandomForestClassifier	8	10	8
SVC	rbf	linear	rbf

Performance Overview

- **Ionosphere Dataset:** The SVC Classifier shows the best performance with the lowest mean error of approximately 0.065, followed closely by the Random Forest Classifier with an error rate of about 0.1. The DecisionTreeClassifier and LogisticRegression have moderately higher errors, while the KNeighborsClassifier has the highest error rate for this dataset.
- **Steelplates Dataset:** The DecisionTreeClassifier achieves perfect performance with an error rate of 0.0. The LogisticRegression and MLPClassifier also show exceptionally low error rates, indicating strong fits to the data. The Random Forest Classifier and KNeighborsClassifier, while not achieving perfect scores, still perform with extremely low error rates.
- **Banknotes Dataset:** The MLPClassifier stands out with the lowest mean error, near perfect, followed by the KNeighborsClassifier. The Random Forest Classifier also shows superior performance, while the DecisionTreeClassifier and LogisticRegression exhibit slightly higher error rates but still indicate strong predictive accuracy.

Hyperparameter Sensitivity

The performance of each classifier across different datasets suggests varying degrees of sensitivity to hyperparameter settings. For instance:

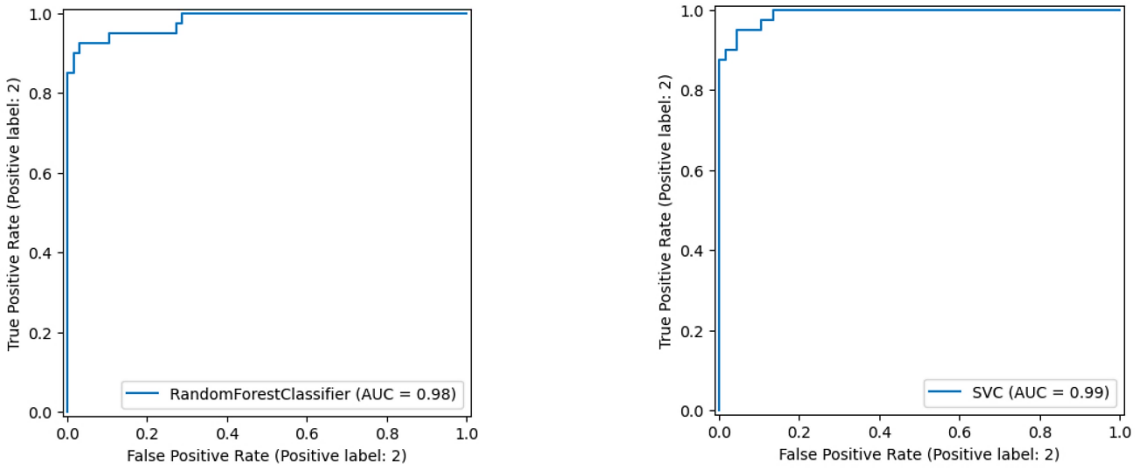
- The **RandomForestClassifier** excels in the ionosphere and banknotes datasets but not as much in the steelplates dataset, suggesting that its performance might be sensitive to the specific characteristics of each dataset and the corresponding hyperparameters.
- The **MLPClassifier** shows consistently robust performance across all datasets, indicating a potentially robust model with respect to different hyperparameter settings, particularly if those settings manage to balance complexity and the risk of overfitting.
- **DecisionTreeClassifier's** perfect performance on the steelplates dataset, yet more moderate performance on the other datasets, suggests that its hyperparameter settings might be highly optimized for specific types of data.
- The performance of the **KNeighborsClassifier** and **LogisticRegression** varies across datasets, which might indicate a need for additional hyperparameter tuning to achieve optimal performance, especially in more complex or varied data environments.

Random Forest and MLP classifiers shine as versatile performers across various data types, often achieving the best results. This adaptability emphasizes the importance of selecting the right model and carefully adjusting hyperparameters based on the specific dataset's characteristics.

