COMP 6721 Applied Artificial Intelligence (Fall 2023)

Lab Exercise #5: Decision Trees & K-means

Solutions

Question 1 Given the training instances below, use information theory to find whether 'Outlook' or 'Windy' is the best feature to decide when to play a game of golf.

Outlook	Temperature	Humidity	Windy	Play / Don't Play
sunny	85	85	false	Don't Play
sunny	80	90	true	Don't Play
overcast	83	78	false	Play
rain	70	96	false	Play
rain	68	80	false	Play
rain	65	70	true	Don't Play
overcast	64	65	true	Play
sunny	72	95	false	Don't Play
sunny	69	70	false	Play
rain	75	80	false	Play
sunny	75	70	true	Play
overcast	72	90	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don't Play

$$H(Output) = H\left(\frac{5}{14}, \frac{9}{14}\right) = -\left(\frac{5}{14}\log_2\frac{5}{14} + \frac{9}{14}\log_2\frac{9}{14}\right) = 0.94$$

$$H(Output|sunny) = H\left(\frac{3}{5}, \frac{2}{5}\right) = -\left(\frac{3}{5}\log_2\frac{3}{5} + \frac{2}{5}\log_2\frac{2}{5}\right) = 0.97$$

$$H(Output|overcast) = H\left(0, 1\right) = -\left(0\log_2 0 + 1\log_2 1\right) = 0$$

$$H(Output|rain) = H\left(\frac{2}{5}, \frac{3}{5}\right) = -\left(\frac{2}{5}\log_2\frac{2}{5} + \frac{3}{5}\log_2\frac{3}{5}\right) = 0.97$$

$$H(Output|Outlook) = \frac{5}{14}H(Output|sunny) + \frac{4}{14}H(Output|overcast) + \frac{5}{14}H(Output|rain)$$

$$H(Output|Outlook) = \frac{5}{14}0.97 + \frac{4}{14}0 + \frac{5}{14}0.97 = 0.69$$

$$gain(Outlook) = H(Output) - H(Output|Outlook) = 0.94 - 0.69 = 0.25$$

$$H(Output|Windy = true) = H\left(\frac{1}{2}, \frac{1}{2}\right) = 1$$

$$H(Output|Windy = false) = H\left(\frac{1}{4}, \frac{3}{4}\right) = 0.81$$

$$H(Output|Windy) = \frac{6}{14}1 + \frac{8}{14}0.81 = 0.89$$

$$gain(Windy) = H(Output) - H(Output|Windy) = 0.94 - 0.89 = 0.05$$

$\Rightarrow$ 'Outlook' is a better feature because it has a bigger information gain.

Question 2 It's time to leave the calculations to your computer: Write a Python program that uses scikit-learn's *Decision Tree Classifier:*[1]

```
import numpy as np
from sklearn import tree
from sklearn import preprocessing
```

Here is the training data from the first question:

```
dataset = np.array([
['sunny', 85, 85, 0, 'Don\'t Play'],
['sunny', 80, 90, 1, 'Don\'t Play'],
['overcast', 83, 78, 0, 'Play'],
['rain', 70, 96, 0, 'Play'],
['rain', 68, 80, 0, 'Play'],
['rain', 65, 70, 1, 'Don\'t Play'],
['overcast', 64, 65, 1, 'Play'],
['sunny', 72, 95, 0, 'Don\'t Play'],
['sunny', 69, 70, 0, 'Play'],
['rain', 75, 80, 0, 'Play'],
['sunny', 75, 70, 1, 'Play'],
['overcast', 72, 90,1, 'Play'],
['overcast', 81, 75, 0, 'Play'],
['rain', 71, 80, 1, 'Don\'t Play'],
])
```

Note that we changed `True` and `False` into `1` and `0`.

For our feature vectors, we need the first four columns:

```
X = dataset[:, 0:4]
```

and for the training labels, we use the last column from the dataset:

```
y = dataset[:, 4]
```

However, you will not be able to use the data as-is: All features must be numerical for training the classifier, so you have to transform the strings into numbers. Fortunately, scikit-learn has a preprocessing class for *label encoding* that we can use:[2]

```
le = preprocessing.LabelEncoder()
X[:, 0] = le.fit_transform(X[:, 0])
```

(Note: you will need to transform `y` as well.)

