TASK

Supervised Learning – Decision Trees

Visit our website

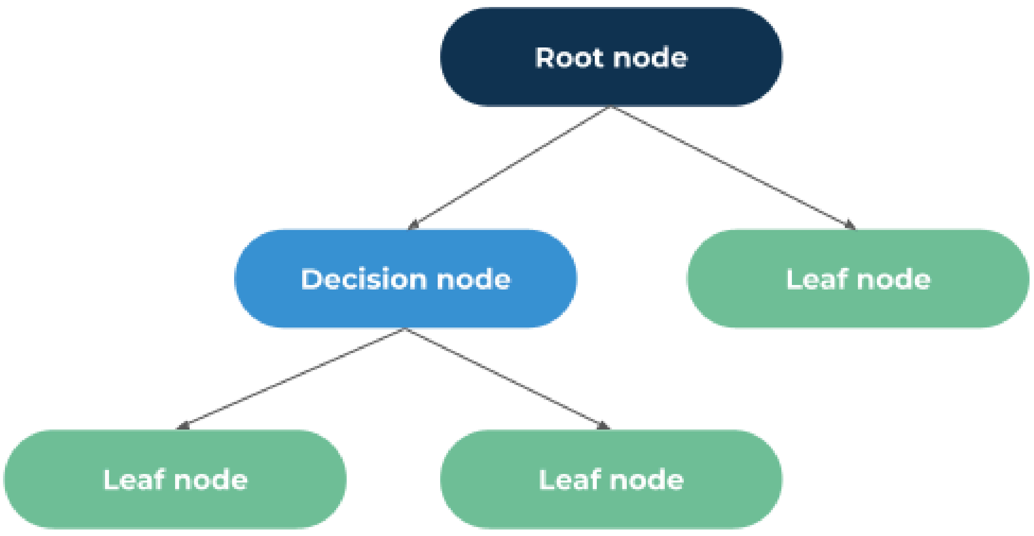
Introduction

In this task, we will describe and go through tree-based methods for regression and classification. Tree-based methods solve problems using a flowchart-like structure that is simple and easy to interpret.

INTRODUCTION TO DECISION TREES

Decision trees are a popular and interpretable model used in machine learning for classification and regression tasks. They function by recursively splitting the data into subsets based on the values of input features, resulting in a tree-like structure composed of various types of nodes.

At the top of this structure is the root node, which represents the entire dataset and the first feature selected for splitting. From the root node, the data is divided into subsets based on specific conditions, creating decision nodes. Each decision node corresponds to a fork in the tree structure, where the data is further partitioned according to the feature's value. As the tree expands, it eventually reaches leaf nodes, which signify the final output or prediction for a given input. Leaf nodes do not contain any further splits and indicate the classification or regression outcome.



Elements of a decision tree

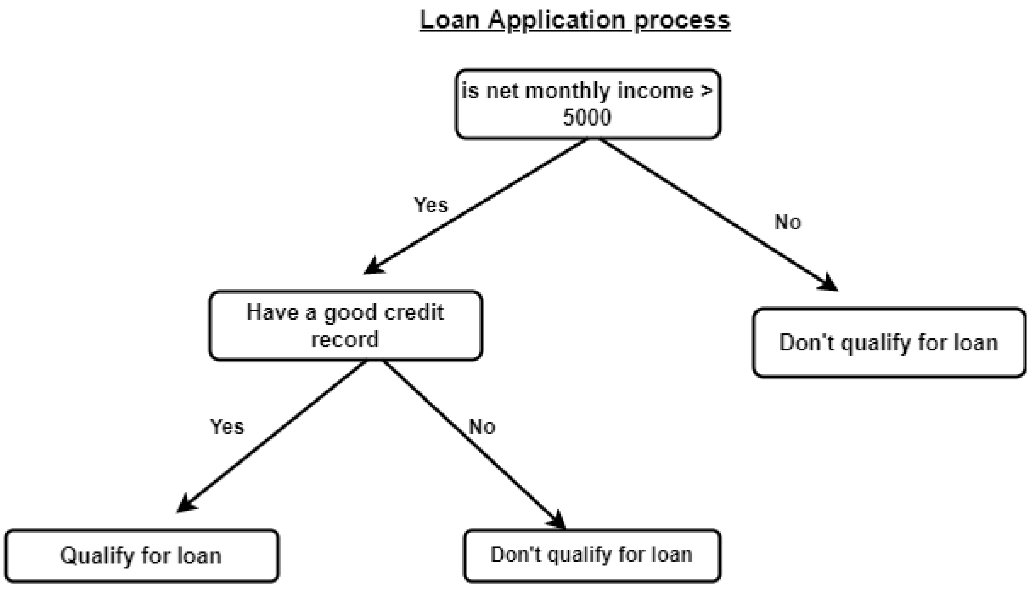
Take note:

Nodes may also be called parent or child nodes. Child nodes are the nodes that branch off from a parent node, and can be either decision nodes or leaf nodes, depending on whether they further split the data or represent final outcomes.

