Hyperiondev

TASK

Supervised Learning – Decision Trees

Visit our website

Introduction

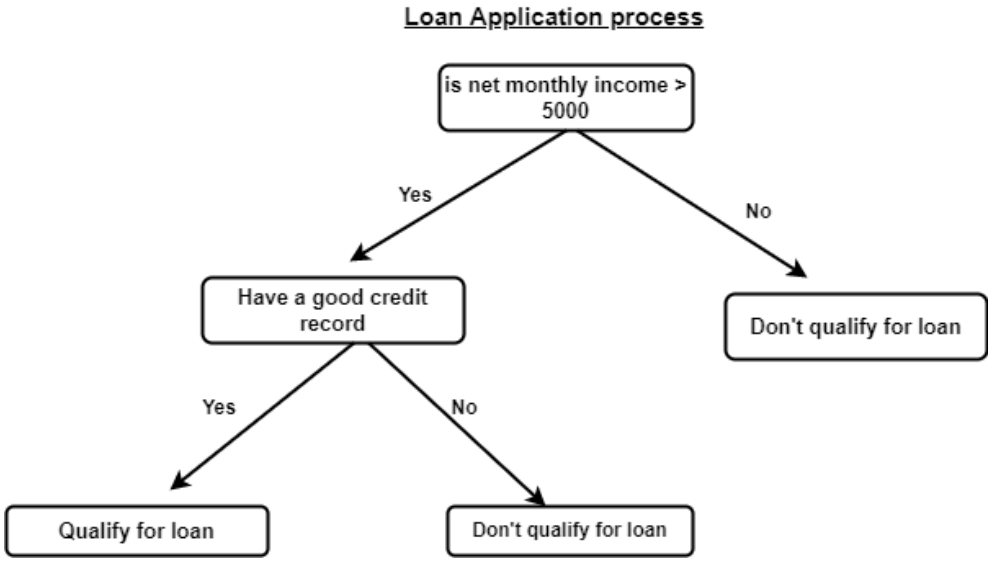
WELCOME TO THE SUPERVISED LEARNING – DECISION TREES TASK!

In this task, we describe 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 work by formulating simple rules that partition data into ever-smaller regions. Each partitioning is like a fork in the road, where a decision must be made. The decision is made based on *rules* that are derived from previous experiences. Decision trees are among the most interpretable machine learning techniques because they resemble the way humans make decisions.