[1] See https://scikit-learn.org/stable/modules/tree.html#classification

[2] See https://scikit-learn.org/stable/modules/preprocessing_targets.html#preprocessing-targets

Now you can create a classifier object:

```
dtc = tree.DecisionTreeClassifier(criterion="entropy")
```

Note that we are using the *entropy* option for building the tree, which is the method we've studied in class and used on the exercise sheet. Train the classifier to build the tree:

```
dtc.fit(X, y)
```

Now you can predict a new value using `dtc.predict(test)`, just like you did for the Naïve Bayes classifier last week. Note: if you want the string output that you transformed above, you can use the `inverse_transform(predict)` function of a label encoder to change the `predict`ed result back into a string.

Visualizing the decision tree can help you understand how the model makes predictions based on the features. It provides insights into the decision-making process of the classifier. A simple way to print the tree is:

```
tree.plot_tree(dtc)
```

but this can be a bit hard to read; to get a prettier version you can use the *Graphviz*[3] visualization software. Graphviz is a powerful open source software tool for creating visual representations of graphs and networks. Here, we'll be using a Python package called `graphviz` to interface with Graphviz and generate visually informative diagrams, like decision trees, directly from our Python code. You can call it from Python like this:[4]

```
# print a nicer tree using graphviz
import graphviz
dot_data = tree.export_graphviz(dtc, out_file=None,
   feature_names=['Outlook', 'Temperature', 'Humidity', 'Windy'],
   class_names=['Don\'t Play', 'Play'],
   filled=True, rounded=True)
graph = graphviz.Source(dot_data)
graph.render("mytree")
```

The result will be stored in a file `mytree.pdf` and should look like Figure 1.

Most of the code is provided above. For transforming the labels, you can use the same label encoder:

```
y = le.fit_transform(dataset[:, 4])
```

To predict a label for a new feature vector:

```
y_pred = dtc.predict([[2, 81, 95, 1]])
print("Predicted output: ", le.inverse_transform(y_pred))
```

[3] See https://www.graphviz.org/

[4] If it is not yet installed, you can use `conda install graphviz python-graphviz` to install it (note that you need both the Graphviz tool and its Python bindings)

