QBUS6850: Tutorial 7 – Advanced Classification Techniques I

Objectives

- To learn how to build Decision Trees;
- To learn how to use decision tree model from scikit;
- To learn how to visualising trees or extract rules

Note: To complete all the tasks in this tutorial, you need install Graphviz and Pydotplus. See the last section for information

1. Understanding Decision Trees

Almost all decision trees algorithms work on the same shared principle. That is, recursively partition the data to maximise the purity of classes in each partition. Under this principle eventually the leaf nodes (partitions) will be pure or close to pure.

Below I will demonstrate how this principle works in practice through a naive implementation of ID3 that builds a single layer tree i.e. a single root node and two leaf nodes. In such a shallow tree we need only compute one split of the data so we can omit the recursive component.

```
ID3_ (Iterative Dichotomiser 3) was developed in 1986 by Ross
Quinlan. The algorithm creates a multiway tree, finding for
each node (i.e. in a greedy manner) the categorical feature
that will yield the largest information gain for categorical
targets.
```

from http://scikit-learn.org/stable/modules/tree.html

Refer to Section "1.10.7. Mathematical formulation" to see the mathematical details that used by Python.

To build our single layer decision tree we only need to calculate one partition. This partition should yield the purest partition. In the case of categorical features this is easy, we can produce *N* candidate splits where *N* is the total number of all categories in all features. Then we need to calculate the impurity or information gain for each split and choose the best split. In the case of numerical features things are more complicated (See later "customer default" example).

Classifying the customer purchase data in Lecture 6

Try the “Lecture06_Example01_Updated.py” cod **by yourself** and generate the results on slide 41 and 42.

Step 1: Let’s pre-process data:

```
from sklearn import tree
from sklearn.preprocessing import LabelEncoder
from __future__ import division
import pandas as pd
import numpy as np

purchase_df = pd.read_csv("Lecture6_Data.csv")

purchase_df_x = pd.get_dummies(purchase_df.iloc[:, 1:-1])
feature_names = purchase_df_x.columns

le = LabelEncoder()
y_train = le.fit_transform(purchase_df['Purchase'])

purchase_df_xy = purchase_df_x
purchase_df_xy['y'] = y_train
```

Step 2: Define the classification criteria. In this example we use the expected entropy to measure node impurity. As we are going to repeatedly calculate node impurity, it is better for you to define a function for this. Copy the following code block to your program.

```
def h_function_entropy (rows):
    classes = np.unique(rows.iloc[:, -1])
    entropy_all = []
    N_node = len(rows)

    for k in classes:
        # calcualte the proportion p(x) for each class. Now we
        # have only two classes. 1 and 0.
        prop_temp= len( rows[purchase_df_xy.iloc[:, -1] == k]
)/N_node

        entropy_temp = -(prop_temp) *np.log2(prop_temp)
        entropy_all.append(entropy_temp)

    entropy_all = np.array(entropy_all)
    # calculate the entropy for each subset
    entropy_final= np.sum(entropy_all)
    return entropy_final
```

We ask you read the code first. Answer the following questions:

(1) Given the second statement in the block, what does this function expect from the data rows?
(2) Suppose `k = 1`, what is the meaning of value in the variable prop_temp
(3) Why `entropy_final= np.sum(entropy_all)` in the second last statement? Can you do this summation inside the `for k in classes` loop? How to change?

Step 3: Iterate over all possible splits and calculate the expected entropy. Note that this is the 1st level of split, so the original set of entropy is the same for all possible splits.

In the following code, you shall do the following things. For each feature in the dataset (i.e., all the features are binary, so take values of 0 or 1), we split the dataset into two groups (left node and right node) according to their feature value, then calculate the expected entropy for each group, and the weighted average entropy of two groups. Store them together. Please complete the code

```
loss_list = []

# For each possible split
for i in range(purchase_df_xy.shape[1] - 1):

    # Find observations falling to the left
    left_x = purchase_df_xy[purchase_df_xy.iloc[:, i] == 0]

    # Add your code to find observations falling to the right
    right_x = . . .
    # Calculate the weighted average based on sample size as
the loss of this split

    loss = . . .

    loss_list.append(loss)
```

Step 3: Find the best feature to split. The one with the smallest expected entropy (highest information gain).

```
feature_index = np.argmin(loss_list)
```

Step 4: Now use the best feature to build the tree, and calculate the split nodes (left and right). Count the number of two classes in each node. The ratio can be used to predict the class of future data.

Notice that there is no overlap in class assignment.

```
final_left_rows = purchase_df_xy[purchase_df_xy.iloc[:,
feature_index] == 0]
final_right_rows = purchase_df_xy[purchase_df_xy.iloc[:,
feature_index] == 1]

n_classes = len(np.unique(purchase_df_xy.iloc[:,-1]))
value_left = np.zeros(n_classes)
value_right = np.zeros(n_classes)

for i in final_left_rows.iloc[:, -1]:
    value_left[i] = value_left[i] + 1

for i in final_right_rows.iloc[:, -1]:
    value_right[i] = value_right[i] + 1

print("Left node: {0}".format(value_left))
print("Right node: {0}".format(value_right))
```

2. Decision Trees in scikit-learn

With `sklearn`, Decision Tree building is very straightforward. Simply create a `DecisionTreeClassifier` object and specify parameters such as tree depth, splitting criterion etc.