However, not all models respond equally to hyperparameter tuning. While models like SVC and MLPClassifier are highly adaptable (evident in the performance plots) and benefit from careful tuning, others like KNeighborsClassifier are more stable and less sensitive to hyperparameter tweaks.

Analyzing SVC and Random Forest Classifier ROC Curves

Analyzing the ROC curves, particularly for Random Forest and Support Vector Classification (SVC), reveals subtle differences in their performance. Here, we see the ROC curves of both classification models trained on 70% of each dataset and assessed on the remaining 30%.



The Random Forest curve (depth 5) sits on the left, while the SVC curve (RBF kernel) is on the right.

Both curves are close to the ideal, with an area under the curve (AUC) near one, indicating a strong fit to the data. The Random Forest curve shows a dip in the true positive rate. This is not necessarily bad, as aiming for 100% accuracy can lead to overfitting, where the model memorizes the training data but performs poorly on unseen data. This slight decrease in true positives suggests Random Forest prioritizes generalizability.

It seems that in machine learning, optimal performance is not about perfect accuracy, but balancing sensitivity (catching true positives) and specificity (avoiding false positives) for robustness. The Random Forest's dip shows a trade-off, favoring a model that works well on both unseen and seen data. Therefore, these 'imperfections' might indicate a well-trained model that prioritizes generalizability.

Part Two | Implementing the K-Nearest Neighbor Classifier

The k-NN classification algorithm implementation is a straightforward and intuitive machine learning technique. By considering the labels of the 'K' closest training instances to a given test point, we can assign the most common label among these neighbors as the prediction. The process involves calculating point to point distances to identify the nearest neighbors, a method well-suited for datasets where point to point distance can accurately reflect similarity.

The accuracy of our k-NN Classifier tends to decrease as K values become larger. In my testing, as K increased beyond a threshold — in this case, around seven — the classifier begins to incorporate information from neighbors that are increasingly distant from the query point. This inclusion dilutes the relevance of the nearest and most similar instances, potentially leading the model to incorporate misleading information when making predictions.

At more extreme values of K, such as K=400, the classifier's decisions are based on a broader, and less relevant, subset of the training data, leading to a loss in specificity for the individual classifications. This results in underfitting, where the model is too simplistic and no longer captures the underlying patterns of the data effectively.

Part Three | Implementation of the Decision Tree Classifier

The Decision Tree classifier builds a predictive model by deriving rules from data features to determine the outcome of a target variable. The decision rules are represented in the form of a tree.

The Algorithm

- **Start with the root node** as the entire dataset.
- **If all data points have the same class** or a stopping criterion is met (e.g., depth limit, minimum data points per node), **stop splitting** and assign the most common class in the node as the node's class.
- **Select the best feature to split on:**
    - Calculate the **Entropy** of the current node.
    - For each feature, calculate the **Information Gain** from splitting on that feature.
    - Choose the feature with the highest information gain.
- **Split the dataset** into subsets using the selected feature. Each subset corresponds to a value of the feature and becomes a child node in the tree.
- **Repeat the process** for each child node. The algorithm is recursive.

Predicted Class Labels and Decision Rational

After training the classifier on the example data set, we can predict the outcome of new, unseen data instances. By feeding these unseen instances into the trained classifier, it will apply the patterns and rules it learned during the training process to make predictions about the outcome. Below is the table of predictions our trained decision tree has made.

Decision Tree Prediction	Outlook	Humidity	Wind
Play Tennis	Overcast	Normal	Weak
Do not Play Tennis	Sunny	High	Strong
Do not Play Tennis	Rain	High	Strong
Play Tennis	Overcast	Normal	Weak

One of the most significant advantages of using a decision tree is its inherent simplicity and interpretability. By following a representation of the decision tree learnt by the classifier, we gain transparent and direct insights into the rationale behind the predictions of our model.