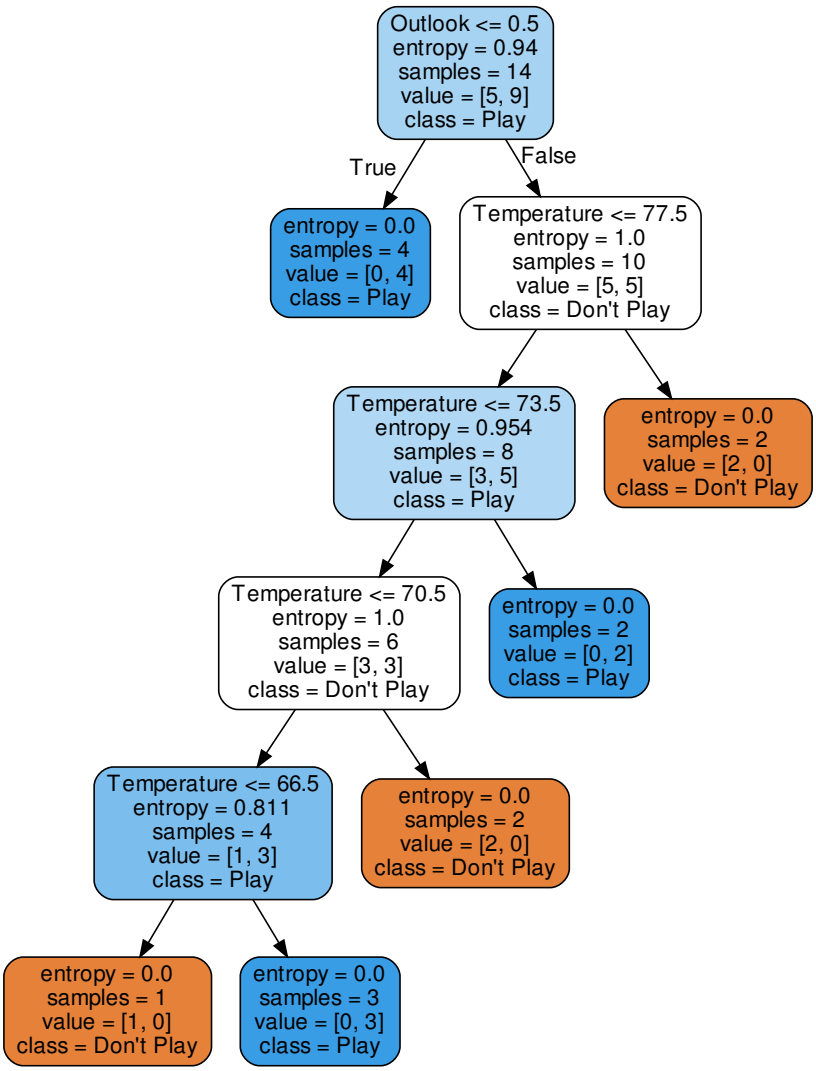

Figure 1: Generated Decision Tree using scikit-learn: Note that the string values for *Outlook* have been changed into numerical values ('overcast'= 0, 'rain'= 1, 'sunny' = 2)

To print the tree using the encoded class names:

```
dot_data = tree.export_graphviz(dtc, out_file=None,
   feature_names=['Outlook', 'Temperature', 'Humidity', 'Windy'],
   class_names=le.classes_,
   filled=True, rounded=True)
```

Question 3 Let's start working with some "real" data: scikit-learn comes with a number of example datasets, including the *Iris* flower dataset. If you do not know this dataset, start by reading https://en.wikipedia.org/wiki/Iris_flower_data_set.[5]

Figure 2: An *Iris versicolor*[6]

(a) Change your program from the previous question to work with this new dataset:

```
from sklearn.datasets import load_iris
X, y = load_iris(return_X_y=True)
```

This code imports the Iris dataset, with the feature data stored in `X` and the target labels in `y`. Print out same data samples to understand this dataset with:

```
print(iris.data[:5]) # Display the first five rows of the dataset
print(iris.feature_names) # Display the names of the features
print(iris.target_names) # Display the target labels
```

Train a decision tree classifier on this dataset and print it out like you did on the previous question.

[5] Also see https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html for the scikit-learn documentation of this dataset.

[6] Photo taken by Danielle Langlois in July 2005 at the Forillon National Park of Canada, Quebec, Canada. Licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. https://commons.wikimedia.org/wiki/File:Iris_versicolor_3.jpg

Here is a minimal solution:

```
from sklearn import tree
from sklearn.datasets import load_iris

# load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# create and print the decision tree
dtc = tree.DecisionTreeClassifier(criterion="entropy")
dtc.fit(X, y)
tree.plot_tree(dtc)
```

For a nicer tree plot, again use graphviz:

```
# print a nicer tree using graphviz
import graphviz
dot_data = tree.export_graphviz(dtc, out_file=None,
                                feature_names=iris.feature_names,
                                class_names=iris.target_names,
                                filled=True, rounded=True,
                                special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("iristree")
```

(b) Now you have to *evaluate* the performance of your classifier. Use scikit-learn's `train_test_split` helper function[7] to split the Iris dataset into a *training* and *testing* subset, as discussed in the lecture. This allows you to assess how well your classifier generalizes to unseen data.

To create an 80%/20% split of the training/testing data, use:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.2, random_state=0)
```

The `random_state` parameter ensures consistency in the results every time the code is run, enabling a reproducible split between the training and test datasets.

After splitting the dataset, apply your decision tree classifier to the *testing set* to make predictions.