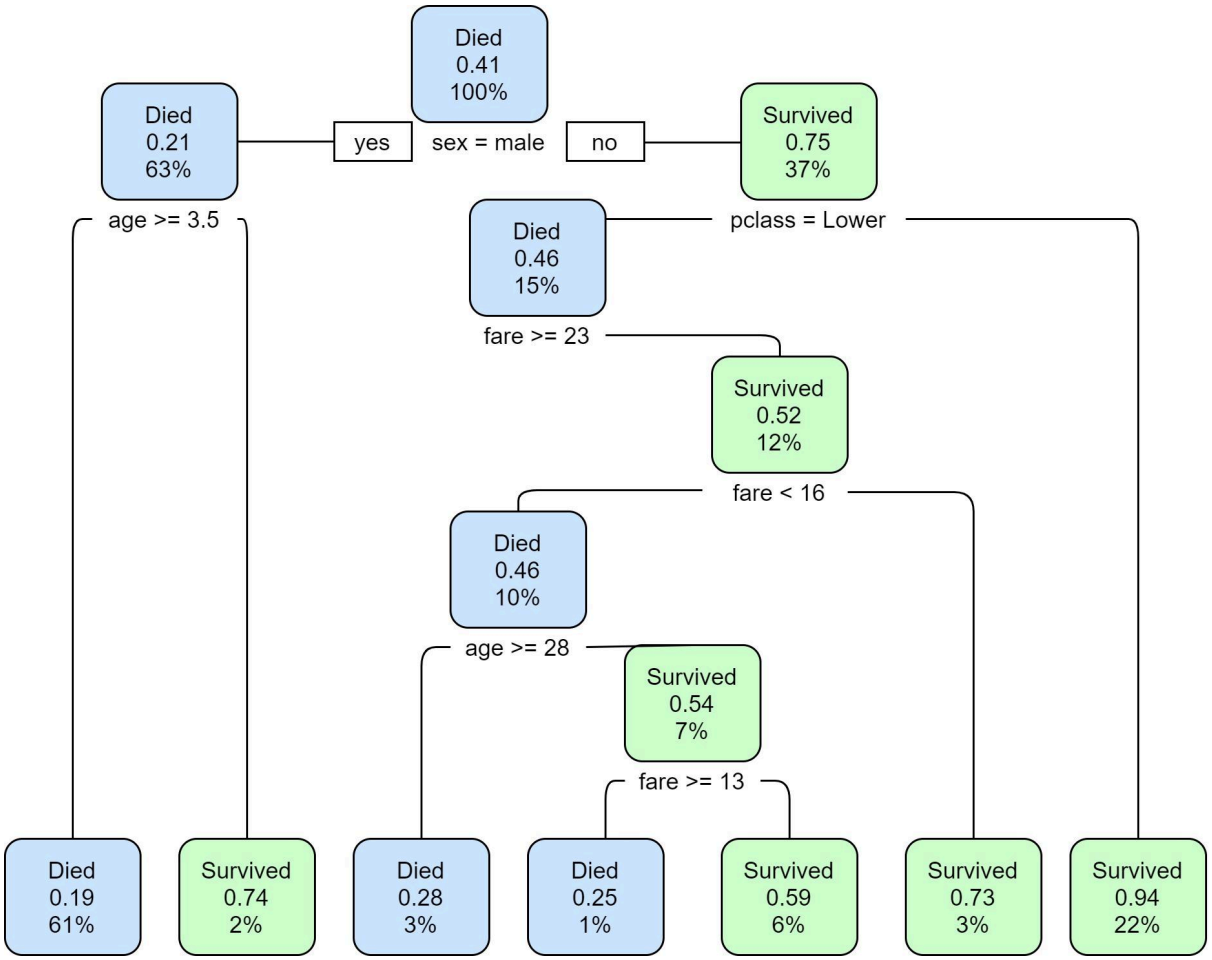
For example, in the loan application process, the decision-making journey can be likened to a machine learning decision tree. To initiate the process, the question is asked, '`Is your net monthly > 5000`.' If the answer is no, the decision stream ends, and the outcome is that you '`Don't qualify for a loan`.' If the answer is yes, the next step involves checking whether you '`Have a good credit record`.' If the answer is no, again the decision stream ends, and the outcome is '`Don't qualify for a loan`.' Conversely, if you do have a good credit record, the ultimate decision is made: '`Qualify for a loan`.' This example briefly illustrates how a decision tree effectively guides the decision-making process and can be visualised in the following tree-like diagram:



For all decision trees, you start at the top, follow the paths that describe the current condition, and keep doing that until you reach a decision. Note that the decision – the bold text in the diagram – is made at the 'leaves' of the tree (the end of a branch, with no more branches coming off of it).

Classification trees

Decision trees created for datasets with a categorical dependent variable are called classification trees. As an example, let's look at the Titanic dataset. A tree model of this dataset shows us the likelihood of different kinds of passengers surviving the sinking of the Titanic. The tree consists of nodes, and each node has a rule that determines whether an instance moves on to the left or right child node. At the end of each possible path is a leaf. In a classification tree, a leaf contains the predicted label for an instance with those input features.

