CS 3430: SciComp with Py
Assignment 11
Building Single Feature Classifiers for the IRIS Data Sets

Vladimir Kulyukin
Department of Computer Science
Utah State University

April 7, 2018

1 Learning Objectives

1. IRIS Data Set
2. Single Feature Classifiers
3. Cross Validation

2 Introduction

In this assignment, you will play with the IRIS data set and build several single feature classifiers and evaluate them using a cross-validation technique. This assignment will generalize the material covered in the last two lectures and give you a deeper appreciation of the IRIS data set.

3 Creating Boolean Indexes

Let's start by creating three boolean indexes we'll use in this assignment. Start by loading the IRIS data set and creating several numpy arrays.

```
from sklearn.datasets import load_iris
data = load_iris()
flowers = data.data
feature_names = data.feature_names
target = data.target
target_names = data.target
```

Recall that the numpy array `data` contains a 150x4 matrix where each row contains a 4 element array of feature values. We can get a couple of those flower feature vectors using the standard array indexing.

```
>>> flowers[0]
array([ 5.1,  3.5,  1.4,  0.2])
>>> flowers[149]
array([ 5.9,  3. ,  5.1,  1.8])
```

If we display the feature names, we can correlate these feature values with their features.

```
>>> feature_names
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

Thus, the flower `flowers[0]` has a sepal length of 5.1cm, a sepal width of 3.5cm, a petal length of 1.4cm, and a petal width of 0.2cm, and the flower `flowers[1]` has a sepal length of 5.9cm, a sepal width of 3.0cm, a petal length of 5.1cm, and a petal width of 1.8cm.

The numpy array `target` contains 150 integer values. The first 50 values are 0's, the second 50 values are 1's, and the third 50 values are 2's. The integers 0, 1, and 2 are target numbers, i.e., numerical labels of flower names.

```
>>> target
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Let’s figure out which target labels correspond to which target names by displaying the values of `target_names`.

```
>>> target_names
array(['setosa', 'versicolor', 'virginica'], dtype='|S10')
```

Thus, the target value `0` corresponds to the `'setosa'` label, the target value `1` to the label `'versicolor'`, and the target value `2` to the label `'virginica'`.

Another way to map the target numbers to the target names is to use the following useful numpy trick that maps numbers to strings. Let’s define a numpy array of `names`.

```
>>> names = np.array(['A', 'B', 'C', 'D'])
array(['A', 'B', 'C', 'D'], dtype='|S1')
```

In `names`, `A` is label 0, `B` is label 1, `C` is label 2, `D` is label 3. Let’s create an array of numerical labels.

```
>>> labels = np.array([0, 0, 2, 2, 1, 3, 3, 0, 1])
```

If we want to map these numerical labels to their symbolic names, all we need to do is to use `labels` to index into `names`.

```
>>> names_of_labels = names[labels]
>>> names_of_labels
array(['A', 'A', 'C', 'C', 'B', 'D', 'D', 'A', 'B'], dtype='|S1')
```

In other words, in `names_of_labels`, all 0’s in `labels` are mapped to `A`, all 1’s in `labels` are mapped to `B`, all 2’s in `labels` are mapped to `C`, and all 3’s in `labels` are mapped to `D`.

We can use this numpy trick to get the string names of all the flowers in the IRIS data set by mapping the numerical values in `target` to `target_names`.

```
>>> flower_names = target_names[target]
>>> flower_names
array(['setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'versicolor',
       'versicolor', 'versicolor', 'versicolor', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
```

```
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica', 'virginica', 'virginica',
       'virginica', 'virginica', 'virginica'],
      dtype='|S10')
```

We can immediately see that in `flower_names` the first 50 items are `'setosa'`, the second 50 items are `'versicolor'`, and the third 50 items are `'virginica'`.

