kNN_Classification_professor

September 19, 2019

1 The k-Nearest Neighbor Classification Algorithm

```
Notebook version: 2.1 (Oct 19, 2018)

Author: Jesús Cid Sueiro (jcid@tsc.uc3m.es)
        Jerónimo Arenas García (jarenas@tsc.uc3m.es)

Changes: v.1.0 - First version
         v.1.1 - Function loadDataset updated to work with any number of dimensions
         v.2.0 - Compatible with Python 3 (backcompatible with Python 2.7)
                 Added solution to Exercise 3
         v.2.1 - Minor corrections regarding notation
```

[ ]:
```
from __future__ import print_function
# To visualize plots in the notebook
%matplotlib inline

# Import some libraries that will be necessary for working with data and␣
 ↪displaying plots
import csv     # To read csv files
import random
import matplotlib.pyplot as plt
import numpy as np
from scipy import spatial
from sklearn import neighbors, datasets
```

1.1 1. The binary classification problem.

In a binary classification problem, we are given an observation vector $\mathbf{x} \in \mathbb{R}^N$ which is known to belong to one and only one *category* or *class*, y, in the set $\mathcal{Y} = \{0, 1\}$. The goal of a classifier system is to predict the value of y based on $\mathbf{x}$.

To design the classifier, we are given a collection of labelled observations $\mathcal{D} = \{(\mathbf{x}^{(k)}, y^{(k)})\}_{k=0}^{K-1}$ where, for each observation $\mathbf{x}^{(k)}$, the value of its true category, $y^{(k)}$, is known. All samples are outcomes of an unknown distribution $p_{\mathbf{X},Y}(\mathbf{x}, y)$.

1.2 2. The Iris dataset

(Iris dataset presentation is based on this Tutorial by Jason Brownlee)

To illustrate the algorithms, we will consider the Iris dataset , taken from the UCI Machine Learning repository . Quoted from the dataset description:

> This is perhaps the best known database to be found in the pattern recognition literature. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. [...] One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

The *class* is the species, which is one of *setosa*, *versicolor* or *virginica*. Each instance contains 4 measurements of given flowers: sepal length, sepal width, petal length and petal width, all in centimeters.

[ ]:
```
# Taken from Jason Brownlee notebook.
with open('datasets/iris.data', 'r') as csvfile:
        lines = csv.reader(csvfile)
        for row in lines:
                print(', '.join(row))
```

1.2.1 2.1. Train/test split

Next, we will split the data into a training dataset, that will be used to learn the classification model, and a test dataset that we can use to evaluate its the accuracy.

We first need to convert the flower measures that were loaded as strings into numbers that we can work with. Next we need to split the data set **randomly** into train and datasets. A ratio of 67/33 for train/test will be used.

The code fragment below defines a function `loadDataset` that loads the data in a CSV with the provided filename and splits it randomly into train and test datasets using the provided split ratio.

[ ]:
```
# Adapted from a notebook by Jason Brownlee
def loadDataset(filename, split):
    xTrain = []
    cTrain = []
    xTest = []
    cTest = []

    with open(filename, 'r') as csvfile:
        lines = csv.reader(csvfile)
        dataset = list(lines)
    for i in range(len(dataset)-1):
        for y in range(4):
            dataset[i][y] = float(dataset[i][y])
        item = dataset[i]
        if random.random() < split:
            xTrain.append(item[0:-1])
            cTrain.append(item[-1])
        else:
            xTest.append(item[0:-1])
            cTest.append(item[-1])
    return xTrain, cTrain, xTest, cTest
```

We can use this function to get a data split. Note that, because of the way samples are assigned to the train or test datasets, the number of samples in each partition will differ if you run the code several times.

[ ]:
```
xTrain_all, cTrain_all, xTest_all, cTest_all = loadDataset('datasets/iris.data',␣
 ↪0.67)
nTrain_all = len(xTrain_all)
nTest_all = len(xTest_all)
print('Train:', nTrain_all)
print('Test:', nTest_all)
```

1.2.2 2.2. Versicolor vs Virginica

In the following, we will design a classifier to separate classes "Versicolor" and "Virginica" using x_0 and x_1 only. To do so, we build a training set with samples from these categories, and a bynary label $y^{(k)} = 1$ for samples in class "Virginica", and 0 for "Versicolor" data.