```
y_pred = dtc.predict(X_test)
```

Now you can run the evaluation by computing the *Precision, Recall,* F_1*-measure, and Accuracy* of your classifier, by using the tools in scikit-learn.[8]

[7] See https://scikit-learn.org/stable/modules/cross_validation.html

[8] See https://scikit-learn.org/stable/modules/model_evaluation.html

Make sure to print the values of these metrics for your decision tree classifier's performance on the testing set. This evaluation will help you understand how well your classifier performs in terms of its ability to correctly classify different Iris flower species.

You can now generate the individual metrics, e.g., for precision, you can use:

```
from sklearn.metrics import precision_score
precision = precision_score(y_test, y_pred, average='weighted')
print("Precision: ", precision)
```

You could do the same for the other metrics. However, scikit-learn also has a convenient helper function to produce a report for all the metrics, the `classification_report`:[9]

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

Which should result in a report like this:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00         6

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

As you can see, the data was easy to learn and your classifier has a perfect score. Try decreasing the training data to just 50% of the dataset to see some errors.

Note: Since we have more than two classes, the above metrics are an *average* of the values for each individual class. You can see that we set this explicitly when computing the precision score separately.

(c) Finally, compute and print out the *confusion matrix.*[10]

This is easy using the helper function `confusion_matrix`:

```
from sklearn.metrics import confusion_matrix
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

which results in:

[9] See https://scikit-learn.org/stable/modules/model_evaluation.html#classification-report

[10] See https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

```
Confusion Matrix:
 [[11  0  0]
 [ 0 13  0]
 [ 0  0  6]]
```

Remember from the lecture that the confusion matrix provides a visual representation of our classifier's performance; in it, each row represents the *true class* while each column indicates the *predicted class.* You can see that here, the classifier did not misclassify any of the samples in the test dataset.

Question 4 Consider the following data set with two attributes for each of seven individuals.

	a1	a2
Data1	1.0	1.0
Data2	1.5	2.0
Data3	3.0	4.0
Data4	5.0	7.0
Data5	3.5	5.0
Data6	4.5	5.0
Data7	3.5	4.5

(a) The data set is to be grouped into two clusters. Initialize the clusters using Data1 and Data4 as initial centroids to the two clusters. That is, allocate the remaining individual to one of the two clusters using the Euclidean distance.

	Individuals	Centroid
Group 1	1	(1.0, 1.0)
Group 2	4	(5.0, 7.0)

For Data2:

Distance to Centroid 1: $\sqrt{(1.5-1.0)^2+(2.0-1.0)^2} \approx 1.1$

Distance to Centroid 2: $\sqrt{(1.5-5.0)^2+(2.0-7.0)^2} \approx 6.1$

	Distance to Centroid 1	Distance to Centroid 2
Data1	0.0	7.2
Data2	1.1	6.1
Data3	3.6	3.6
Data4	7.2	0.0
Data5	4.7	2.5
Data6	5.3	2.1
Data7	4.3	2.9

(b) Recalculate the centroids based on the current partition, reassign the individuals based on the new centroids. Which individuals (if any) changed clusters as a result?

For Group 1:

$$\frac{1.0+1.5+3.0}{3} = 1.83$$

$$\frac{1.0 + 2.0 + 4.0}{3} \approx 2.33$$

	Individuals	Centroid
Group 1	1, 2, 3	(1.83, 2.33)
Group 2	4, 5, 6, 7	(4.125, 5.375)

	Distance to Centroid 1	Distance to Centroid 2
Data1	1.6	5.4
Data2	0.5	4.3
Data3	2.0	1.8
Data4	5.6	1.9
Data5	3.1	0.7
Data6	3.8	0.5
Data7	2.7	1.1

Data3 changed from Group 1 to Group 2. All the other individuals remained in the same cluster.