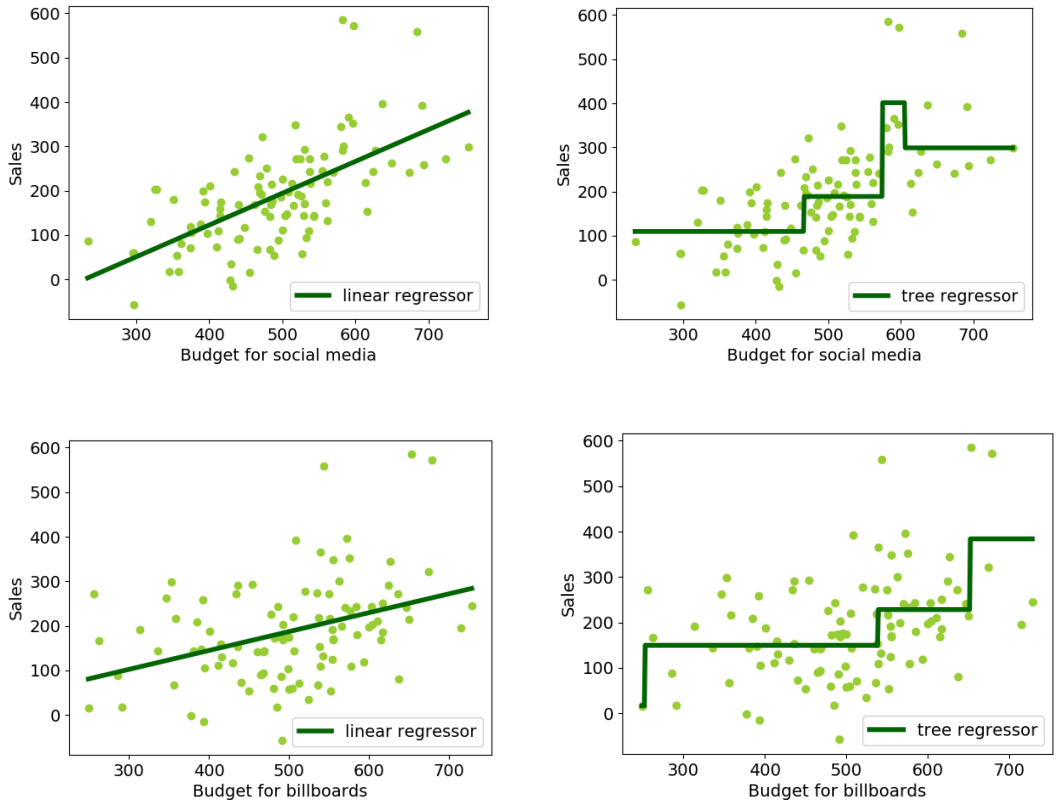


This particular tree also shows at each node what the probability of survival is. Without any prior knowledge, a passenger's chance of survival was 41%. But if we know that an instance is male, survival is a lot less likely. This tree shows that the lowest chance of survival was for adult males.

Regression trees

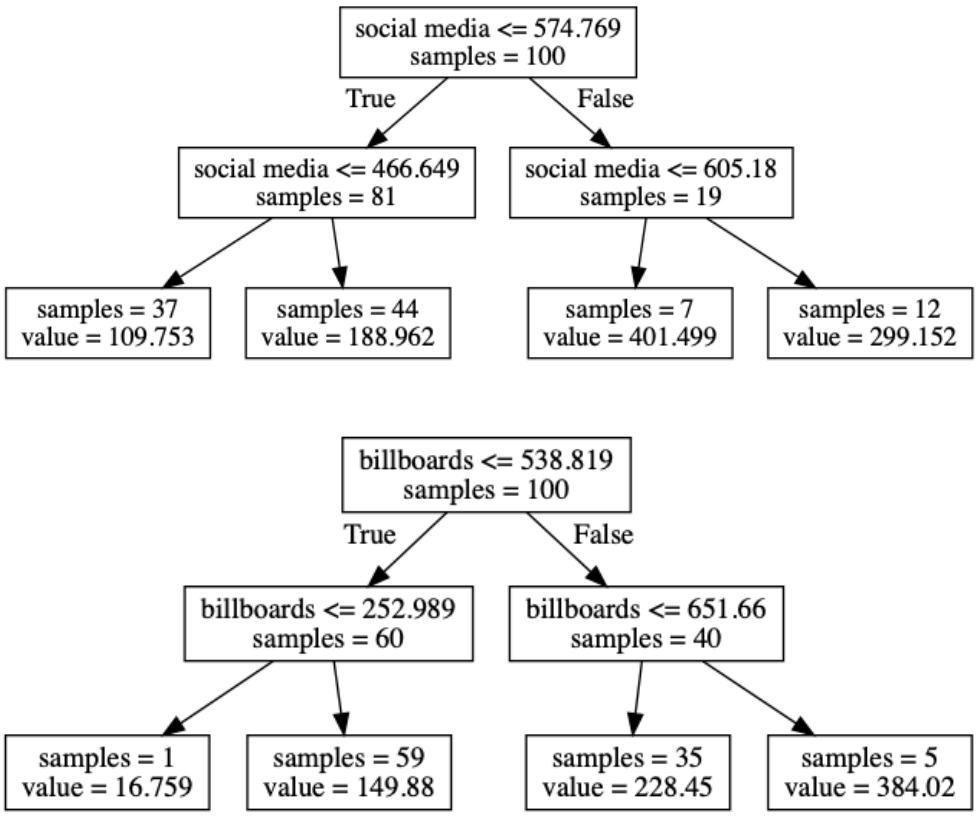
Decision trees can also be used for problems with a numerical dependent variable. A regression decision tree divides input features into regions and assigns categories to those regions. The regions and their labels are based on what best fits the data. Again, best fit here means the set of regions that minimises the distance between the predictions of the model and the observed values.