[ ]:
```
# Select two classes
c0 = 'Iris-versicolor'
c1 = 'Iris-virginica'

# Select two coordinates
ind = [0, 1]

# Take training test
X_tr = np.array([[xTrain_all[n][i] for i in ind] for n in range(nTrain_all)
                  if cTrain_all[n]==c0 or cTrain_all[n]==c1])
C_tr = [cTrain_all[n] for n in range(nTrain_all)
          if cTrain_all[n]==c0 or cTrain_all[n]==c1]
Y_tr = np.array([int(c==c1) for c in C_tr])
n_tr = len(X_tr)

# Take test set
X_tst = np.array([[xTest_all[n][i] for i in ind] for n in range(nTest_all)
                 if cTest_all[n]==c0 or cTest_all[n]==c1])
C_tst = [cTest_all[n] for n in range(nTest_all)
         if cTest_all[n]==c0 or cTest_all[n]==c1]
Y_tst = np.array([int(c==c1) for c in C_tst])
n_tst = len(X_tst)

# Separate components of x into different arrays (just for the plots)
x0c0 = [X_tr[n][0] for n in range(n_tr) if Y_tr[n]==0]
x1c0 = [X_tr[n][1] for n in range(n_tr) if Y_tr[n]==0]
x0c1 = [X_tr[n][0] for n in range(n_tr) if Y_tr[n]==1]
x1c1 = [X_tr[n][1] for n in range(n_tr) if Y_tr[n]==1]
```

```python
# Scatterplot.
labels = {'Iris-setosa': 'Setosa',
          'Iris-versicolor': 'Versicolor',
          'Iris-virginica': 'Virginica'}
plt.plot(x0c0, x1c0,'r.', label=labels[c0])
plt.plot(x0c1, x1c1,'g+', label=labels[c1])
plt.xlabel('$x_' + str(ind[0]) + '$')
plt.ylabel('$x_' + str(ind[1]) + '$')
plt.legend(loc='best')
plt.show()
```

1.3 3. Baseline Classifier: Maximum A Priori.

For the selected data set, we have two clases and a dataset with the following class proportions:

[ ]:
```python
print('Class 0 (' + c0 + '): ' + str(n_tr - sum(Y_tr)) + ' samples')
print('Class 1 (' + c1 + '): ' + str(sum(Y_tr)) + ' samples')
```

The maximum a priori classifier assigns any sample **x** to the most frequent class in the training set. Therefore, the class prediction y for any sample **x** is

[ ]:
```python
y = int(2*sum(Y_tr) > n_tr)
print('y = ' + str(y) + ' (' + (c1 if y==1 else c0) + ')')
```

The error rate for this baseline classifier is:

[ ]:
```python
# Training and test error arrays
E_tr = (Y_tr != y)
E_tst = (Y_tst != y)

# Error rates
pe_tr = float(sum(E_tr)) / n_tr
pe_tst = float(sum(E_tst)) / n_tst
print('Pe(train):', pe_tr)
print('Pe(test):', pe_tst)
```

The error rate of the baseline classifier is a simple benchmark for classification. Since the maximum a priori decision is independent on the observation, **x**, any classifier based on **x** should have a better (or, at least, not worse) performance than the baseline classifier.

1.4 4. The Nearest-Neighbour Classifier (1-NN).

The 1-NN classifier assigns any instance **x** to the category of the nearest neighbor in the training set.

$$d = f(\mathbf{x}) = y^{(n)}, \text{ where} n = \arg\min_k \|\mathbf{x} - \mathbf{x}^{(k)}\|$$

In case of ties (i.e. if there is more than one instance at minimum distance) the class of one of them, taken arbitrarily, is assigned to **x**.

[ ]:
```python
def nn_classifier(X1,Y1,X2):
    """ Compute the 1-NN classification for the observations contained in
```

```
        the rows of X2, for the training set given by the rows in X1 and the
        class labels contained in Y1.
    """
    if X1.ndim == 1:
        X1 = np.asmatrix(X1).T
    if X2.ndim == 1:
        X2 = np.asmatrix(X2).T
    distances = spatial.distance.cdist(X1,X2,'euclidean')
    neighbors = np.argsort(distances, axis=0, kind='quicksort', order=None)
    closest = neighbors[0,:]
    y_values = np.zeros([X2.shape[0],1])
    for idx in range(X2.shape[0]):
        y_values[idx] = Y1[closest[idx]]

    return y_values
```

Let us apply the 1-NN classifier to the given dataset. First, we will show the decision regions of the classifier. To do so, we compute the classifier output for all points in a rectangular grid from the sample space.