Let's create three boolean indexes.

```
>>> is_setosa     = (flower_names == 'setosa')
>>> is_virginica  = (flower_names == 'virginica')
>>> is_versicolor = (flower_names == 'versicolor')
```

In `is_setosa`, the first 50 values are `True` and the remaining 100 values are `False`. Let's check out if this is true.

```
>>> is_setosa
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False], dtype=bool)
```

In `is_virginica`, the first 100 values are `False`, the remaining 50 values are `True`. Let's verify this.

```
>>> is_virginica
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True], dtype=bool)
```

In `is_versicolor`, the first 50 values are `False`, the second 50 values are `True`, and the third 50 values are `False`. Let's see if it's true.

```
>>> is_versicolor
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
```

```
       False, False, False, False, False,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False], dtype=bool)
```

We can use the boolean indexes to retrieve all flower items of a particular type. For example, if we want to get all the setosas, we can do so as follows:

```
>>> setosas = flowers[is_setosa]
```

We can check if the array `setosas` is equal to the array of the first 50 elements in `flowers`.

```
>>> np.array_equal(setosas, flowers[0:50])
True
```

We can do the same for versicolors and virginicas.

```
>>> virginicas = flowers[is_virginica]
>>> versicolors = flowers[is_versicolor]
>>> np.array_equal(virginicas, flowers[100:])
True
>>> np.array_equal(versicolors, flowers[50:100])
True
```

4 Computing Model's Accuracy

Our objective is to create three models: one model for each flower. In the context of this assignment, the term *model* refers to a single feature classifier. When applied to an array of flowers, the model returns a 1D numpy array of boolean values. Suppose we have learned a setosa model. If we apply this model to an array of flowers, it returns an array of boolean values where each `True` denotes that the corresponding flower in the array of flowers is classified as `setosa` and each `False` is classified as not `setosa`.

In ML terminology, the array of values returned by a model is called *predictions*. To access the accuracy of the model, the model's predictions are compared against the true values. This array of the true values is called the *ground truth*. For example, the ground truth for setosas is given in `is_setosa`.

Implement the function `compute_model_accuracy` that takes the numpy array of predictions and the numpy array of ground truth values and returns the percentage of predictions that coincide with the ground truth values.

```
def compute_model_accuracy(predictions, ground_truth):
    pass
```

Let me show you about a few numpy tricks that may help you implement this function with just a few lines of code. You can use the predicate `==` on the two numpy arrays to compute which corresponding values in the two arrays are equal and which are not. Let's define two arrays.

```
>>> predictions = np.array([True, False, False, True])
>>> ground_truth = np.array([False, False, True, True])
```

Then we can use `==` to obtain a boolean array of comparisons.

```
>>> predictions == ground_truth
array([False,  True, False,  True], dtype=bool)
```

This means that the two corresponding values at index 0 are not the same, the two values at index 1 are the same, the two values at index 2 are not the same, and the two values at index 3 are the same. If we want to count the number of the corresponding values that are equal, we can use the numpy sum function that, when applied to a boolean array, counts the number of `True` values in it.

```
>>> np.sum(predictions == ground_truth)
2
```

5 Learning The Best Threshold Models

Recall that we developed several single feature classifiers by analytically looking at the feature plots and deciding on the best feature and its thresholds. Let's generalize our reasoning by implementing the function `learn_best_th_model_for`.

```
def learn_best_th_model_for(flower_name, flowers, bool_index):
    assert len(flowers) == len(bool_index)
    pass
```

This function takes the name of the flower, e.g., `'virginica'`, the array of flowers, i.e., `data.data`, and a boolean index `bool_index` to retrieve specific flowers from `flowers` as `flowers[bool_index]`. Let's call `flowers[bool_index]` the selected flowers.

This function goes through each feature `fn` and selects all possible thresholds for `fn` by selecting the values of `fn` in the selected flowers.

For each possible threshold `pt` in the possible thresholds for `fn`, it computes the percentage of the flowers of the type specified by the first argument for which this value is strictly above the threshold. This value is the accuracy of the direct model.

The function also computes the percentage of the flowers that are *not* of the type specified by the first argument for which the feature value is strictly above the threshold. This is called the accuracy of the reverse model.

After going through all features and all possible thresholds the function returns a 4-tuple: `best_fn`, `best_th`, `reverse`, and `best_acc`, where `best_fn` is `fn` that gives the best accuracy, `best_th` is the best threshold for `best_fn`, `reverse` is `True` if the best accuracy specified by the fourth returned value is the accuracy of the reverse model and `False` otherwise, and `best_accuracy` is the best accuracy.