A key concept in building decision trees is entropy, which measures how mixed or uncertain a dataset is. Simply put, entropy shows how jumbled the classes are within a group of data points. When constructing a decision tree, the aim is to select features that provide the most information gain, which means reducing the uncertainty (entropy) after the data is split. By repeatedly choosing the features that reduce this uncertainty the most, the decision tree becomes more accurate and effective at classifying the data.

To further illustrate how a decision tree operates, consider the loan application process. In this scenario, the decision-making journey can be compared to a classification decision tree model. For instance, the first question asked might be, “Is your net monthly income greater than 5000?” If the answer is no, the decision stream ends, and the outcome is “Don't qualify for a loan.” However, if the answer is yes, the next step is to assess whether you “Have a good credit record.” If the credit record is poor, the decision again terminates with “Don't qualify for a loan.” On the other hand, if you do have a good credit record, the final decision is that you “Qualify for a loan.”

This example highlights how a decision tree breaks down the decision-making process, using a series of questions (or features) that reduce uncertainty (entropy) at each step. By selecting features that provide the most information gain, the decision tree can streamline the classification process, as shown in the tree-like structure below:



Decision tree of a loan application process

In all decision trees, you start at the top, following paths that describe the current condition until you reach a final decision. Each point where a decision is made is called a “node”. There are two types of nodes: “decision nodes”, where a choice is made based on a condition (like income level or credit record), and “leaf nodes”, which represent the final outcomes or decisions (such as qualifying or not for a loan). The leaf nodes are at the end of branches, with no further decisions to be made beyond them. This structure helps guide the decision-making process step by step.

Classification trees

Classification trees are a type of decision tree used when the target variable is categorical, meaning the outcome is a distinct class or label. These trees are widely used in classification tasks, where the goal is to predict a category based on input features.

For instance, consider a classification tree built from the Titanic dataset. The objective is to predict whether a passenger survived or not based on features such as gender, age, and ticket class. In a classification tree, each decision node represents a test on a feature, and the dataset is split based on the values of that feature. For example, one of the initial splits could be based on the feature "gender", dividing passengers into "male" and "female" groups.

To decide which feature to split the data on, the decision tree algorithm evaluates the "purity" of each potential split using metrics like Gini Impurity and Entropy. Both of these metrics help measure how well a feature separates the data into homogeneous groups (where the outcome is mostly one class).

- **Gini Impurity** is a measure of how likely it is that a randomly chosen instance would be misclassified if it were labelled according to the current split. In the Titanic example, if you were to split passengers by "gender", Gini Impurity would calculate how mixed the "survived" and "did not survive" outcomes are within each subgroup.
- **Entropy**, on the other hand, measures the level of disorder or uncertainty within a group. A perfect split, where all passengers in one branch survived and all in the other did not, would have an entropy of 0, indicating no uncertainty. The decision tree aims to choose splits that reduce entropy the most, maximising information gain.

In a classification tree, the algorithm selects the feature that results in the highest reduction in impurity (either through Gini or Entropy) at each decision node. This process continues, with the data being split further and further until the tree reaches its leaf nodes. The leaf nodes represent the final predictions – in the Titanic example, whether a passenger is predicted to have survived or not based on their characteristics.

By using Gini Impurity or Entropy to evaluate splits, classification trees ensure that the chosen features lead to increasingly pure subgroups, improving the tree's ability to make accurate predictions.

We recommend that you watch this **video** to see how the logic works when using classification trees.

Regression trees

Where classification trees are used for categorical outcomes, regression trees are utilised for problems where the target variable is numerical. Instead of predicting a category, a regression decision tree aims to predict a continuous value, such as house prices or temperatures.