[ ]:
```
# Create a regtangular grid.
n_points = 200
x_min, x_max = X_tr[:, 0].min(), X_tr[:, 0].max()
y_min, y_max = X_tr[:, 1].min(), X_tr[:, 1].max()
dx = x_max - x_min
dy = y_max - y_min
h = dy / n_points
xx, yy = np.meshgrid(np.arange(x_min - 0.1 * dx, x_max + 0.1 * dx, h),
                     np.arange(y_min - 0.1 * dx, y_max + 0.1 * dy, h))
X_grid = np.array([xx.ravel(), yy.ravel()]).T

# Compute the classifier output for all samples in the grid.
Z = nn_classifier(X_tr, Y_tr, X_grid)

# Put the result into a color plot
plt.plot(x0c0, x1c0,'r.', label=labels[c0])
plt.plot(x0c1, x1c1,'g+', label=labels[c1])
plt.xlabel('$x_' + str(ind[0]) + '$')
plt.ylabel('$x_' + str(ind[1]) + '$')
plt.legend(loc='best')

Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z)
plt.show()
```

We can observe that the decision boudary of the 1-NN classifier is rather intricate, and it may contain small *islands* covering one or few samples from one class. Actually, the extension of these small regions usually reduces as we have more training samples, though the number of them may increase.

Now we compute the error rates over the training and test sets.

[ ]:
```
# Training errors
Z_tr = nn_classifier(X_tr, Y_tr, X_tr)
E_tr = Z_tr.flatten()!=Y_tr

# Test errors
Z_tst = nn_classifier(X_tr, Y_tr, X_tst)
E_tst = Z_tst.flatten()!=Y_tst

# Error rates
pe_tr = float(sum(E_tr)) / n_tr
pe_tst = float(sum(E_tst)) / n_tst
print('Pe(train):', pe_tr)
print('Pe(test):', pe_tst)
```

The training and test error rates of the 1-NN may be significantly different. In fact, the training error may go down to zero if samples do not overlap. In the selected problem, this is not the case, because samples from different classes coincide at the same point, causing some classification errors.

Consistency of the 1-NN classifier

Despite the 1-NN usually reduces the error rate with respect to the baseline classifier, the number of errors may be too large. Errors may be attributed to diferent causes:

1. The class distributions are overlapped, because the selected features have no complete information for discriminating between the classes: this would imply that, even the best possible classifier would be prone to errors.
2. The training sample is small, and it is not enough to obtaing a good estimate of the optimal classifiers.
3. The classifier has intrinsic limitations: even though we had an infinite number of samples, the classifier performance does not approach the optimal classifiers.

In general, a classifier is said to be consistent if it makes nearly optimal decisions as the number of training samples increases. Actually, it can be shown that this is the case of the 1-NN classifier if the classification problem is separable, i.e. if there exist a decision boundary with zero error probability. Unfortunately, in a non-separable case, the 1-NN classifier is not consistent. It can be shown that the error rate of the 1-NN classifier converges to an error rate which is not worse than twice the minimum attainable error rate (Bayes error rate) as the number of training samples goes to infinity.

Exercise 1: In this exercise we test the non-consistency of the 1-NN classifier for overlapping distributions. Generate an artificial dataset for classification as follows:

- Generate N binary labels at random with values '0' and '1'. Store them in vector $\mathbf{y}$
- For every label $y^{(k)}$ in $\mathbf{y}$:
  - If the label is 0, take sample $x^{(k)}$ at random from a uniform distribution $U(0,2)$.
  - If the label is 1, take sample $x^{(k)}$ at random from a uniform distribution $U(1,5)$.

Take $N = 1000$ for the test set. This is a large sample to get accurate error rate estimates. Also, take $N = 10, 20, 40, 80, ...$ for the training set. Compute the 1-NN classifier, and observe the test error rate as a function of N.

Now, compute the test error rate of the classifier making decision 1 if $x^{(k)} > 1.5$, and 0 otherwise.