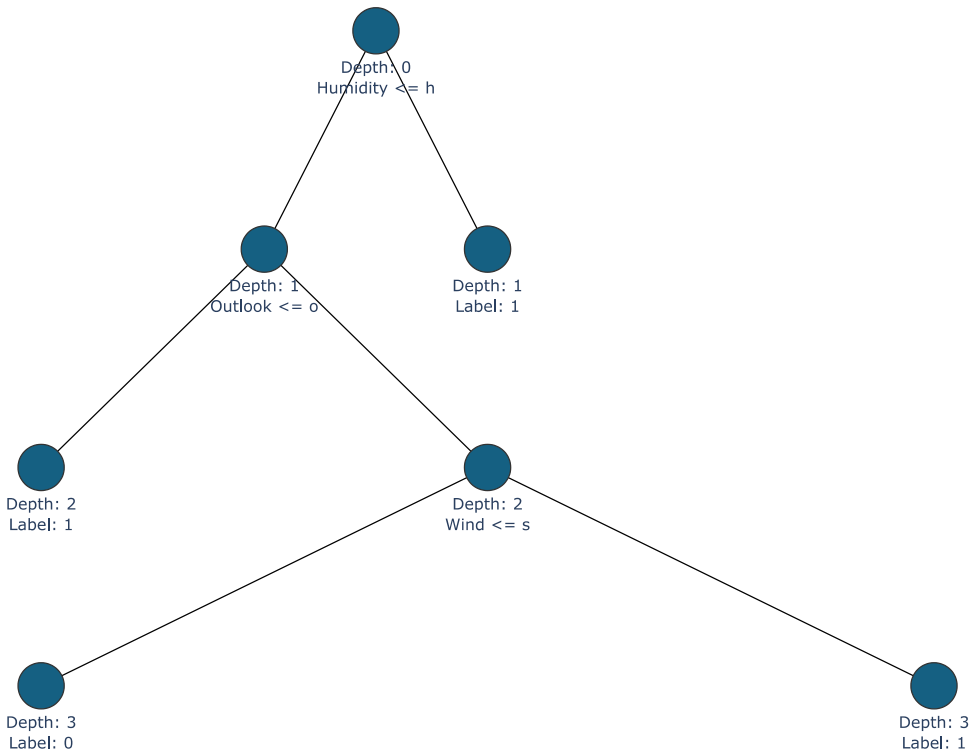


Figure 1: Diagram of the Decision Tree of the Classifier Trained and Tested on the Example Data

Figure 1 presents the decision tree learned by our classifier implementation. Following the structure, we see how the classifier made the prediction of the first instance in the table:

- **Starting at the Root Node:** The root node checks the value of the humidity feature. Since the instance has a humidity value of "Normal" and the decision at the root node is if the humidity is "High" move to the left node, the value is not "High," so we proceed to the child node on the right.
- **Moving to the Right Node (Depth 1):** The node on the right at depth 1 has a label of 1, indicating that the classification decision for this branch is "Play Tennis". There is no further feature to check here because it is a leaf node, and so the decision tree would classify this instance as a 1 for "Play Tennis".

For the second instance:

- **Starting at the Root Node:** Check the value of the humidity feature. Humidity has a value of "High," so we proceed to the left node of the tree.
- **Moving to the Left Node (Depth 1):** The left node is the outlook feature. The decision here is if the outlook is "Overcast" move to the left. The outlook is "Sunny," so we move to the right child node.
- **Moving to the Right Node (Depth 2):** The decision here is if the wind feature has a value of "Strong" move to the right node.
- **Moving to the Left Node (Depth 3):** This a leaf node and assigns the 0 label for "Don't Play Tennis".

Calculating Entropy and Information Gain

Information gain evaluates which feature best splits the data at each node in a decision tree. It is calculated using entropy.

Humidity	Play Tennis
High	1
High	0
High	1
High	1
Normal	1
Normal	1
Normal	1
High	0
Normal	1
Normal	1
Normal	1
High	1
Normal	1
High	0

Table 1: The Humidity Feature and Corresponding Play Tennis Labels

Step One | Entropy of the Dataset

Entropy in the context of machine learning is a metric used to measure the disorder or randomness in data. In a decision tree, the goal is to create pure data subsets, in a way that reduces the overall entropy.