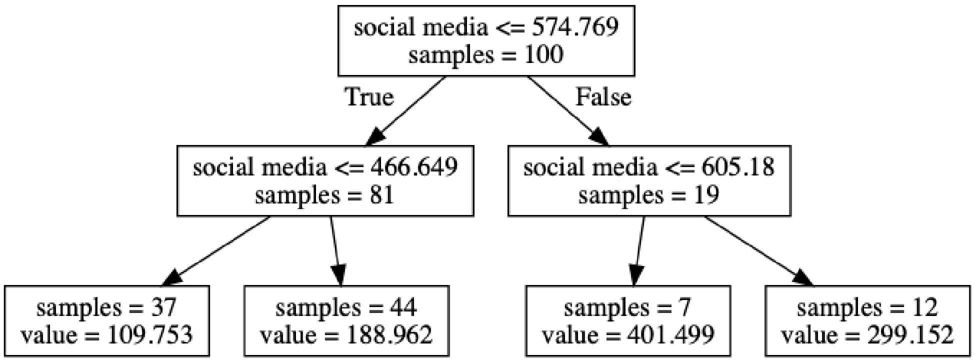
In a regression tree, the data is split into regions based on input features, similar to how a classification tree divides data. However, instead of aiming to classify instances into categories, the goal is to predict a number by finding the best fit for the data. At each decision node, the tree splits the data based on the feature that minimises prediction error, often measured using metrics such as mean squared error (MSE) or variance. This means that the target values in the subsets should be as close to each other as possible. This process continues until the tree reaches its leaf nodes, where the final prediction for that region is calculated.

For example, if we were using a regression tree to predict house prices, the tree might first split the data based on “number of bedrooms” and then further divide it based on “square footage”. The final leaf nodes would contain the predicted price for houses within those feature ranges.

In a regression decision tree, the goal is for the predicted values to be as close as possible to the actual target values. This ensures that the model accurately represents the relationship between the input features and the target variable. The closer the predicted values are to the actual values, the lower the error, which improves the model's performance. While it's helpful if the target values within a node are closer together (low variance), the ultimate aim is for the predictions made by the model to closely match the actual outcomes.

In summary, while classification trees assign categories based on input features, regression trees assign numerical predictions by creating splits that reduce prediction error. Both types of decision trees rely on a similar structure of nodes and leaves but differ in how they handle the target variable.

Consider the following diagram:



Decision tree for social media budget

To interpret these diagrams, imagine a value for a social media budget of, say, 400. Since this value is below 574.769 and below 466.649, the model predicts that 109.753 items will be sold with this budget. This prediction is based on 37 samples in the data – you can follow the branches to the leaf node that shows this on the left-hand side of the above image.

OVERFITTING AND UNDERFITTING

Now that we've explored decision trees for both classification and regression tasks, it's important to address a common challenge they face: overfitting. Overfitting occurs when a model becomes too complex, capturing not just the genuine patterns in the data but also noise or random fluctuations. This often happens with flexible models, like decision trees, which can fit the training data perfectly but struggle to generalise to new, unseen data.

Conversely, underfitting is the result of a model that is too simple, failing to capture the underlying relationships within the data. This can happen when a tree is too shallow or when the model is overly constrained. Both overfitting and underfitting lead to poor performance in machine learning, making it crucial to strike a balance between the model's complexity and its ability to generalise.

Underfitting

As mentioned above, underfitting occurs when a model is too simple to capture the patterns in the data, leading to poor performance on both the training data and new data. The model struggles to learn and make accurate predictions, resulting in a **high training error**. This often happens because the model isn't complex enough, but underfitting can also be caused by other factors, such as poor feature selection (where the input data doesn't explain the outcome well), insufficient training data, or errors within the data itself. For example, using a basic linear model to predict a non-linear relationship in the data would likely result in underfitting, as the model wouldn't be able to capture the complexity needed for accurate predictions.

Overfitting

Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise and specific details that do not generalise to new data. This results in **low training error** but poor performance on test data. A common misconception is that low training error guarantees good test performance. However, an overfitted model becomes too complex and too focused on the training dataset, leading to rules that apply only to that specific data.

For example, imagine building a model to predict house prices. If the model becomes overfitted, it might pay too much attention to certain unique features in the training data, such as an unusual house design or a rare type of material used. While these features might explain price differences in the training set, they may not be relevant when predicting prices for other houses. As a result, the model would fail to generalise effectively when applied to new data.