[ ]:
```python
# <SOL>
from sklearn.neighbors import KNeighborsClassifier

Ntest = 10000
Ntr = [10, 20, 40, 80, 200, 1000]
nruns = 100

xtest = []
ytest = []

for k in range(Ntest):
    if k<Ntest/2:
        ytest.append(0)
        xtest.append([2*np.random.random()])
    else:
        ytest.append(1)
        xtest.append([1+4*np.random.random()])

#print(np.mean(np.array(ytest)))
#print(xtest)

Etest = np.zeros((len(Ntr),))

for k in range(nruns):
    for kk,Ntrain in enumerate(Ntr):
        xtrain = []
        ytrain = []
        for k in range(Ntrain):
            if k<Ntrain/2:
                ytrain.append(0)
                xtrain.append([2*np.random.random()])
            else:
                ytrain.append(1)
                xtrain.append([1+4*np.random.random()])

        #Entreno el clasificador y obtengo predicciones para ytest
        neigh = KNeighborsClassifier(n_neighbors=1)
        ytest_pred = neigh.fit(xtrain,ytrain).predict(xtest)
        tasa_error = np.mean(np.array(ytest)!=ytest_pred)

        Etest[kk] += tasa_error/nruns

print(Etest)
# </SOL>
```

1.5 5. *k*-NN classifier

A simple extension of the 1-NN classifier is the *k*-NN classifier, which, for any input sample **x**, computes the *k* closest neighbors in the training set, and takes the majority class in the subset. To avoid ties, in the binary classification case *k* is usually taken as an odd number.

[ ]:
```
def knn_classifier(X1,Y1,X2,k):
    """ Compute the k-NN classification for the observations contained in
        the rows of X2, for the training set given by the rows in X1 and the
        components of S1. k is the number of neighbours.
    """
    if X1.ndim == 1:
        X1 = np.asmatrix(X1).T
    if X2.ndim == 1:
        X2 = np.asmatrix(X2).T
    distances = spatial.distance.cdist(X1,X2,'euclidean')
    neighbors = np.argsort(distances, axis=0, kind='quicksort', order=None)
    closest = neighbors[range(k),:]

    y_values = np.zeros([X2.shape[0],1])
    for idx in range(X2.shape[0]):
        y_values[idx] = np.median(Y1[closest[:,idx]])

    return y_values
```

[ ]:
```
k = 7

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, m_max]x[y_min, y_max].
Z = knn_classifier(X_tr, Y_tr, X_grid, k)

# Put the result into a color plot
plt.plot(x0c0, x1c0,'r.', label=labels[c0])
plt.plot(x0c1, x1c1,'g+', label=labels[c1])
plt.xlabel('$x_' + str(ind[0]) + '$')
plt.ylabel('$x_' + str(ind[1]) + '$')
plt.legend(loc='best')

Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z)
plt.show()
```

[ ]:
```
# Plot training and test error as a function of parameter k.
pe_tr = []
pe_tst = []
k_list = [2*n+1 for n in range(int(np.floor(n_tr/2)))]

for k in k_list:
    # Training errors
    Z_tr = knn_classifier(X_tr, Y_tr, X_tr, k)
    E_tr = Z_tr.flatten()!=Y_tr

    # Test errors
    Z_tst = knn_classifier(X_tr, Y_tr, X_tst, k)
    E_tst = Z_tst.flatten()!=Y_tst

    # Error rates
    pe_tr.append(float(sum(E_tr)) / n_tr)
    pe_tst.append(float(sum(E_tst)) / n_tst)

# Put the result into a color plot
markerline, stemlines, baseline = plt.stem(k_list, pe_tr,'r', markerfmt='s',␣
 ↪label='Training')
plt.plot(k_list, pe_tr,'r:')
plt.setp(markerline, 'markerfacecolor', 'r', )
plt.setp(baseline, 'color','r', 'linewidth', 2)
markerline, stemlines, baseline = plt.stem(k_list, pe_tst, label='Test')
plt.plot(k_list, pe_tst,'b:')
plt.xlabel('$k$')
plt.ylabel('Error rate')
plt.legend(loc='best')
plt.show()
```

We can analyze the influence of parameter $k$ by observing both traning and test errors.

Exercise 2: Observe the train and test error for large $k$. Could you relate the error rate of the baseline classifier with that to the $k$-NN for certain value of $k$?

The figure above suggests that the optimal value of $k$ is

[ ]:
```
i = np.argmin(pe_tst)
k_opt = k_list[i]
print('k_opt:', k_opt)
```

However, using the test set to select the optimal value of the hyperparameter $k$ is not allowed. Instead, we should recur to cross validation.