Considering a dataset with n features, its entropy can be calculated with the following formula:

$$H(D) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

where p_i is the is the proportion of instances of class i in the dataset. Inputting our values into the formula, we see the at the entropy of the dataset or specifically the 'Play Tennis' class label is:

$$H(D) = -(p_0 \log_2(p_0) + p_1 \log_2(p_1)) = -\left(\frac{3}{14}\log_2\left(\frac{3}{14}\right) + \frac{11}{14}\log_2\left(\frac{11}{14}\right)\right) \approx 0.750$$

The Humidity feature is at the root node of the constructed decision tree, there were 7 "High" instances and 7 "Normal" instances, there are 14 instances in total. We can immediately see that the dataset is perfectly balanced, this means that entropy here is at its maximum, which is why the algorithm chose this as its root. We can show this mathematically, that given our probabilities, the entropy $H(D_{\text{Humidity}})$ for 'Humidity' is:

$$H(D_{\text{Humidity}}) = -(p_h \log_2(p_h) + p_n \log_2(p_n)) = -\left(\frac{1}{2}\log_2\left(\frac{1}{2}\right) + \frac{1}{2}\log_2\left(\frac{1}{2}\right)\right) = 1$$

The entropy for our Humidity data is 1. The next step is to calculate the entropy after splitting on the feature.

Step Two | Entropy After the Split

For each unique value v of the feature, we partition the dataset into subsets D_v, each containing data for which the feature has value v:

$$H(D_v) = -\sum_{i=1}^{n} p_{i|v} \log_2(p_{i|v})$$

where $p_{i|v}$ is the proportion of class i instances in the subset D_v. In our dataset, "High" humidity corresponds to 4 "Play Tennis" outcomes and 3 "Don't Play Tennis outcomes", "Normal" humidity corresponds to 7 "Play Tennis" outcomes and 0 "Don't Play Tennis" outcomes.

Entropy of High Humidity ('h'):

$$H(D_{\text{High}}) = -\left(p_{0|h}\log_2(p_{0|h}) + p_{1|h}\log_2(p_{0|h})\right)$$

$$H(D_{\text{High}}) = -\left(\frac{4}{7}\log_2\left(\frac{4}{7}\right) + \frac{3}{7}\log_2\left(\frac{3}{7}\right)\right)$$

$$H(D_{\text{High}}) \approx 0.985$$

Entropy of Normal Humidity ('n'):

$$H(D_{\text{Normal}}) = -\left(p_{0|n}\log_2(p_{0|n}) + p_{1|n}\log_2(p_{0|n})\right)$$

$$H(D_{\text{Normal}}) = -(1 \times \log_2(1) + 0 \times \log_2(0))$$

$$H(D_{\text{Normal}}) = -(1 \times \log_2(1) + 0) \;\Big|\; \text{By convention as } \lim_{x\to 0^+} x\log_2(x) = 0$$

$$H(D_{\text{Normal}}) = 0$$

Step Three | Information Gain

$$IG(D, A) = H(D) - \sum_{v \in Values(A)} \left(\frac{|D_v|}{|D|}\right) H(D_v)$$

Where, $IG(D, A)$ is the Information Gain of the dataset D after splitting on feature or attribute A, $H(D)$ is the entropy of the entire dataset before the split, we calculated this in step one.

$Values(A)$ represents the set of all possible values for the attribute A the dataset can be split by. $|D_v|$ is the number of instances in the dataset that have the value v for attribute A. $|D|$ is the total number of instances in the dataset. $H(D_v)$ is the entropy of the subset of D that has the value v for attribute A. Calculating Information Gain for our dataset looks like:

$$IG(D, \text{Humidity}) = H(D) - \left(\frac{|D_{\text{High}}|}{|D_{\text{Humidity}}|} H(D_{\text{High}}) + \frac{|D_{\text{Normal}}|}{|D_{\text{Humidity}}|} H(D_{\text{Normal}})\right)$$

$$IG(D, \text{Humidity}) = 0.750 - \left(\frac{7}{14} \times 0.985 + \frac{7}{14} \times 0\right)$$

$$IG(D, \text{Humidity}) = 0.257$$