So while a linear regression approach to the advertising problem below gives different predictions for each unique value of *x*, a regression tree gives different predictions for regions of *x*.



You can imagine that this approach is more flexible. The regions could capture a steeper or less steep increase in sales if that fits the data better than a linear model. The regions can also be arbitrarily specific or broad. Trees can also be visualised easily to get information about the decisions the model makes, in case we want to change the parameters of the model, as we will discuss in a bit.

Consider the following diagrams.



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

When discussing trees, it's important to note that, due to their flexibility, they are prone to something called 'overfitting'. Overfitting is one of the biggest causes of poor performance of machine learning algorithms, together with its counterpart, underfitting. Let's consider these concepts in more detail.

Underfitting

Underfitting refers to a model that can neither model the training data nor generalise to new data. The training error is high because the model was not able to make good predictions based on the input features. A model may fail to fit the data because it is too simple to capture the patterns, but underfitting is more commonly caused by a problem with the task setup (e.g., the features x are not a

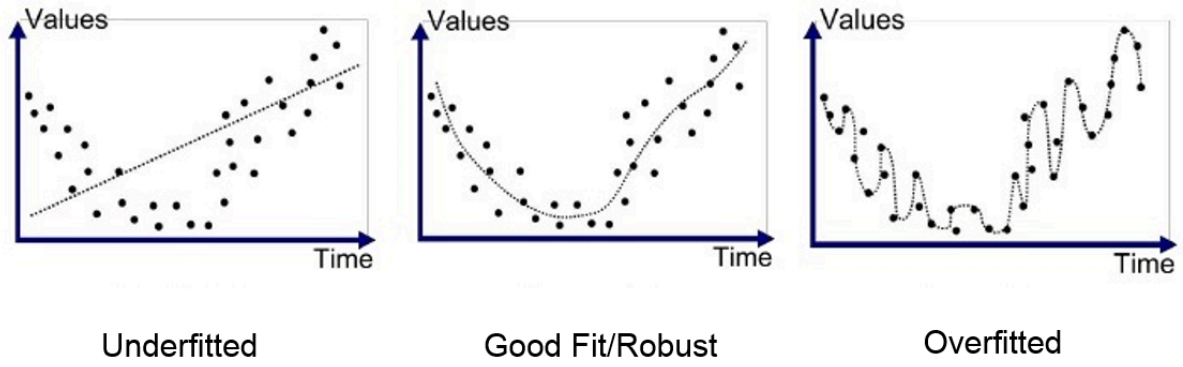
good predictor of *y*) or with the training data (e.g., there is too little training data to learn from, or it contains too many mistakes).

Overfitting

An underfitting model has low training error, but will not perform well on test data. However, that does not mean that a model with low training error is automatically going to do well on test data. A model with very low training error may suffer from overfitting: Some of the rules the model learned are only applicable to training data and are not useful for solving the overall problem we want to solve. The model is too tailored and does not generalise well. It is important that the model is able to generalise beyond the training data, as this data is only a sample. If the model is overfitted, it will perform very well on training data, but it won't translate to good performance on test data.

Diagnosing overfitting and underfitting

What are the causes of overfitting and underfitting? There are many reasons, and it can be typically difficult to determine. Generally speaking, an overly complex model will overfit, and an overly simple model will underfit. Think of overfitting as simply 'remembering' the training data piece by piece, without learning how to construct a rule. Examine the three graphs below:



The first graph on the left is an incredibly simple model: it's just a straight line. There is no way that a straight line will be able to predict that data, hence you will see a high training error and a high testing error.

The middle graph is probably the best fit for the data. I'm sure that you can picture more samples of this data being added, and the model being able to give a rough estimate of what that should be. Note that the training error will be lower than an underfit, but higher than an overfit model. However, the testing error will be lower

than both – testing error is what is most important. Let this be a lesson: Zero error is rarely a good thing, and lower training error doesn't always mean a better model.

The last graph on the right shows a model that has overfit. The training error here will be incredibly low, and practically zero! That being said, if more data were added, it's likely that the model will make a valuable prediction, as it doesn't seem to follow the overall trend of the data.

Fixing overfitting and underfitting

Theoretically, fixing overfitting and underfitting should be easy. If it is underfitting, then use a more complex model. If it is overfitting, then use a simpler model. However, this isn't always the case, and there could be several issues causing overfitting and underfitting. However, this is generally a good starting point.

