Practical 3: Classification with Nominal Data

1 Overview

The aim of these lab exercises is to give you some hands-on experience with classic classifiers that could be used on data that contains **nominal** values. The **scikit-learn** library is very helpful for many things, but it really only works with numbers. The numbers could be treated as **numeric** data, where there is an inherent ordering and distance between any two values. This is the case with the data we used in Practical 2. The numbers could also be treated as **nominal** labels, where numbers represent encodings for nominal labels; here there is no inherent ordering or distance between two values. We'll use this latter kind of example in the exercises today.

You will use the Iris data set, which contains a set of 150 instances where each instance contains 4 attributes (*sepal length (cm)*, *sepal width (cm)*, *petal length (cm)* and *petal width (cm)*) and a label or class (one of: *setosa*, *versicolor* or *virginica*). The data set is stored as numbers, to the labels are encoded as 0, 1 or 2, representing each of the three classes above, respectively.

You will build three classifiers: **OneR** for classification rules, a **Decision Tree** classifier, and a **Logistic Regression** classifier. For the first method (OneR), you will build simple rules that rely on finding exact values for combinations of attributes (i.e., *equality*). For the second method (Decision Tree), you will build decision points that rely on finding threshold values for comparing attributes to (i.e., *inequality*). For the third method (Logistic Regression), you will build decision boundaries for each class that rely on finding linear separations between pairs of attributes—we will use only two attributes for simplicity, but the method generalises to more than two attributes.

You will also experiment with a number of different types of metrics for evaluating your classifiers, as we discussed in class.

2 Getting Started

First, you need some data.

The **scikit-learn** package has some built-in data sets that are used frequently within the data mining community. This includes the **Iris** data set that we have discussed in class. We will use this data set here.

You can read about the **sklearn.datasets** package here:

`https://scikit-learn.org/stable/datasets/index.html#datasets`

The code in Figure 1 will load in the Iris data set into four arrays:

- iris .target_names is the list of classes (plant classes: *Iris-Setosa*, *Iris-Versicolour*, *Iris-Virginica*);
- iris .feature_names is the set of attributes (sepal length in cm, sepal width in cm, petal length in cm, petal width in cm);
- iris .data contains the instances, i.e., an $n \times 4$ array containing values for each of the four features for each instance, for n instances — these are the χ values; and

- iris.target contains the classes corresponding to the data array (above) and is n in length — these are the y values.

```
from sklearn.datasets import load_iris

iris = load_iris()
print 'classes = ', iris.target_names
print 'attributes = ', iris.feature_names
# iris.data = X values
# iris.target = y values
M = len( iris.data )
print 'number of instances = %d' % ( M )
```

Figure 1: Sample code to load the Iris data set from the **sklearn.datasets** package

→ Write a short script in Python to load the Iris data set (as shown in Figure 1) and test it to make sure it works. Your output should look something like this:

```
classes =  ['setosa' 'versicolor' 'virginica']
attributes =  ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
 'petal width (cm)']
number of instances =  150
```

3 Constructing a Set of Classification Rules

Build a set of **Classification Rules** using the **OneR (1R)** method discussed in class.

→ Write a script in Python that does the following:

In order to assess how accurate a prediction can be made using only one attribute.

1. For the first attribute, *sepal length*, count how often each class value appears for each attribute value in the data set. Then determine which class occurs most frequently for that attribute value. Do this by generating a table, like the one below, indicating which attribute value is the most popular for each attribute (in rows) belonging to each class (in columns). I've filled in part of it to get you started.

sepal length (cm)	*class frequencies:* { 0, 1, 2 }	*most frequent*
4.3	$\{1, 0, 0\}$	0
4.4	$\{3, 0, 0\}$	0
...	...	...
4.9	$\{4, 1, 1\}$	0
5.0	$\{8, 2, 0\}$	0
...	...	...
7.9	$\{0, 0, 1\}$	2