Question 5 scikit-learn also has an implementation for K-means.[11]

(a) Complete clustering the data from the previous question using scikit-learn's K-means implementation:

```
dataset = np.array([
[1.0, 1.0],
[1.5, 2.0],
[3.0, 4.0],
[5.0, 7.0],
[3.5, 5.0],
[4.5, 5.0],
[3.5, 4.5]])
```

Note that with K-means, you do not need any labels, since we are doing unsupervised learning:

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2)
kmeans.fit(dataset)
```

To print the results, you can use:

```
print(kmeans.labels_)
```

This will show you for each element in the `dataset` which cluster it belongs to.

You should get an output like this:

```
[0 0 1 1 1 1 1]
```

(b) Now cluster the *Iris* flower dataset: How well do the clusters represent the three species of *Iris?*

To answer this question, we'll use *Matplotlib*,[12] a comprehensive library for creating static, animated, and interactive visualizations in Python. Originally developed by John D. Hunter in 2003, it has since become one of the most widely used data visualization tools in the Python ecosystem. You can use it to create a 3D-plot of your clusters as shown below:

```
# Plot the results using matplotlib
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(10, 8))
plot = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
plot.set_xlabel('Petal width')
plot.set_ylabel('Sepal length')
plot.set_zlabel('Petal length')
plot.scatter(X[:, 3], X[:, 0], X[:, 2], c=kmeans.labels_, edgecolor='k')
```

[11] See https://scikit-learn.org/stable/modules/clustering.html#k-means

[12] See https://matplotlib.org/

The result should look similar to Figure 3. The 3D plot visualizes the clustering based on three features: petal width, sepal length, and petal length. This visual representation aids in discerning how well the algorithm has clustered the data points based on these features.

Here is a minimal, complete program to cluster the *Iris* flower dataset:

```
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

# load the Iris dataset
iris = load_iris()
X = iris.data

# use k-means to cluster the dataset
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

Plotting the results using Matplotlib works just as described above.

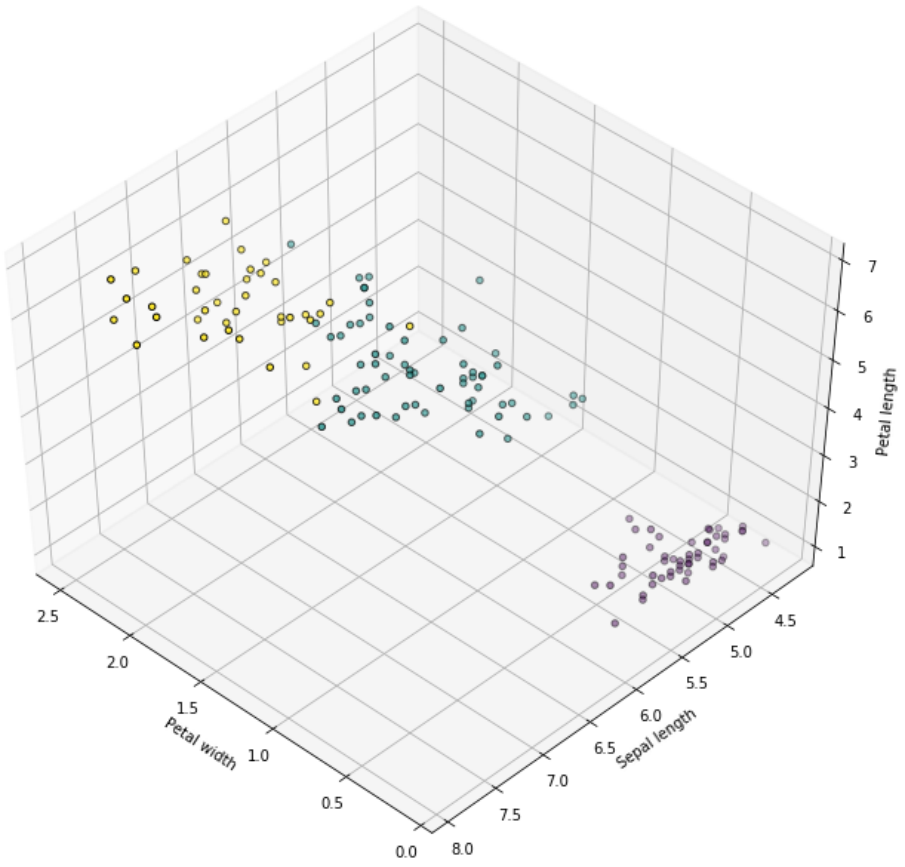

Figure 3: Plotting the results of clustering with K-means on the *Iris* dataset with Matplotlib