This lack of generalisation means that, while the model performs excellently on the training set, it performs poorly when tested on new data, which is crucial for solving real-world problems. A well-performing model should strike a balance between capturing the important patterns in the data without being too focused on the specific details of the training set.

The following tables summarise possible causes and solutions for overfitting and underfitting in machine learning models, as well as the techniques used to implement the solutions. Some of the techniques will be covered later in the course while others you can explore as you become more skilled in building machine learning models.

Table 1: Overfitting

Possible Causes	Possible Solution	Techniques
Excessively complex model	Simplify the model	Use a less complex model
Too many features compared to the number of observations	L1 or L2 regularisation	Monitor validation performance during training
Insufficient training data	Cross-validation	Use k-fold cross-validation
High variance in the model	Data augmentation	Generate new data points through transformation
Overly complex model	Dropout (for neural networks)	Randomly drop units during training
Lack of pruning (decision trees)	Pruning (for decision trees)	Remove unimportant branches
Lack of pruning (decision trees)	Complexity pruning	Simplify the decision tree by removing complex branches

Table 2: Underfitting

Possible Causes	Possible Solution	Techniques
Too simple a model	Increase model complexity	Use a more complex model
Lack of relevant features	Feature engineering	Create new features to capture patterns
Insufficient training time	Add more features	Include additional relevant features
Poor data quality	Increase training time	Allow more epochs or iterations
Incorrect hyperparameter settings	Improve data quality	Clean and preprocess the data
Inadequate model configuration	Tune hyperparameters	Adjust hyperparameters for optimal model performance
Lack of diversity in model selection	Use ensemble methods	Combine multiple models for improved performance

Extra resource

Deepen your understanding of model performance by exploring the relationship between overfitting, underfitting, and the **bias–variance tradeoff**.

PREVENTING OVERFITTING IN TREES

Decision trees are particularly susceptible to overfitting as they will continue to split the data until there is a single target value per leaf node if allowed to continue without restraints. When the model is used on unseen data it's likely that the data does not fit into any of the unique leaf nodes.

Preventing overfitting in tree-based methods is important to ensure that the model generalises well to new, unseen data. Overfitting occurs when the model becomes too complex and captures noise in the training data, reducing its ability to perform well on test data. There are several methods to prevent overfitting in tree-based methods that are summarised in the table below.

Table 3: Methods of preventing overfitting

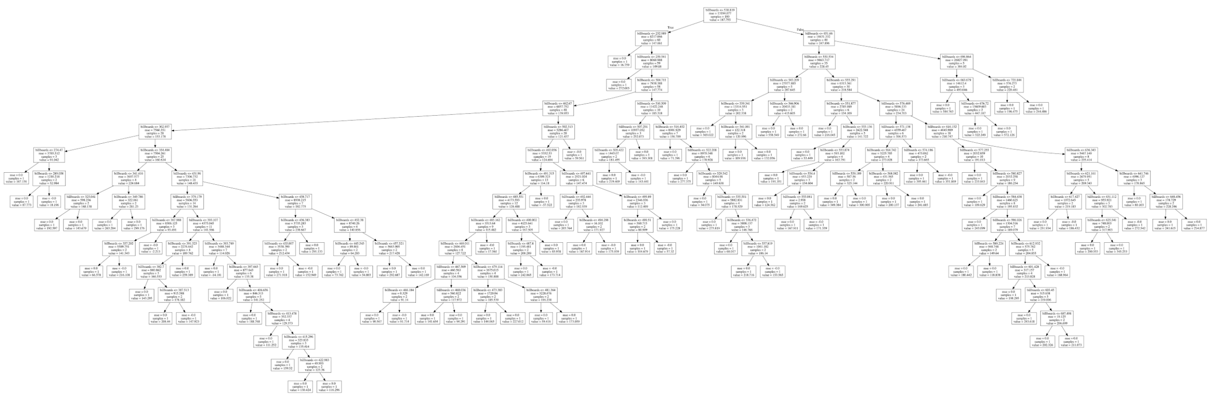
Method	Description	When/How to Identify
Setting a Max Depth	Limits the maximum depth of the tree to avoid capturing noise and becoming overly complex.	Use when you notice the tree is growing too deep, leading to overfitting and capturing noise in the data.
Minimum Samples Split	Sets a minimum threshold for the number of samples required to split a node, preventing the model from generating overly specific branches.	Apply when the model is creating overly specific branches, especially when there are very few samples in certain splits.
Minimum Samples Leaf	Specifies the minimum number of samples required in a leaf node, ensuring the leaves are not too small, which would lead to overfitting.	Use when leaves contain very few samples, indicating overfitting to specific data points.
Maximum Features	Limits the number of features considered for each split in the tree, which reduces model complexity and mitigates overfitting.	Apply when overfitting is caused by too many features, leading to splits that are too specific to the training data.
Reduced Error Pruning	Trims unimportant branches to simplify the tree and improve generalisation.	Apply when the tree is too large and complex after training, and pruning improves performance on validation data.