When fixing overfitting, sometimes a simpler model will cause major underfitting. In some cases, we want to keep *some* of the complexity of our model, but not all of it. This can be fixed using a technique called **regularisation**. This is an important technique present in many (if not all) ML applications. In short, regularisation is the technique of penalising more complex components of the model during training to force the model to minimise its usage. Let's say that you have a complex model that looks something like this:

$$y = ax^5 + bx^4 + cx^3 + dx^2 + ex^1 + f$$

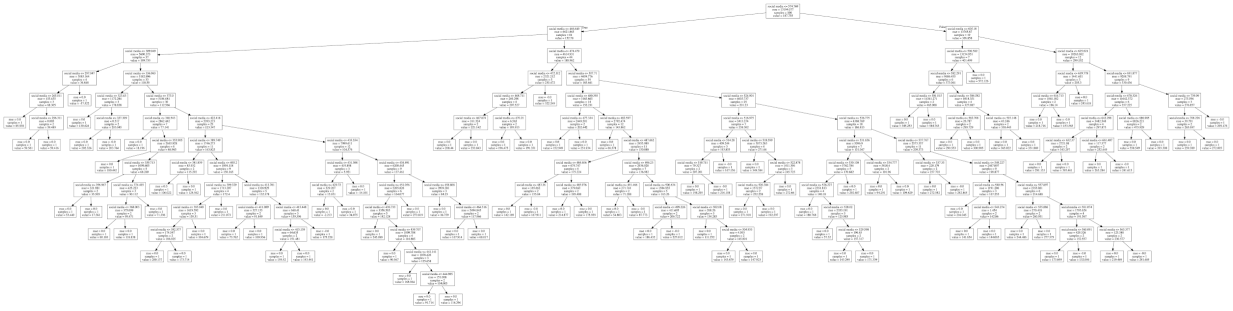
And you need to find the values of *a*, *b*, *c*, *d*, *e*, and *f*. This model is overfitting. However, you can't construct anything simpler than this, as it underfits the data. During training, you use the mean squared error to calculate the loss function. However, you can also add something like $5a + 4b + 3c + 2d + e$ to the error. This means that the model will benefit more by using lower values of the higher-order parameters.

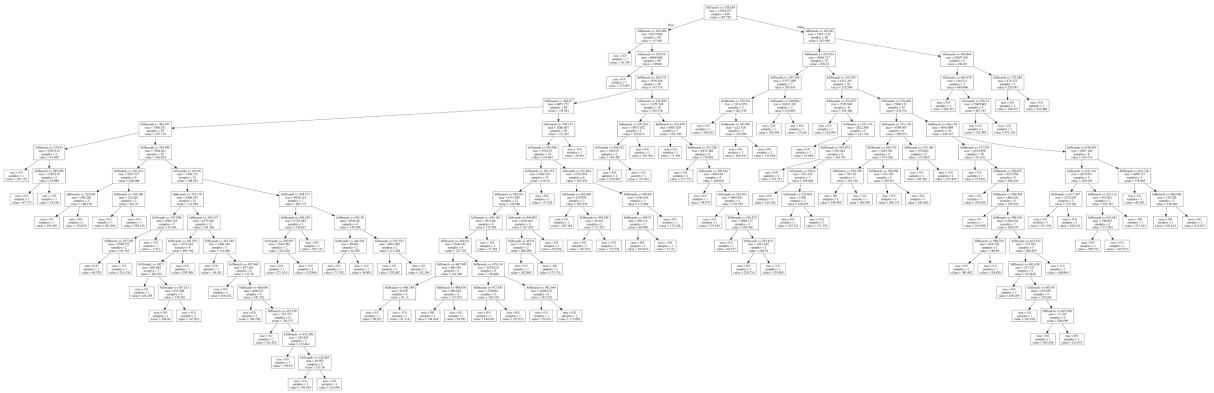
Pruning

As mentioned before, decision trees are very flexible and, therefore, are likely to overfit training data. This problem can be addressed by 'pruning' a tree after training in order to remove some of the detail it has picked up, and to therefore make the model more abstract and general.

A strategy to prevent overfitting is thus to grow a very large tree without any restrictions at first, which is then pruned to retain a more general subtree. In fact, that is how the example trees for the advertising budgets were created. If each of

those trees were left to develop without restrictions, they would have become this detailed:





Creating simpler trees

To create simpler trees, the `max_depth` parameter can be used to control the maximum depth or levels that the decision tree can grow.

