IV. Decision Trees (DT) (pp.72-85)

The *two-half-moons* dataset:

The *concentric-circles* dataset (not in text):

The *xor-xnor* dataset (not in text):

No-name dataset (not in text):

Is there a linear boundary that correctly separates the two classes?

Conclusion: There are non-linear problems, on which linear models/algorithms perform poorly.

Decision trees (DTs) are one family of **non-linear** classification/regression algorithms.

How to use a DT for classification:
For continuous features, the decisions are comparisons with a threshold value:

How are the threshold values chosen?

p.74: “Splitting the dataset horizontally at x[1]=0.0596 yields the most information, it best separates the points in class 1 from the points in class 0.”
To define *most information*, or *best separation* rigorously, we need the concept of *entropy* (not in text).

We define the entropy of a set with two classes:

- The total nr. of points is $N$.
- There are $m$ points of class 0 and $n$ points of class 1. We have $m + n = N$.
- $p = m/N$ and $q = n/N$. We have $p + q = 1$ (normalization).

$$E = - p \cdot \log_2 p - q \cdot \log_2 q$$

Explain: sign, base, 0, 1, and 0.5.

**EC**

Calculate the entropy of this set:

- Hint: Write a Python function to implement the formula above!
After the set is split, we combine the entropies of the two subsets like so:

\[ E_{\text{split}} = \frac{N_1}{N} E_1 + \frac{N_2}{N} E_2 \]

- Calculate the entropy after the split:

- Which split has less entropy? (No need to calculate anything!)

- Which split has less entropy, L2 or L3? (Calculation needed!)

- What is the entropy of L4? (Calculation needed!)

In practice, either entropy or a related measure, the Gini impurity are used. Scikit-learn’s DecisionTreeClassifier supports both:

```
criterion : string, optional (default=”gini”)  
The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity 
and “entropy” for the information gain.
```

Here is a comparison of the two:

Solutions:

□ Calculate the entropy of this set:

```
from math import log

def entropy(n, m):
    if n == 0 or m == 0:
        return 0
    p = float(n)/(n+m)
    q = float(m)/(n+m)
    return -p*log(p, 2) - q*log(q, 2)

print entropy(3, 4)  # 0.9852
```

□ Calculate the entropy after the split:

```
print 3.0/7*entropy(1, 2) + 4.0/7*entropy(2, 2)  # 0.954
```

□ Which split has less entropy? (No need to calculate anything!)
A: The splits are symmetrical, so they have the same entropy.

□ Which split has less entropy, L2 or L3? (Calculation needed!)  
L2 has less entropy.

□ What is the entropy of L4? (Calculation needed!)

```
print 2.0/7*entropy(2, 0) + 5.0/7*entropy(2, 3)  # 0.693
```
Code example

```python
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Accuracy on training set: 1.000
Accuracy on test set: 0.937

The model has memorized all the training data! ... but generalization is still very good!

A note on the ordering of features from the scikit-learn docs:\(^3\):

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and `max_features=n_features`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

What does the tree look like? We will show next time how to visualize the tree, but here is a sample. Bear in mind that this is not the tree we obtained with the code above:

![Decision Tree Diagram]

□ Can you figure out the meaning of `class` and the different shades of blue and orange in the figure above?

---

How can we find information about our tree?

(Not in text): The fitted classifier has a tree data structure that was developed during the training phase.

We can extract the DT descriptors directly from the tree, which is implemented as a dictionary:

```python
In [18]: print tree.tree_.__getstate__() 
```

Here is a better view, using the dictionary's key-value pairs:

- **node_count** includes the root.
- The first node in the array (node 0) is the root, and the order is based on Depth First Search (DFS) - preorder traversal.
- Internal nodes have left_child and right_child: node [0] has nodes [1] and [4] as children. Leaves have -1 value for the children: nodes 2, 3, 5, 6 are leaves.
- The next element in the node descriptor is the index of the feature that was used for decision: in node 0, it is feature 20.
- The next element is the numeric threshold of the feature: in node 0, it is 16.79
- The next element is the impurity (similar to entropy): only node 5 is pure.
- **n_node_samples** tells us how many data points are in the node. **values** goes further and shows the split of those points between the classes.
- **max_depth** is measured from the root, which has zero depth.
Remember DFS-preorder traversal:

Apply DFT-preorder on this tree to number the nodes:

The nodes of the example tree above are listed below. Draw the tree and verify that they are listed in DFS-preorder:

**Visualizing the tree**

In order to run the code on p.78 of our text, we first need to install the visualization program **graphviz** and its Python interface:

```
> conda install graphviz  (only installs graphviz binaries/executables)
> conda install python-graphviz  (installs a DLL that gives Python access to the graphviz binaries)
```

And now we are ready for

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot",
                class_names=["malignant", "benign"],
                feature_names=cancer.feature_names,
                impurity=False, filled=True)

import graphviz
with open("tree.dot") as f:
    dot_graph = f.read()
display(graphviz.Source(dot_graph))
```

Notes:

- DOT is a graph description language, used by graphviz to represent graphs. If you want to understand it, start with these simple examples: [https://graphs.grevian.org/example](https://graphs.grevian.org/example)

- Unlike the previous uses of iPython’s **display** function, this one cannot be replaced by python’s **print**, which does not “know” what to do with a DOT file.