Let's define and run three unit tests.

```
def unit_test_01():
    '''learn single feature classifier for setosa'''
    setosa_model = learn_best_th_model_for('setosa', flowers,
                                           is_setosa)
    print 'setosa model:', setosa_model

def unit_test_02():
    '''learn single feature classifier for virginica'''
    virginica_model = learn_best_th_model_for('virginica', flowers,
                                              is_virginica)
    print 'virginica model:', virginica_model

def unit_test_03():
    '''learn single feature classifier for versicolor'''
    versicolor_model = learn_best_th_model_for('versicolor', flowers,
                                               is_versicolor)
    print 'versicolor model:', versicolor_model

if __name__ == '__main__':
     unit_test_01()
     unit_test_02()
     unit_test_03()
```

Here is my output.

```
setosa model: (2, 1.8999999999999999, True, 1.0)
virginica model: (3, 1.7, False, 0.95999999999999996)
versicolor model: (1, 2.8999999999999999, True, 0.73999999999999999)
```

Now define the function `run_model`

```
def run_model(model, flowers):
    pass
```

This function takes a model returned by `learn_best_th_model_for`, applies the model to each flower in `flowers`, and returns an array of predictions. Let's define and run a few more tests.

```
def unit_test_04():
    '''learn and run single feature classifier for setosa'''
    model = learn_best_th_model_for('setosa', flowers, is_setosa)
    predictions = run_model(model, flowers)
```

```
    print 'setosa model acc:', compute_model_accuracy(predictions, is_setosa)

def unit_test_05():
    '''learn and run single feature classifier for virginica'''
    model = learn_best_th_model_for('virginica', flowers, is_virginica)
    predictions = run_model(model, flowers)
    print 'virginica model acc:', compute_model_accuracy(predictions, is_virginica)

def unit_test_06():
    '''learn and run single feature classifier for versicolor'''
    model = learn_best_th_model_for('versicolor', flowers, is_versicolor)
    predictions = run_model(model, flowers)
    print 'versicolor model acc:', compute_model_accuracy(predictions, is_versicolor)

if __name__ == '__main__':
     unit_test_04()
     unit_test_05()
     unit_test_06()
     unit_test_07()
     unit_test_08()
```

Here is my output.

```
setosa model acc: 1.0
virginica model acc: 0.96
versicolor model acc: 0.74
```

6 Cross Validation

In this assignment, we'll focus on a simple type of cross validation called `leave-one-out`. The basic idea is to take a sample (i.e., a flower feature vector in our case) out of the training data (i.e., the numpy array of 150 flower feature vectors), learn a model without this sample, and then see if the learned model classifies the removed sample correctly.

In our case, the model is learned/trained with `learn_best_th_model_for` implemented in the previous section. The `leave-one-out` method is an extreme case of cross validation, because the fold that we remove from the training data contains exactly 1 sample.

Specifically, the method goes through each flower feature vector in the array of 150 flower feature vectors. Each feature vector is removed from the array of the flower feature vectors, and the model is learned on the remaining 149 flower feature vectors. The learned model is then applied to the removed flower feature vector. If the model classifies the removed flower correctly, the accuracy count for the removed flower's type is incremented by 1. If the model does not classify the removed flower correctly, the accuracy count for the removed flower's type remains the same. In the end, the percentages are computed and displayed for each flower type. Note that on the IRIS data set, the `leave-one-out` method learns 150 models. Implement the function `leave_one_out_cross_validation` to do the `leave-one-out` cross validation on the IRIS data set.

```
def leave_one_out_cross_validation(flower_name, flowers):
    pass
```

This function takes a flower name and an array of flowers and returns the accuracy obtained by running the `leave-one-out` cross validation on the flowers of the type specified by the given flower name. Let's define and run a few unit tests.

```
def unit_test_07():
    '''run leave-one-out cross-validation for setosas'''
    acc = leave_one_out_cross_validation('setosa', flowers)
    print 'leave-1-out cross_valid acc for setosa:', acc

def unit_test_08():
    '''run leave-one-out cross-validation for virginicas'''
    acc = leave_one_out_cross_validation('virginica', flowers)
    print 'leave-1-out cross_valid acc for virginica:', acc

def unit_test_09():
    '''run leave-one-out cross-validation for versicolors'''
    acc = leave_one_out_cross_validation('versicolor', flowers)
    print 'leave-1-out cross_valid acc for versicolor:', acc
```

Below are my `leave-one-out` cross validation accuracies.

```
leave-1-out cross_valid acc for setosa: 1.0
leave-1-out cross_valid acc for virginica: 0.966666666667
leave-1-out cross_valid acc for versicolor: 0.733333333333
```

7 What To Submit

Implement all functions in `leave_one_out_cross_validation.py` and submit this file through Canvas.

Happy Hacking!