By limiting the `max_depth` you control the complexity and size of the tree. A shallow tree (with a low `max_depth`) may underfit the data by oversimplifying, while a deep tree (with a high `max_depth`) may overfit the data by capturing too many patterns or noise. Therefore, it's important to experiment with different values of `max_depth` to find the optimal balance between underfitting and overfitting.

It's important to note that while both pruning and `max_depth` contribute to simplifying decision trees, they differ in their approach. Pruning involves removing branches after the tree is trained, based on a cost complexity parameter, to

determine the optimal tree size. On the other hand, `max_depth` sets a predefined limit on the tree's depth during the training process.

In the example below, the decision tree classifier is created with a low `max_depth` of 3. This means that the resulting tree will have a maximum depth of 3 levels, making it relatively shallow and simpler.

```
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.

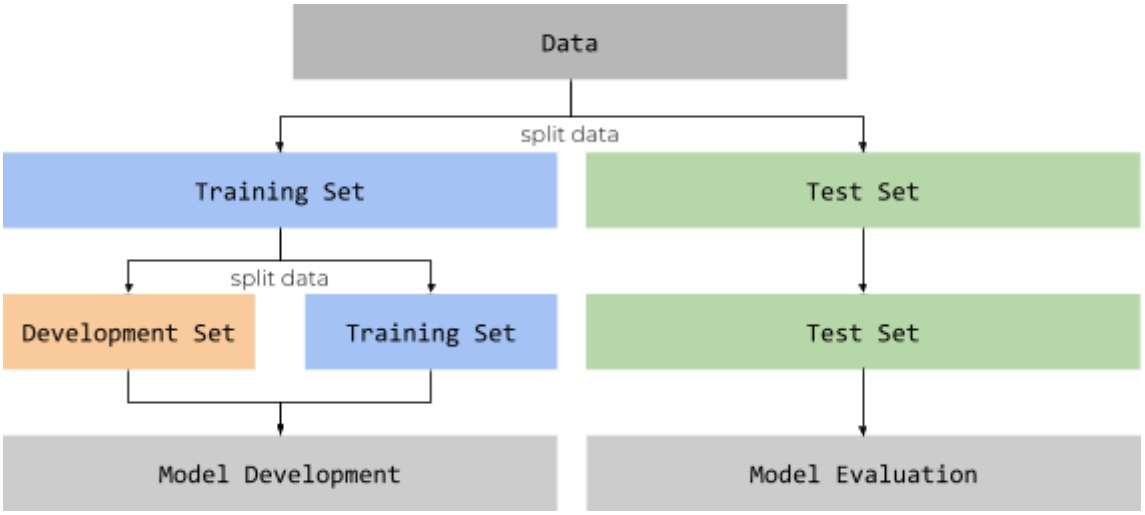
```
from sklearn.tree import DecisionTreeClassifier

# 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)
```

DEVELOPMENT DATA

How do we determine the best way to prune a tree? We can take subtrees of an unpruned tree and compute the test error of predictions of that subtree. We can then select the subtree with the lowest test error rate.

However, once we have done that, we have fitted our pruning parameter to the `test` data. This means the `test` data is no longer completely unseen.

The best way to prune a decision tree is to assess its performance on **unseen data**. To avoid fitting the `test` data, we can split the dataset into three parts, instead of two, before training: a `training` set, a **`development`** set, and a `test` set.



Take note:

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

The `development` set allows us to assess how well the model generalises to data that is not part of the `training` set. This helps us identify and address issues like underfitting or overfitting. By separating the `development` set from the `test` set, we ensure that the model is evaluated on completely unseen data at the end of the training process.

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.

Instructions

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

Practical Task

Follow these steps:

- Create a decision tree that can predict the survival of passengers on the Titanic. Make sure not to impose any restrictions on the depth of the tree.
- Use the **decision_tree_titanic.ipynb** file provided, which has the **titanic.csv** dataset loaded (sourced **here**) to complete the task.
- Select relevant variables from the data and split the data into a `training`, `development`, and `test` set.
- Train a decision tree and make a plot of it.
- Compute your model's accuracy on the **development set**.
- Try building your model with different values of `max_depth` (2–10). At each step, create a plot of your tree and store the accuracies on both the training and development data.
- Plot a line of your training accuracies and another of your development accuracies in the same graph. Write down what shape the lines have and what this shape means.
- Report the accuracy of your final model on the **test data**.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.