- Visualizing the DT is impractical for large models, e.g. a human is probably giving up at this stage:
That is why we have other summarizing descriptors, like

**Feature importances**: They are a summary of the DT

```python
print(tree.feature_importances_)
print(tree.feature_importances_.sum())
```

![Decision Tree Image](https://www.datacamp.com/community/tutorials/decision-tree-classification-python)

What feature importances tell us about the data:

- A large feature importance means that that feature separates the classes well ...
- ... however, a small importance does not necessarily mean that the feature is not a good separator: It may actually be better than the “best” feature, it just wasn’t chosen first in the fitting process. This is a major weakness of DTs (and any other algorithm based on selections of features) because it runs into the problem of combinatorial explosion.
- Unlike linear coefficients, importances cannot be negative.
- Unlike linear boundaries, the two sides of a decision don’t necessarily mean a clear separation for classes. Further decision may be needed, as seem in the “non-monotone” example below:

```python
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)
```
In Scikit-learn’s `DecisionTreeClassifier`, when selecting a split point, the default behavior is to look through all features and all values of each feature in order to select the optimal split-point:

```python
splitter : string, optional (default=“best”)
The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split.
```

However, this is a greedy algorithm, that does not guarantee that the two-, three-, ..., n-step sequences of features are also optimal. The only way to guarantee optimality for a sequence of n decisions from among d features is to try out all d^n combinations. (This is the combinatorial explosion problem mentioned above).

---

Controlling complexity in DT models

If all leaves are pure, the DT performs with 1.0 accuracy on the training set, but very likely we have overfitting:

Two methods to prune the DT:
- **pre-pruning** ⇒ Stop the branching based on some criterion, e.g. max. depth, max. nr. of leaves, or min. nr. of nodes in a leaf.
- **post-pruning** ⇒ Build the full tree, with pure leaves, and then collapse nodes with little information (under a certain threshold). Not implemented in scikit-learn!

Remember the scores we got in the cancer code example, with an unconstrained tree:

<table>
<thead>
<tr>
<th>Accuracy on training set</th>
<th>1.000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy on test set</td>
<td>0.937</td>
</tr>
</tbody>
</table>