Cost Complexity Pruning	Adds a cost to the number of splits in the tree, balancing between accuracy and simplicity by penalising overly complex trees.	Apply when the model is overfitting (high variance) and you want to reduce model complexity while retaining predictive performance.
Cross-Validation	Splits the dataset into multiple folds to train and evaluate the model's performance on different subsets, ensuring better generalisation to unseen data.	Use cross-validation when tuning hyperparameters or comparing models to ensure the model generalises well to new data.
Ensemble Methods (Random Forests)	Uses multiple decision trees and averages their predictions to reduce the chances of overfitting by avoiding reliance on a single complex tree.	Use when you want to improve model stability and performance by combining predictions from multiple trees, reducing overfitting.

These methods help to create more generalised models that perform well not only on the training data but also on unseen data. We'll discuss pruning methods here and other methods such as cross-validation and ensemble methods in a future task.

Pruning

Pruning is a technique used to reduce the size of a decision tree by removing branches that do not contribute much to the model's accuracy. This helps prevent the model from overfitting the training data. For example, after growing a large tree, you might find that some branches only improve the accuracy by a small margin but add too much complexity. Pruning removes these branches to simplify the model and improve its ability to generalise to unseen data.



An overfitted decision tree

Pruning can be done in two main ways: pre-pruning (early stopping) and post-pruning (pruning after the tree is fully grown).

For the purposes of this lesson, we will focus on the following specific methods:

1. Post-pruning
    a. Reduced error pruning
    b. Cost complexity pruning
2. Pre-pruning
    a. Setting stopping criteria

Post-pruning

Post-pruning is applied after the tree has been fully grown, where the tree may initially overfit the training data with hundreds of branches. Post-pruning works by evaluating the performance of the tree on a validation/test set and removing parts that do not significantly improve accuracy.

There are several ways this can be done, and we will discuss two methods briefly:

Reduced error pruning

After the tree is grown, each node is evaluated based on its impact on accuracy. If removing a node (and its branches) does not decrease the accuracy on a validation set (or only decreases it slightly), the node is pruned.

The basic steps are as follows:

1. **Grow the full tree:** Build the tree without any early stopping criteria to let it overfit the data.
2. **Use a validation/test set:** Evaluate the tree's performance on a validation set.
3. **Prune the tree:** Iteratively remove nodes that do not improve performance on the validation set. Start from the leaves and work upwards.

The final pruned tree is smaller and generalises better to unseen data.

Cost complexity pruning

Cost complexity pruning is a common method used in decision trees to avoid overfitting. Here, the tree is penalised for being too complex by adding a cost for every additional split. This helps to balance the trade-off between accuracy and simplicity. In practice, we can create a large tree without any restrictions and then apply cost complexity pruning to retain only the most important splits and avoid overfitting.

Let’s have a look at how cost complexity pruning is implemented. This technique adds a penalty term for the complexity of the tree, often represented by the number of nodes or splits. The objective is to find a balance between a small tree and a highly accurate one.

The implementation steps are as follows:

1. A large, fully grown tree is created.
2. Each split in the tree is associated with a cost related to its complexity.
3. Pruning is performed by calculating an optimal “cost complexity parameter” $\alpha\alpha$, which controls the trade-off between complexity and accuracy.
4. Nodes are pruned if their contribution to reducing the error isn't large enough to justify their complexity.

For example, in sklearn's **`DecisionTreeClassifier`**, `ccp_alpha` can be tuned to control the level of pruning. Larger values of `ccp_alpha` result in more aggressive pruning.

Extra resource

Explore the scikit-learn documentation to learn more about **cost complexity pruning using scikit-learn**.