2. Now do the same thing for the second attribute, *sepal width (cm)*.

sepal width (cm)	*class frequencies:* { 0, 1, 2 }	*most frequent*
2.0	$\{0, 1, 0\}$	1
2.2	$\{0, 2, 1\}$	1
...	...	...
4.4	$\{1, 0, 0\}$	0

3. Next, create a matrix over each attribute value for *sepal length* by *sepal width* and fill in the most frequent class(es). For example, the first pair:

$$(sepalLength = 4.3, sepalWidth = 2.0) \rightarrow (0, 1)$$

If more than one class appears, then you need to decide how to choose which class. You could, for example, choose the class with the highest frequency. Choose randomly if there are ties. In the example above, the frequency of class 0 and 1 for each attribute value is 1, so we choose randomly.

Here's another example:

$$(sepalLength = 4.9, sepalWidth = 2.0) \rightarrow (0, 1)$$

In this case, the frequency of $(sepalLength = 4.9, class = 0)$ is 4, while the frequency of $(sepalWidth = 2.0, class = 1)$ is 1, so we pick 0.

for each cell in the table above, create a classification rule

4. Generate a full set of classification rules for all pairs of *sepalLength* and *sepalWidth* values in the Iris data set.

→ **I strongly suggest that you write a Python script to do this!**

5. Now compute the **accuracy** or **score** of your 1R classifier by counting how many classes are predicted correctly over all the instances:

$$score = \frac{correct}{M}$$

where: $correct$ is the number of correctly predicted labels; and M is the number of instances. Or you could compute the **error**, which is:

$$error = \frac{incorrect}{M}$$

where: $incorrect$ is the number of incorrectly predicted labels.

- For this exercise, don't worry about splitting the data into training and test sets, though you could also add that step and then compute the accuracy of your model with both the training set (*resubstitution error*) and the test set (*test error*). If you don't split the data, then the error you compute is the resubstitution error.

4 Constructing a Decision Tree Classifier

Build a **Decision Tree** classifier in **scikit-learn**. The sample code in Figure 2 shows you how to use this classifier to build a decision tree that models the Iris data set.

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

# load the built-in iris data set
iris = load_iris()
print 'classes = ', iris.target_names

# split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split( iris.data, iris.target,
    random_state=0 )

# initialise the decision tree
clf = tree.DecisionTreeClassifier( random_state = 0 )

# fit the tree model to the training data
clf.fit( X_train, y_train )
```

Figure 2: Sample code to run the **sci-kit** Decision Tree classifier on the Iris data set

→ Write a Python script to build a decision tree that models the Iris data set (as shown in Figure 2).

4.1 How good is your model?

As discussed in class, there are a number of ways to score a classifier. Try each of these:

1. Count the number of correctly predicted labels.

```
# predict the labels for the test set
y_hat = clf.predict( X_test )
# count the number of correctly predicted labels
count = 0.0
for i in range( M_test ):
    if ( y_hat[i] == y_test[i] ):
        count += 1
score = ( count / M_test ) * 100
print 'number of correct predictions = %d out of %d = %f%%' % ( count,
    M_test, score )
```

2. Use the **scikit-learn** classifier **score()** function:

```
print 'training score = ', clf.score( X_train, y_train )
print 'test score = ', clf.score( X_test, y_test )
```

   Note that above, you don't have to call the **predict()** function because the **score()** function does that for you.

3. Use the **scikit-learn metrics** package to compute the **accuracy**:

```
print 'accuracy score = ', metrics.accuracy_score( y_test, y_hat )
```

- Note that you should get the same value for the **accuracy score** on the test data for all three methods, above. I get 0.973684210526.

4. You could compute a **confusion matrix**:

```
cm = metrics.confusion_matrix( y_test, y_hat )
```

I get the following:

```
confusion matrix =
           predicted-->
   actual:     setosa versicolor  virginica
    setosa         13          0          0
versicolor          0         15          1
 virginica          0          0          9
```

5. You can also can compute **precision**:

```
precision = metrics.precision_score( y_test, y_hat, average=None )
```

I get the following:

```
precision score = tp / (tp + fp) =
  setosa = 1.000000
  versicolor = 1.000000
  virginica = 0.900000
```

6. You can also can compute **recall**:

```
recall = metrics.recall_score( y_test, y_hat, average=None )
```

I get the following:

```
recall score = tp / (tp + fn) =
  setosa = 1.000000
  versicolor = 0.937500
  virginica = 1.000000
```

7. You can also can compute **F1 score**:

```
f1 = metrics.f1_score( y_test, y_hat, average=None )
```

I get the following:

```
f1 score = 2 * (precision * recall) / (precision + recall) =
  setosa = 1.000000
  versicolor = 0.967742
  virginica = 0.947368
```

4.2 What does the decision tree look like?

The **scikit-learn** Decision Tree classifier returns a data structure called the decision_path which lists the nodes in the tree, starting with the root, and includes information about which feature and threshold values were used to make decisions at that node. Below are two ways to use decision_path .

- Raw Decision Tree Path

  The code below shows you how to get the decision_path data structure:

```
# what does the tree look like?
decision_path = clf.decision_path( iris.data )
                                                print 'decision path: '
print decision_path
```

  When I print it, I get the following (abbreviated below):

```
decision path:
  (0, 0) 1
  (0, 1) 1
  (1, 0) 1
  (1, 1) 1
  (2, 0) 1
 : :
  (149, 0) 1
  (149, 2) 1
  (149, 8) 1
  (149, 12) 1
```

  This is pretty ugly and hard to understand. An explanation of the content of the decision tree path is here:

  https://scikit-learn.org/stable/auto_examples/tree/plot_unveil_tree_structure.html

  This page includes some sample code to decipher the structure. Feel free to try that out.

- GraphViz

  A lovely tool for drawing graphs is called **GraphViz**. This is built around a unix utility called **dot**. The **dot** "language" specifies nodes and edges for drawing graphs, among other features. You may be able to call **dot** directly, depending on how unix is installed on the computer you are using. If you are curious, you can read about the **dot** language here:

  https://www.graphviz.org/doc/info/lang.html

  Graphviz is an open source package that reads **dot** files and creates image files. You can download **Graphviz** here:

  https://www.graphviz.org

The **scikit-learn** Decision Tree package fortunately includes a very handy function called **export_graphviz()** which generates **dot** code for the decision tree. The sample code below shows how to call this function:

```
# output the tree to "dot" format for later visualising
tree.export_graphviz( clf, out_file = 'iris-tree.dot', class_names=iris.
    target_names, impurity=True )
```

You can then either run **dot** on the unix command line (if you have it installed), something like this:

```
unix-prompt$ dot iris-tree.dot -Tpng >iris-tree.png
```

Or you can run **Graphviz**, inputting your **dot** file using the GUI.

Note that there is also a way to run **Graphviz** as a Python package in Anaconda (`https://anaconda.org/anaconda/graphviz`), but this may take some time to install and configure, so I suggest that, if you are interested, you can investigate after you've completed the lab.

→ My Decision Tree is shown in Figure 3.

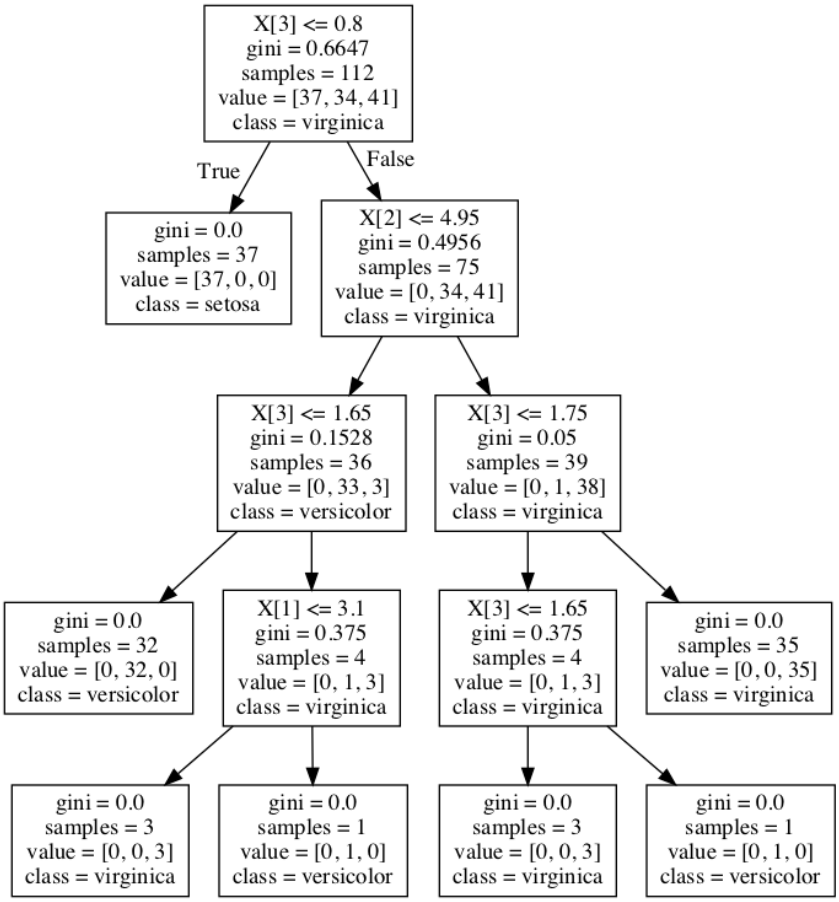


Figure 3: Sample Decision Tree on the Iris data set (drawn with dot). If you call the export_graphviz() function with the argument impurity=False, then the gini values will not be included.

5 Constructing a Logistic Regression Classifier

Build a **logistic regression** model using **sklearn.linear_model.LogisticRegression()**.

See:
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression
for information.

To keep things simple, we'll again only work with two attributes in the Iris data set: *sepal length* and *sepal width*, which are the 0-th and 1-st columns in the data.

→ Write a Python script to load the data, extract the first two columns and then initialise and fit the Logistic Regression model, as shown in Figure 4.

```
import sklearn.datasets as data
import sklearn.model_selection as model_select
import sklearn.linear_model as linear_model

# load iris data
iris = data.load_iris()

# we'll only look at two dimensions (0th and 1st)
X = iris.data[:,:2]
y = iris.target
X_train, X_test, y_train, y_test = model_select.train_test_split( X[:,:2], y,
    random_state=0 )

# initialise logistic regression model
clf = linear_model.LogisticRegression( solver='lbfgs', multi_class='multinomial'
    )

# fit model to iris data
clf.fit( X_train, y_train )
```

Figure 4: Sample code to run the **sci-kit** Logistic Regression classifier on the Iris data set

5.1 How good is your model?

As with the previous models, you should start by computing the score for your classifier. You can compute the **training score**, **test score**, **accuracy score**, **confusion matrix**, **precision**, **recall** and **F1 score** just like you did with the Decision Tree. My results are shown in Figure 5. As with the decision tree, the *accuracy score* computed by the metrics package is the same as the *score* on the test data computed by the classifier package. It doesn't matter which method you use—this just shows you that they are the same thing.

5.2 What do the decision boundaries look like?

As mentioned earlier and discussed in class, a Logistic Regression model computes decision boundaries. You can plot these boundaries.

Imagine that you have a 2D space that covers all the possible values of each attribute, *sepal length* and *sepal width*. The minimum and maximum values for each are listed below:

```
training score =  0.839285714286
test score =  0.789473684211
accuracy score =  0.789473684211
confusion matrix =
           predicted-->
   actual:     setosa versicolor  virginica
    setosa         13          0          0
versicolor          0         11          5
 virginica          0          3          6

precision score = tp / (tp + fp) =
  setosa = 1.000000
  versicolor = 0.785714
  virginica = 0.545455
recall score = tp / (tp + fn) =
  setosa = 1.000000
  versicolor = 0.687500
  virginica = 0.666667
f1 score = 2 * (precision * recall) / (precision + recall) =
  setosa = 1.000000
  versicolor = 0.733333
  virginica = 0.600000
```

Figure 5: Sample results on Logistic Regression with Iris data set

	minimum	*maximum*
sepal length	4.3	7.9
sepal width	2.0	4.4

You need to generate a 2×2 grid or *mesh* that covers this limits. The size of the granularity of the grid is up to you. A smaller granularity will produce a smoother decision boundary. Let's choose a granularity of 0.1. This means that you will generate a grid with coordinates starting at $(4.3, 2.0)$ and incrementing by $(0.1, 0.1)$ until you reach $(7.9, 4.4)$. I actually started my grid one step (0.1) below the minimum and ended one step above the maximum, to make sure that the boundaries extend beyond the edges of the data set, i.e., so my grid goes from $(4.2, 1.9)$ to $(8.0, 4.5)$. This means that my grid is $(8.0 - 4.2)/0.1 = 38$ by $(4.5 - 1.9)/0.1 = 26$, so I have $38 \times 26 = 988$ coordinate pairs in my grid.

For each of the points on this grid, you need to use the classifier to predict a value, like this:

```
1 y_hat_pairs = clf.predict( X_pairs )
```

where `y_hat_pairs` is the set of classes predicted for each of the coordinate pairs. Note that the `X_pairs` argument needs to be a 2-dimensional array of shape (988,2) in order to be handled correctly by the `clf.predict()` function. (The value of 988 in the first dimension is due to the granularity of my grid; if I had a finer granularity, say with a step size of 0.001, then this dimension would be larger.)

You can compute the *score* for these predictions:

```
print 'mesh score = ', clf.score( X_pairs, y_hat_pairs )
```

and you should get `mesh score = 1.0`, because the mesh contains all the values in our training set and contains all the right answers.

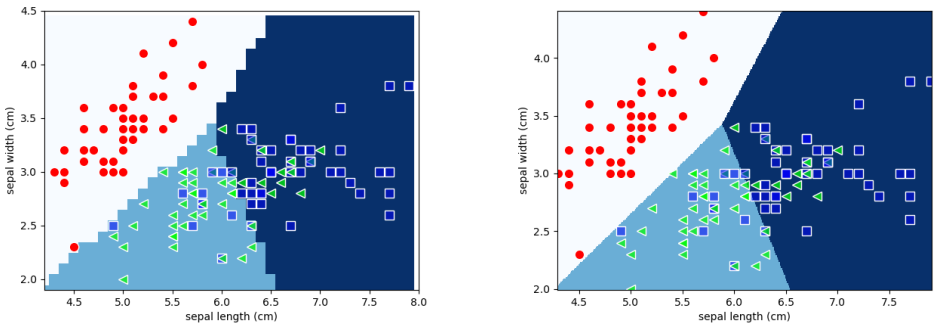
In order to plot the decision boundaries, you need to create a mesh that you can plot:

```
import numpy as np
import matplotlib.pyplot as plt

# generate a 2D surface (mesh) that covers all possible combinations
# of the two attribute value ranges we just initialised above so that
# we can use it below to colour the decision boundaries.
x0_mesh, x1_mesh = np.meshgrid( x0_range, x1_range )

y_hat_mesh = y_hat_pairs.reshape( x0_mesh.shape )
# x0_mesh and x1_mesh are the coordinates of the quadrilateral corners
# for each cell in the colour mesh and y_hat_mesh contains the
# corresponding class for that coordinate pair, which is used to set
# the colour for that quadrilateral.
plt.pcolormesh( x0_mesh, x1_mesh, y_hat_mesh, shading='flat' )
```

My code, above, uses two arrays: `x0_range` and `x1_range`. These are arrays I created earlier in order to facilitate generation of the `X_pairs` array of coordinates, for example:

```
x0_range = np.arange( x0_min, x0_max, STEP_SIZE )
```

The `plt.pcolormesh()` function generates the plot of the mesh. Essentially, this is a 2-dimensional surface of rectangles, each one 0.1×0.1 in size and filled with the colour that corresponds to the class. Remember that the class labels are represented as $[0, 1, 2]$, so this just takes the 0-th, 1-st and 2-nd entries in the current colour map and uses those to fill the rectangles accordingly. In my code, I used `plt.set_cmap( 'Blues' )` to apply the "Blues" colour map. The plot of my decision boundaries are shown in Figure 6, with two different granularities. You can see that the smaller granularity of 0.01 (Figure 6b) has a smoother boundary than the plot showing granularity of 0.1. Note that I also plotted the points in the data set, using different coloured and shape markers for each class, so it is easy to see visually how well the boundaries separate the data.



(a) granularity $= 0.1$ (b) granularity $= 0.01$

Figure 6: Sample Decision Boundaries for Logistic Regression on Iris data set

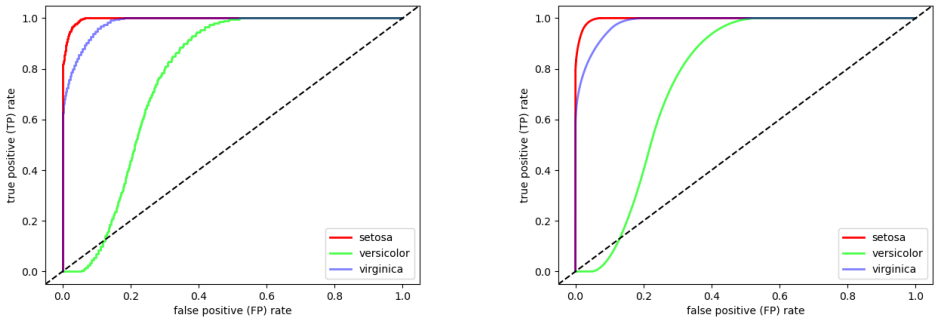
5.3 How good are the decision boundaries?

Finally, you can create a **Receiver Operating Characteristic (ROC)** curves to score how well the decision boundaries perform for each class. The code in Figure 7 illustrates how to do this. The comments explain how the **scikit-learn** functions work.

```
1  import sklearn.preprocessing as preprocess
2  import sklearn.metrics as metrics
3
4  # we can compute a "decision function" which gives "confidence scores"
5  # corresponding to the samples. this is the signed distance from each
6  # value in X to the decision boundary. we can find the furthest from the boundary
7  conf_scores = clf.decision_function( X_pairs )
8
9  # binarize the output, since we have a multi-class data set
10 y_binary = preprocess.label_binarize( y_hat_pairs, classes=sorted( set( y )) )
11
12 # compute ROC curve
13 fpr = dict() # false positive rates, for each class
14 tpr = dict() # true positive rates, for each class
15 for c in range( num_classes ):
16     fpr[c], tpr[c], tmp = metrics.roc_curve( y_binary[:,c], conf_scores[:,c] )
17 # plot curves for each class
18 for c in range( num_classes ):
19     plt.plot( fpr[c], tpr[c], label=iris.target_names[c] )
20 plt.xlabel( 'false positive (FP) rate' )
21 plt.ylabel( 'true positive (TP) rate' )
```

Figure 7: Sample code to plot ROC curve

Note that the Iris data set is called a **multi-class** data set, because there are not just two classes (there are three). We compute an ROC curve for each of the three classes. In order to do this, we have to create a binary representation of the class labels. For example, if we have $y[0] = 0$, meaning that the target value for the first instance is the class *setosa*, then we can create a binary representation that looks like this: $y\_binary[0] = [1,0,0]$, meaning that the instance belongs to the first class ("true" or 1) and the instance does not belong to either of the other two classes ("false" or 0). This step is performed on line 10 of the sample code. My ROC curves are shown in Figure 8.



(a) granularity $= 0.1$ (b) granularity $= 0.01$

Figure 8: ROC Curves for Logistic Regression on Iris data set