But LogR with C = 100 had 0.965 on the test set, which means better generalization! Let us experiment with the hyper-parameter *max_depth*, i.e. max nr. of questions that can be asked:

```python
for md in range(6, 0, -1):
    tree = DecisionTreeClassifier(max_depth=md, random_state=0)
    tree.fit(X_train, y_train)
    print("Accuracy on train set: {:.3f}".format(tree.score(X_train, y_train)))
    print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

<table>
<thead>
<tr>
<th>max_depth</th>
<th>Accuracy on train set</th>
<th>Accuracy on test set</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0.998</td>
<td>0.937</td>
</tr>
<tr>
<td>5</td>
<td>0.995</td>
<td>0.950</td>
</tr>
<tr>
<td>4</td>
<td>0.986</td>
<td>0.951</td>
</tr>
<tr>
<td>3</td>
<td>0.977</td>
<td>0.951</td>
</tr>
<tr>
<td>2</td>
<td>0.964</td>
<td>0.950</td>
</tr>
<tr>
<td>1</td>
<td>0.923</td>
<td>0.923</td>
</tr>
</tbody>
</table>

Other hyper-parameters available to control complexity are *min_samples_split*, *min_samples_leaf* and *max_features*. Their meaning is intuitive, but let us elaborate on the subtle difference between *min_samples_split* and *min_samples_leaf*: 
As stated in the Documentation\(^7\), \texttt{min\_samples\_split} applies to the internal nodes of the tree, which explains the following recommendation from the User’s Guide:

“While \texttt{min\_samples\_split} can create arbitrarily small leaves, \texttt{min\_samples\_leaf} guarantees that each leaf has a minimum size, avoiding low-variance, over-fit leaf nodes in regression problems\(^8\).” For example, although the following tree was built with \texttt{min\_samples\_split=4}, it has several leaves with only one datapoint:

\begin{center}
\includegraphics[width=0.5\textwidth]{tree.png}
\end{center}

You may be wondering what use is \texttt{min\_samples\_split} if it allows these “skinny”, overfit trees to develop. Is \texttt{min\_samples\_leaf} not always the better solution? There are several reasons why it is not so:

- \texttt{min\_samples\_split} is a heuristic that is easy to code and cheap (fast) to evaluate, as it is based only on the number of points in the node. Because of this, it has been extensively studied and validated\(^9\). Compare with \texttt{min\_samples\_leaf}, which requires evaluation of various combinations of features and thresholds.

- Even if/when \texttt{min\_samples\_leaf} is implemented, its usefulness as a pruning heuristic is doubtful. As the author of our text notes somewhere else\(^10\), “Anything that forbids particular splits (even if they are all for a given feature) will mean that we just use worse features, not stop splitting”.


\(^8\) [https://scikit-learn.org/stable/modules/tree.html#tips-on-practical-use](https://scikit-learn.org/stable/modules/tree.html#tips-on-practical-use)

\(^9\) For example see Sec. 3.2 in this classical paper: Extremely randomized trees, by Pierre Geurts, Damien Ernst, Louis Wehenkel, 2005.

\(^10\) [https://github.com/scikit-learn/scikit-learn/issues/8399](https://github.com/scikit-learn/scikit-learn/issues/8399)
Using DTs for regression (p.82)

(This explanation is not in text:) We split the (one) target feature into a number of classes. The splitting process is similar to the one used in classification, only instead of entropy or Gini impurity, we use a statistical method, named variance reduction.

To make a prediction, we classify the new data point, and then we take the average of the points in that class:

There's also a variation where the decision tree fits a regression line to the data points of each region, creating a jagged piecewise line. However, trees constructed this way are more prone to overfitting, especially in regions with fewer data points, because noise is weighted more than it should be.

---

11Source of images: https://ml.berkeley.edu/blog/2017/12/26/tutorial-5/
What will this DT regressor predict for:

- -5.5?
- -1.5?
- 8?

We see from these examples that, if the new data points are within the range of the data used to train the model, we can expect reasonable performance. This is called **interpolation**. But what happens if we try to **extrapolate**, i.e. predict for a range outside the one used for training?

Back to our text (p.82): We are using a new dataset, whose points are historical prices of computer RAM:

```
import os
ram_prices = pd.read_csv(os.path.join(mgllearn.datasets.DATA_PATH, 
                                   "ram_price.csv"))
print mgllearn.datasets.DATA_PATH
plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Year")
plt.ylabel("Price in $/Mbyte")
```

C:\ProgramData\Anaconda2\lib\site-packages\mgllearn\data
Intentionally, we train the DT model on one range of data, and then use it to predict another range:

```python
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]
```

We also train a linear regression model for comparison:

![Graph showing the comparison between DT and Linear regression models.](image)

However, it turns out that Linear Regression only looks good because the author gave it a huge helping hand by applying a logarithmic transformation to the data. If we remove that, the DT model wins by a huge margin:

```
DT score : -50.7682517057
Linear score: -5.20335813047e+15
```

![Graph showing the comparison between DT and Linear regression models on transformed data.](image)

This point being made, the larger point about DTs for regression is perfectly valid: They do not generalize well on data outside the training range.

(But then neither does Linear Regression if the data is non-linear, as seen above!)
Note: The coverage of regression in the text and these notes assumes continuous values for the features. If you want to find out how it is done for categorical (discrete) features, a nice, long and detailed example is found here: [http://www.saedsayad.com/decision_tree_reg.htm](http://www.saedsayad.com/decision_tree_reg.htm)

Conclusions on DTs:

- Pre-pruning is a simple method to control DT complexity and limit overfitting. Use one or a combination of the following hyper-parameters: `max_depth`, `max_leaf_nodes`, `min_samples_leaf`.

- Problem with pre-pruning: It is hard to tell when a tree algorithm should stop because the addition of a single extra node may dramatically decrease error (by capturing important structural information about the sample space). This problem is known as the horizon effect.

- DTs are invariant to scaling of the features, b/c each feature is used for decision in isolation. No normalization is needed. They work well when the features have different orders of magnitude, or when they are mixtures of continuous and binary.

- Further reduction in overfitting may be achieved with ensemble methods (next section). With these methods (and especially gradient boosting), DTs are the state of the art for classification problems involving many (1000 or more) attributes, or binary/categorical attributes, or a combination of continuous and binary/categorical attributes.

- Before (deep) neural networks (NNs) had their resurgence (~2012), DTs were the state of the art for most classification problems. Now, they “share” “first place” with NNs.

- Unlike NNs, DTs work well “off the shelf”, i.e. without tinkering with the internal architecture. They also work well when the nr. of data points is not huge (NNs need a lot of training data).