1.6 2.3 Hyperparameter selection via cross-validation

An inconvenient of the application of the $k$-nn method is that the selection of $k$ influences the final error of the algorithm. In the previous experiments, we noticed that the location of the minimum is not necessarily the same from the perspective of the test and training data. Ideally, we would like that the designed classification model works as well as possible on future unlabeled patterns that are not available during the training phase. This property is known as generalization. Fitting the training data is only pursued in the hope that we are also indirectly obtaining a model that generalizes well. In order to achieve this goal, there are some strategies that try to guarantee a correct generalization of the model. One of such approaches is known as cross-validation.

Since using the test labels during the training phase is not allowed (they should be kept aside to simultate the future application of the classification model on unseen patterns), we need to figure out some way to improve our estimation of the hyperparameter that requires only training data. Cross-validation allows us to do so by following the following steps:

- Split the training data into several (generally non-overlapping) subsets. If we use M subsets, the method is referred to as M-fold cross-validation. If we consider each pattern a different subset, the method is usually referred to as leave-one-out (LOO) cross-validation.
- Carry out the training of the system M times. For each run, use a different partition as a validation set, and use the restating partitions as the training set. Evaluate the performance for different choices of the hyperparameter (i.e., for different values of k for the k-NN method).
- Average the validation error over all partitions, and pick the hyperparameter that provided the minimum validation error.
- Rerun the algorithm using all the training data, keeping the value of the parameter that came out of the cross-validation process.

[ ]:
```
## This fragment of code runs k-nn with M-fold cross validation

#ăObtain the indices for the different folds
n_tr = X_tr.shape[0]
M = n_tr
permutation = np.random.permutation(n_tr)

# Initialize sets of indices
set_indices = {n: [] for n in range(M)}

# Distribute data amont M partitions
n = 0
for pos in range(n_tr):
    set_indices[n].append(permutation[pos])
    n = (n+1) % M

# Now, we run the cross-validation process using the k-nn method
k_max = 30
k_list = [2*j+1 for j in range(int(k_max))]

# Obtain the validation errors
pe_val = 0
for n in range(M):
    i_val = set_indices[n]
    i_tr = []
    for kk in range(M):
        if not n==kk:
            i_tr += set_indices[kk]

    pe_val_iter = []
    for k in k_list:
        y_tr_iter = knn_classifier(X_tr[i_tr], Y_tr[i_tr], X_tr[i_val], k)
        pe_val_iter.append(np.mean(Y_tr[i_val] != y_tr_iter))
    pe_val = pe_val + np.asarray(pe_val_iter).T

pe_val = pe_val / M

# We compute now the train and test errors curves
pe_tr = [np.mean(Y_tr != knn_classifier(X_tr, Y_tr, X_tr, k).T) for k in k_list]

k_opt = k_list[np.argmin(pe_val)]
pe_tst = np.mean(Y_tst != knn_classifier(X_tr, Y_tr, X_tst, k_opt).T)

plt.plot(k_list, pe_tr,'b--o',label='Training error')
plt.plot(k_list, pe_val.T,'g--o',label='Validation error')
plt.stem([k_opt], [pe_tst],'r-o',label='Test error')
plt.legend(loc='best')
plt.title('The optimal value of $k$ is ' + str(k_opt))
plt.xlabel('$k$')
plt.ylabel('Error rate')
plt.show()
```

1.7 6. Scikit-learn implementation

In practice, most well-known machine learning methods are implemented and available for python. Probably, the most complete library for machine learning is Scikit-learn. The following piece of code uses the method

```
KNeighborsClassifier
```

available in Scikit-learn, to compute the *k*-NN classifier using the four components of the observations in the original dataset. This routine allows us to classify a particular point using a weighted average of the targets of the neighbors:

To classify point **x**:

- Find *k* closest points to **x** in the training set
- Average the corresponding targets, weighting each value according to the distance of each point to **x**, so that closer points have a larger influence in the estimation.