Sklearn's `DecisionTreeClassifier` implements a variant of the CART algorithm which can produce both decision and regression trees and only accept numerical features. Thus you must convert your categorical features into hot-one features. Please note sklearn takes these 0 or 1 as normal numeric values for calculation.

Step 1: Doing Decision Tree is very simple once you have got your data ready. Note `DecisionTreeClassifier` asks us to provide the data X and t separately. Hence we have to extract them from our DataFrame

```
purchase_df_x = pd.get_dummies(purchase_df.iloc[:, 1:-1])

clf = tree.DecisionTreeClassifier(max_depth = 1)

clf = tree.DecisionTreeClassifier(criterion='entropy',
max_depth = 1)

clf = clf.fit(purchase_df_x, y_train)
```

Step 2: Let's compare the tree that was created by sklearn with our own tree

```
# sklearn tree class assignment
print(clf.tree_.value[1:3])
clf.feature_importances_

# Show our results
print("Left values: {0}".format(value_left))
print("Right values: {0}".format(value_right))
```

Step 3: You can get the class assignment from the sklearn tree by using the `predict()` function. The final class assignment in a decision tree is given by the class with the highest proportion in that node from the training data.

```
most_likely_class =
clf.predict(purchase_df_x.iloc[1,:].values.reshape(1,-1))
print(most_likely_class)
```

Do you know why we use reshape function here? Ask your tutor.

Add your own statements to predict for 2nd to 5th data.

Step 4: If you have successfully install Graphviz and Pydotplus, skip this step. Or copy the following code into your program. Please be careful of indentation

```
from sklearn.tree import _tree

def tree_to_code(tree, feature_names):
     tree_ = tree.tree_
     feature_name = [
          feature_names[i] if i != _tree.TREE_UNDEFINED
else "undefined!"
          for i in tree_.feature
     ]
     print ("def tree({}):".format(",
".join(feature_names)))

     def recurse(node, depth):
          indent = "  " * depth
          if tree_.feature[node] != _tree.TREE_UNDEFINED:
               name = feature_name[node]
               threshold = tree_.threshold[node]
               print ("{}if {} <= {}:".format(indent,
name, threshold))
               recurse(tree_.children_left[node], depth +
1)
               print ("{}else:  # if {} >
{}".format(indent, name, threshold))
               recurse(tree_.children_right[node], depth
+ 1)
          else:
               print ("{}return {}".format(indent,
tree_.value[node]))
     recurse(0, 1)
```

Then you print out the tree rules as if-else statement:

```
# extract true rules
tree_to_code(clf, feature_names)
```

Step 5: Visualising Trees. You have installed Graphviz and Pydotplus, you can visualize the tree in the following way. Note: you may need to change those highlighted in red according to your own machine setting. This shall work on lab computers.

```
import pydotplus
from sklearn.externals.six import StringIO
from IPython.display import Image, display
import os
os.environ["PATH"] += os.pathsep + 'C:/Program Files
(x86)/Graphviz2.38/bin/'

# Create a string buffer to write to (a fake text file)
f = StringIO()
```

```
# Write the tree description data to the file
tree.export_graphviz(clf, out_file=f, proportion=False)

# Produce a visulation from the file
graph = pydotplus.graph_from_dot_data(f.getvalue())

display(Image(graph.create_png()))
```

And you can save the drawing into a file

```
# Write visualisation to image file
graph.write_png("decision_tree.png")
```

3. Business Example

Let's take a look at how you could use a decision tree to determine if someone will default on their loan or not

Step 1: Prepare data

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report,
confusion_matrix

default = pd.read_csv("default.csv")

default = pd.get_dummies(default)

y = default['default']

X = default.iloc[:, 1:]

X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state = 0)
```

Step 2: Write code to explore how many cases of default and how many cases of non-default.

```
# Write your code to count the number of two classes: default
or non-default
```

Step 3: You may have noticed that in this dataset there are very few cases where people default on their loans (most people pay their loans). So we can weight the default class more to improve classification results

This time we will use more layer (more depth) for decision tree. The code creating a decision tree looks like this

```
default_clf = tree.DecisionTreeClassifier(max_depth = 5,
class_weight = 'balanced')

default_clf.fit(X_train, y_train)
```

Step 4: Print the training result and visualise it:

```
print(default_clf.tree_.value[:, 0][0:2])

# Create a string buffer to write to (a fake text file)
f = StringIO()

# Write the tree description data to the file
tree.export_graphviz(default_clf, out_file=f, proportion=True)

# Produce a visulation from the file
graph = pydotplus.graph_from_dot_data(f.getvalue())

display(Image(graph.create_png()))
```

You may print this tree into an image file.

Step 5: Now let's check the prediction accuracy

```
y_pred = default_clf.predict(X_test)

print("Number of defaults in test set:
{0}".format(sum(y_test)))

print(confusion_matrix(y_test, y_pred))

print(classification_report(y_test, y_pred))
```

Here you have seen how we calculate confusion matrix for the classifier and its report.

4. Discussion

Like other methods you have already learnt about, decision trees have hyper parameters. The main parameter is the tree depth. You can try increase or decreasing the tree depth to see how it affects classification results.

To pick the optimal depth you should use cross validation.

The number of samples from each class also plays a roll in the success of our tree. I have shown you how you can apply more favourable weighting to a smaller class using the "balanced" class_weights option. How you weight your classes depends also on the application. In some areas you would prefer to be more conservative, i.e. prefer false positives rather than false negatives, for example in diagnosing patients or lending money. Your weighting scheme should reflect this.