Pre-pruning (early stopping)

Pre-pruning involves halting the growth of the decision tree before it reaches full complexity. This is done by setting specific stopping criteria during tree construction to avoid overfitting. The common pre-pruning criteria include:

- **Maximum depth:** Limiting how deep the tree can grow. A shallower tree will have fewer branches and will be less likely to fit noise in the training data. For example, setting `max_depth=5` will prevent the tree from going beyond five levels of splits.

- **Minimum samples split:** A threshold for the minimum number of samples required to split an internal node. This prevents the model from splitting nodes that only have a few samples, which may result in overfitting. For example, if `min_samples_split=10`, the tree will only split a node if there are at least 10 samples available at that node.

- **Minimum samples per leaf:** Ensures that a leaf node (final node) has at least a minimum number of samples. This avoids creating very small, specific branches, which could overfit the model. For example, setting `min_samples_leaf=5` ensures each leaf node has at least five samples.

By setting these constraints, the tree stops growing before it becomes too complex.

Pre-pruning with `max_depth` example

In the example provided, the tree is constrained with a maximum depth of 3, making it simpler and less likely to overfit, as it will not grow to capture every small detail in the data. This demonstrates how setting `max_depth` directly relates to preventing overfitting by controlling the tree's complexity right from the start of the training process.

```python
from sklearn.tree import DecisionTreeClassifier

# Create a classifier with a low max_depth
tree_low_depth = DecisionTreeClassifier(max_depth=3)

# Train the classifier on your training data
tree_low_depth.fit(X_train, y_train)

# Make predictions on the test data
predictions_low_depth = tree_low_depth.predict(X_test)
```

In the example below, the decision tree classifier is created with a high `max_depth` of 10. This means that the resulting tree will have a maximum depth of 10 levels, making it relatively deep and intricate.

```python
from sklearn.tree import DecisionTreeClassifier

# Create a classifier with a high max_depth
tree_high_depth = DecisionTreeClassifier(max_depth=10)

# Train the classifier on your training data
tree_high_depth.fit(X_train, y_train)

# Make predictions on the test data
predictions_high_depth = tree_high_depth.predict(X_test)
```

In summary:

- Pre-pruning prevents the tree from growing unnecessarily complex during training by setting thresholds on depth, number of samples, and other parameters.
- Post-pruning simplifies an already-grown tree by removing unnecessary nodes based on their impact on accuracy. Techniques like reduced error pruning and cost complexity pruning are commonly used for this purpose.

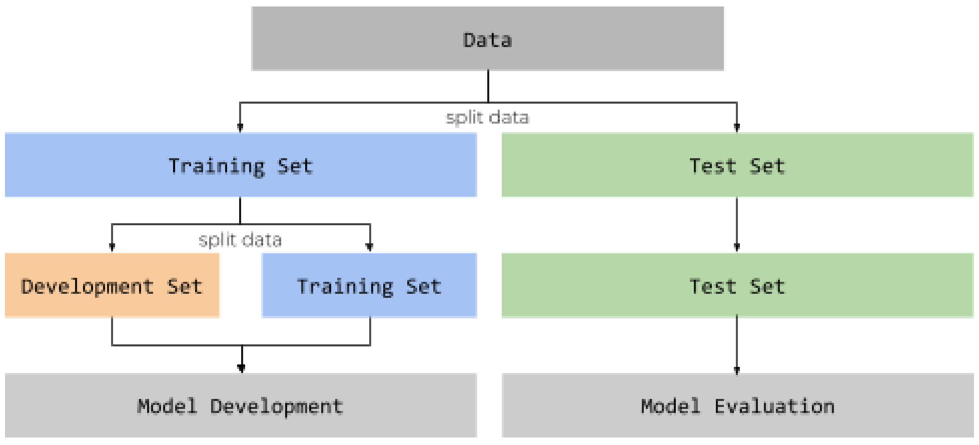
By applying pruning, we reduce the complexity of the decision tree and tree-based methods, thereby improving its generalisation ability and reducing the risk of overfitting.

OPTIMISING DECISION TREE PRUNING

To find the best way to prune a decision tree, we can test different subtrees of an unpruned tree and check their prediction errors. The subtree with the lowest error will be chosen. However, if we use the test data to make these decisions, we risk fitting the **pruning parameters**, which refers to the hyperparameters that control how the tree is pruned, such as the maximum depth, minimum samples required to split a node, or the cost complexity pruning parameter. Hyperparameters are settings that govern the model's structure and behaviour during training but are not learned from the data itself. These include factors like how deep the tree can grow, the minimum number of samples needed to make a split, or how much complexity to penalise through pruning. If we tune these hyperparameters using the test data, the model might overfit, learning specific patterns that do not apply to new data.

