Machine Learning Assignment 2

Classification problems

Sander Poelmans, Damianus Wakker FIIW INF

Decision Trees

For the first classification technique, we chose the decision tree. The sklearn library in Python has an excellent implementation of this.

This implementation works in the following way:

1. **Decision Tree Structure:**

   The decision tree is a regular Tree data structure with each tree node testing a particular attribute. When going through all the testing attributes, you reach the leaves (The final node of the tree), and this leave is the final classification of the decision tree.

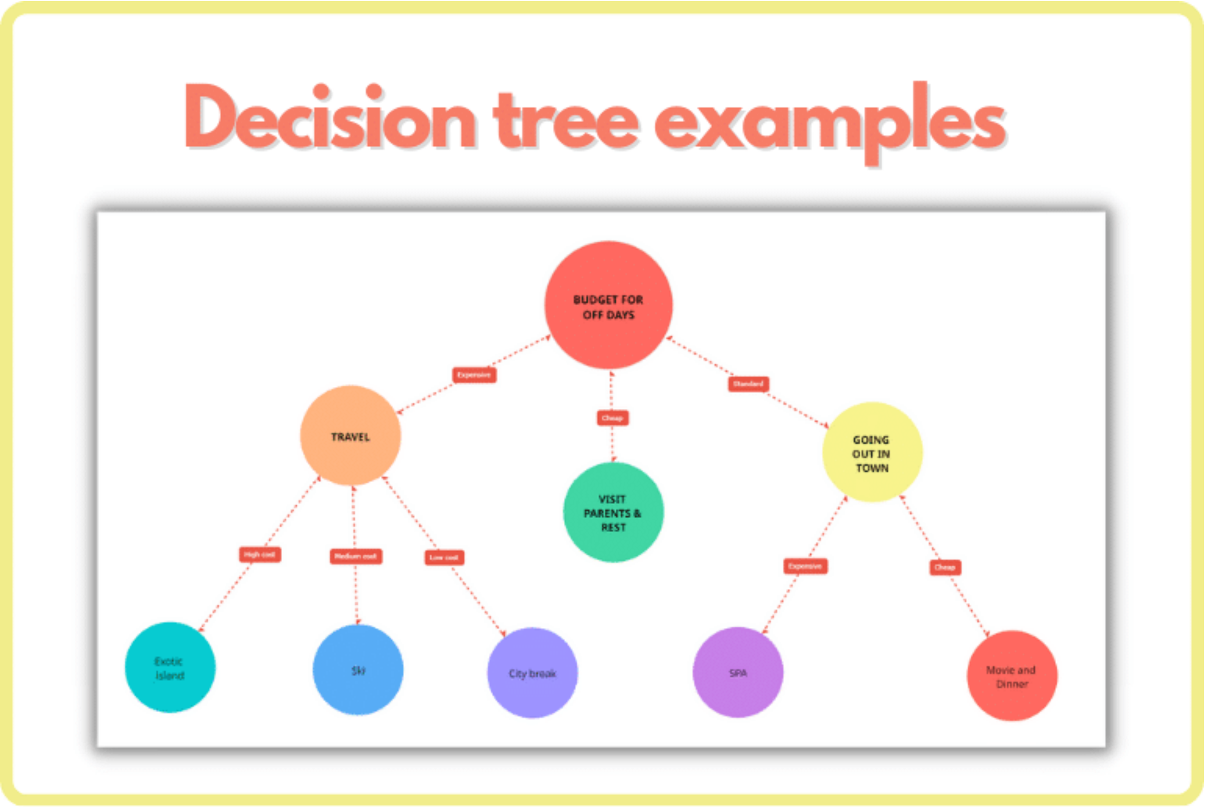

Figure 1: decision tree example (https://www.mindomo.com/blog/decision-tree-examples/)

2. **Splitting Criteria:**

To generate the decision tree, there need to be criteria to split the data you input in the tree based on decisions. The primary way these are decided is by:

- Gini impurity: how often a random element will be wrongly classified
- Entropy: The disorder/ uncertainty in the dataset

3. **Overfitting :**

   Just like in regression, there can be overfitting in decision trees. To prevent this, there are a bunch of parameters you can change, like:

- Max depth: This limits the depth of the tree and prevents it from getting too large
- Min Sample Split: This sets the minimum amount of samples required to split in the node
- Min Sample Leaf: This sets the minimum amount of samples required to be at a leaf node

4. **Feature Importance:**

   Once the classifier finishes training the Decision tree, it provides a “Feature importance score”. This is done for each feature and indicates how important each feature is. This helps reduce the errors in the tree.