[ ]:
```
k = 5

# import some data to play with
iris = datasets.load_iris()

# Take training test
X_tr = np.array([xTrain_all[n] for n in range(nTrain_all)
                 if cTrain_all[n]==c0 or cTrain_all[n]==c1])
C_tr = [cTrain_all[n] for n in range(nTrain_all)
          if cTrain_all[n]==c0 or cTrain_all[n]==c1]
Y_tr = np.array([int(c==c1) for c in C_tr])
n_tr = len(X_tr)
```

```
# Take test set
X_tst = np.array([xTest_all[n] for n in range(nTest_all)
                  if cTest_all[n]==c0 or cTest_all[n]==c1])
C_tst = [cTest_all[n] for n in range(nTest_all)
         if cTest_all[n]==c0 or cTest_all[n]==c1]
Y_tst = np.array([int(c==c1) for c in C_tst])
n_tst = len(X_tst)

for weights in ['uniform', 'distance']:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(k, weights=weights)
    clf.fit(X_tr, Y_tr)
    Z = clf.predict(X_tst)
    pe_tst = np.mean(Y_tst != Z)
    print('Test error rate with ' + weights + ' weights = ' + str(pe_tst))
```

Here you can find an example of the application of `KNeighborsClassifier` to the complete 3-class Iris flower classification problem.

1.8 *k*-NN Classification and Probability Estimation.

If a sample $\mathbf{x}$ has m neighbors from class 1 and $k-m$ neighbors from class 0, we can estimate the posterior probability that an observation $\mathbf{x}$ belongs to class 1 as

$$\hat{P}(\{y=1\}) = \frac{m}{k}$$

Therefore, besides computing a decision about the class of the data, we can modify the k-NN algorithm to obtain posterior probability estimates.

Note that the above equation is equivalent

$$\hat{P}(\{y=1\}) = \frac{\sum_{n\in\mathcal{N}(\mathbf{x})} y^{(n)}}{k}$$

where $\mathcal{N}(\mathbf{x})$ is the set of indices for the samples in the neighborhood of $\mathbf{x}$.

In other words, $\hat{P}(\{y=1\})$ is the *average* of the neighbors labels. Averages can be computed using `KNeighborsRegressor`, which is useful for regression applications.

Exercise 3: Plot a k-NN posterior probability map for the Iris flower data, for $k=15$.

[ ]:
```
# Select two classes
c0 = 'Iris-versicolor'
c1 = 'Iris-virginica'

# Select two coordinates
ind = [0, 1]

# Take training test
X_tr = np.array([[xTrain_all[n][i] for i in ind] for n in range(nTrain_all)
                  if cTrain_all[n]==c0 or cTrain_all[n]==c1])
C_tr = [cTrain_all[n] for n in range(nTrain_all)
```

```
        if cTrain_all[n]==c0 or cTrain_all[n]==c1]
Y_tr = np.array([int(c==c1) for c in C_tr])
n_tr = len(X_tr)

# Take test set
X_tst = np.array([[xTest_all[n][i] for i in ind] for n in range(nTest_all)
                 if cTest_all[n]==c0 or cTest_all[n]==c1])
C_tst = [cTest_all[n] for n in range(nTest_all)
         if cTest_all[n]==c0 or cTest_all[n]==c1]
Y_tst = np.array([int(c==c1) for c in C_tst])
n_tst = len(X_tst)

# Separate components of x into different arrays (just for the plots)
x0c0 = [X_tr[n][0] for n in range(n_tr) if Y_tr[n]==0]
x1c0 = [X_tr[n][1] for n in range(n_tr) if Y_tr[n]==0]
x0c1 = [X_tr[n][0] for n in range(n_tr) if Y_tr[n]==1]
x1c1 = [X_tr[n][1] for n in range(n_tr) if Y_tr[n]==1]
```

[ ]:
```
#<SOL>
k = 15

from sklearn import neighbors

neigh = neighbors.KNeighborsRegressor(n_neighbors=k)
Z = neigh.fit(X_tr,Y_tr).predict(X_grid)

# Put the result into a color plot
plt.plot(x0c0, x1c0,'r.', label=labels[c0])
plt.plot(x0c1, x1c1,'g+', label=labels[c1])
plt.xlabel('$x_' + str(ind[0]) + '$')
plt.ylabel('$x_' + str(ind[1]) + '$')
plt.legend(loc='best')

Z = Z.reshape(xx.shape)
CS = plt.contourf(xx, yy, Z)
CS2 = plt.contour(CS, levels=[0.5],
                  colors='m', linewidths=(3,))
plt.show()
#</SOL>
```