Fitting the pruning parameters involves adjusting how much we simplify the decision tree to strike a balance between accuracy and generalisation. This means selecting settings such as how deep the tree can grow or the minimum number of samples needed to make a split. If we tune these parameters using test data, the model might overfit, meaning it learns specific patterns that do not apply to new data.

To avoid this problem, it's helpful to divide the dataset into three parts: a training set, a development set, and a test set. The development set, also known as a validation set, is used to adjust model settings, including pruning parameters. By checking the performance of different pruning methods on the development set, we can keep the test set as a true measure of how well the model will perform on new data. This approach helps create more accurate and reliable machine learning models, ensuring they generalise well to unseen situations.



Take note:

The terms "development set" and "validation set" are often used interchangeably in machine learning. Although their usage can vary depending on the context, both terms generally refer to a subset of the data that is used for model development and tuning.

For this task, we will use the term "`development` set" to refer to the subset of data that is used for model development and tuning.

Here's an example of how to create a development set and split the training set into a separate training and test set using sklearn:

```
# Split the original dataset into training and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Split the training set further into training and development sets
X_train, X_dev, y_train, y_dev = train_test_split(X_train_full,
y_train_full, test_size=0.2, random_state=42)
```

In this example, `X` represents the feature data, and `y` represents the corresponding labels. We first use `train_test_split` to split the original dataset into a training set (`X_train_full` and `y_train_full`) and a test set (`X_test` and `y_test`). Then, we use `train_test_split` again on the training set to further split it into a training set (`X_train` and `y_train`) and a development set (`X_dev` and `y_dev`).

By utilising a development set in addition to the training and test sets, we can make more informed decisions about pruning the decision tree and mitigate the risk of overfitting to the test data.

Extra resource

Check out this informative **blog post** to learn more about the differences between the train, validation (or development), and test datasets.

ADVANTAGES AND DISADVANTAGES

Decision trees have several advantages that make them popular in machine learning. They are easy to interpret and visualise, making it simple to understand how the model is making decisions. They can handle both numerical and categorical data and require minimal data pre-processing. Additionally, decision trees are non-parametric, meaning they don't make assumptions about the distribution of the data.

However, decision trees also have some disadvantages. They can easily overfit the training data, especially when the tree grows deep and becomes too complex. This leads to poor generalisation of new data. Trees are also sensitive to small changes in the data, meaning that slight variations can result in significantly different tree structures. Furthermore, decision trees can be biased towards features with more levels or categories, which can affect their accuracy.

These shortcomings can be addressed through ensemble methods, which combine multiple models to improve performance. One such method is random forest, which builds multiple decision trees and averages their predictions to reduce variance and overfitting. Random forests are more robust than single decision trees, as they mitigate the instability and overfitting issues by introducing randomness in the tree-building process. In the next lesson, we will explore random forests and other ensemble methods in more detail, focusing on how they address the limitations of decision trees and enhance model performance.

Instructions

Run the **Decision_Trees.ipynb** provided with this task for an example of how to implement a decision tree.

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select "Request Review", the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you've done that, feel free to progress to the next task.

Auto-graded task

Create a decision tree that can predict the survival of passengers on the Titanic.

1. Make a copy of the **decision_tree_titanic.ipynb** file provided, which has the **titanic.csv** dataset loaded (sourced **here**) to complete the task. Rename the file **decision_tree_titanic_task.ipynb**.
2. Clean and preprocess the data as needed. Remember to be mindful of the **dummy variable trap** if you encode any categorical features.
3. Select relevant variables from the data and split the data into a training, development, and test set.
4. Using the training set, build and train a decision tree classifier without imposing any restrictions on its depth. Once the model is trained, plot the tree diagram to visualise how it makes decisions.
5. Compute and report the accuracy of your model on the **development set**.
6. Train your decision tree model using different values of `max_depth` from 2 to 10. For each value of `max_depth`, visualise the decision tree and record the accuracy for both the training and development datasets.
7. Plot a line graph comparing your training accuracies with your development accuracies on the same graph. Then, analyse the shapes of

the lines and explain what these shapes indicate about your model's performance.

8. Based on the analysis of `max_depth`, select the optimal value for the final model. Train this final model using the training set and report its accuracy on the **test set**.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.