<u>Our model & result:</u>

Some key decision points are dropping the neighbourhood column, splitting the data into training and testing sets, and variables in the Decision Tree regressor function. Most of the other things are hidden within parts of the scikit-learn library, and explanation of the decision tree can be found in the previous paragraph.

We removed the neighbourhood column because the function DecisionTreeRegressor could not handle not numerical values. If this was allowed the r-squared value would have been higher then with the current model.

For splitting the data, we went with the standard split in the field, 80% training data and 20% testing data. This is best as this is a large dataset of 2MB but not enormous like 200MB, where giving less percentage to testing may be possible. But in our case, if this split was not applied, the test was not possible.

For the variables in the DecisionTreeRegressor function, we changed the max depth to 5. At first, we did not set a max depth, causing the program to run very long, which we let run overnight once. This resulted in a decision tree graph where Adobe Acrobat had issues opening it, but it had a low r-squared value. Even when the max depth was 10, the R-squared was lower, with a value of 0.56 instead of 0.57.

This R-squared value is lower but still acceptable as it is above 0.5. We hypothesise that when the neighbourhood can be considered in the variables, the R-squared value will be higher. The model has an acceptable outcome to this dataset but could be better.

Neural Network

The second technique of classification we used is Neural Networks.

This implementation works in the following way:

1. **Splitting the data:**

   Before we can even start training the model, we need to split the dataset into two main groups. The training dataset and the testing dataset:

- Training set: about 70-80% of the entire dataset is generally used to train the classification model. This amount can change slightly.
- Testing data: We do not want to test the model on the training data since then, the model would still know the correct classification. Therefore, we hold ~20% of the dataset on the side to test the model's accuracy.

2. **Creating the Model:**
   To create the Neural network model, we need to add a bunch of layers. Each of these layers has its purpose:

- **Convolution layer**:
  model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))

  This example line uses 32 kernels of size 3x3. These are the things that figure out the patterns in the input image.

  You can also see that the input shape of the image used in this network is 28x28 pixels.

  The real activation means that this neural network can deal with non-linearity. This is required to learn the patterns in the images.

- **Max Pooing layer:**
  model.add(layers.MaxPooling2D((2, 2)))

  This layer takes the features in the layer above and condenses them into a 2x2 kernel, making the whole model more efficient and can help fix some problems with small translations in the features.

- **Flatten layer:**
  model.add(layers.Flatten())

  The flattened layer is the most simple one. It simply converts the 2D feature map from the layer above it into a 1D array.

- **Dense layer**:
  model.add(layers.Dense(64, activation='relu'))

  The dense layer is a layer that is fully connected. It connects all inputs to all outputs, helping with processing flattened layers, for example.

```
model.add(layers.Dense(10, activation='softmax'))
```

This version of the dense layer can be used for the ten outputs of a digit recognising NN. The softmax activation converts the output into probabilities, making the results easier to read.

These layers can be used to build the ideal model for the Neural Network we need.

Our model & results :

For the creation of our model, we used the following layers:

```
layers.Conv2D(filters=128, kernel_size=(3, 3), activation='relu', input_shape=(32, 32, 3)),

layers.MaxPooling2D((2, 2)),

 layers.Conv2D(filters=96, kernel_size=(3, 3), activation='relu'), layers.MaxPooling2D((2, 2)),

 layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu'), layers.MaxPooling2D((2, 2)),

layers.Flatten(),

layers.Dense(64, activation='relu'),

layers.Dense(10, activation='softmax')
```

So, our model starts with a 128 filter 3x3 kernel-sized layer that takes 32x32 image input. After this, we use the maxpooling layer to condense the previous layer's output. This also removes some of the translation errors. This process is repeated two more times with 96 and 32 filters each.

After this, the flattened layer is used to convert the 2D array output into a one-dimensional one. Then, we run it through a fully connected layer with 64 nodes to deliver it to the last dense layer with the ten outputs with probabilities.

To test our model, we used the CIFAR-10 dataset (https://www.kaggle.com/datasets/fedesoriano/cifar10-python-in-csv). We chose this model as this is a very well-known dataset to perform image classification tasks. The combination of the dataset and our model gave us an accuracy of 69.4%. This was with fourty epochs but also with ten epochs around 69%.