Data Sampling, Visualization, Learning and Classification

Machine Learning – Course Laboratory

Battista Biggio
battista.biggio@diee.unica.it

Luca Didaci
didaci@diee.unica.it

Dept. Of Electrical and Electronic Engineering
University of Cagliari, Italy
Exercise 1

• Create and use a function that returns the average $\ell_2$ norm of the samples of the requested class
  – The Euclidean (or $\ell_2$) norm is given as: $||x||_2 = (|x_1|^2 + |x_2|^2 + \cdots + |x_d|^2)^{1/2}$

HINT: use `np.linalg.norm()` and `np.mean()`

Example:

```
x=
[[ 0.68555608  0.20344581  0.80653113  0.15499494]
 [ 0.0914079  0.29410236  0.31590947  0.18369847]
 [ 0.58229255  0.18869802  0.12665256  0.04362462]
 [ 0.3732148  0.57234487  0.87225276  0.49599479]
 [ 0.67107298  0.14610644  0.93995237  0.10073935]]

y=[1 1 0 1 0]

Patterns belonging to class 0 :
[[ 0.58229255  0.18869802  0.12665256  0.04362462]
 [ 0.67107298  0.14610644  0.93995237  0.10073935]]

Length of patterns belonging to class 0 :
[0.62659041 1.16847974]

Average length of the patterns belonging to class 0 :
0.897535073518
```
import numpy as np

def avg_norm_samples_of_class(x, y, y0):
    """Computes the average norm of samples belonging to y0""
    norm_values = np.linalg.norm(x[y == y0, :], ord=2, axis=1)
    return np.mean(norm_values)

n_patterns = 5
n_features = 4
x = np.random.rand(n_patterns, n_features)
y = np.random.randint(0, 2, n_patterns)
y0 = 0

print 'x=', x
print 'y=', y
avg_norm = avg_norm_samples_of_class(x, y, y0)
print 'Average norm of samples of class ', y0, ': ', avg_norm
Exercise 2

- Consider the function `make_gaussian_dataset` defined in the previous lab session.

- Extend it to handle:
  1. more than two dimensions
  2. more than two classes
  3. non-isotropic Gaussians
     - covariance matrix is a positive-definite matrix, not necessarily proportional to the identity matrix (namely, features are correlated!)
Exercise 2 – This is the starting point...

```python
import numpy as np

def make_gaussian_dataset(n0, n1, mu0, mu1):
    """ Creates a 2-class 2-dimensional Gaussian dataset. """
    d = 2  # hard-coded for convenience, we will improve this later on
    x0 = np.random.randn(n0, d) + mu0  # uses broadcasting...
    x1 = np.random.randn(n1, d) + mu1

    # sample labels
    y0 = np.zeros(n0)
    y1 = np.ones(n1)

    # concatenate data and labels
    x = np.vstack((x0, x1))
    y = np.hstack((y0, y1))

    return x, y

# generate data with 10 samples/class, and means [-1,-1], [1, 1]
xn, yn = make_gaussian_dataset(10, 10, [-1, -1], [+1, +1])
print 'xn: ', xn
print 'yn: ', yn
```

def make_gaussian_dataset(n, mu):
    """
    Creates a k-class d-dimensional Gaussian dataset.
    :param n: vector containing the number of samples for each class
    :param mu: matrix containing the mean vector for each class
    :return: x, y, the gaussian dataset
    """

    n = np.array(n)  # convert to np.array if list is passed as input
    mu = np.array(mu)
    n_classes = mu.shape[0]  # number of classes
    n_features = mu.shape[1]  # number of features
    n_samples = n.sum()  # total number of samples

    x = np.zeros(shape=(n_samples, n_features))
    y = np.zeros(shape=(n_samples,))

    start_index = 0
    for i in xrange(n_classes):
        x_tmp = np.random.randn(n[i], n_features) + mu[i, :]
        # broadcasting...
        x[start_index:start_index + n[i], :] = x_tmp
        y[start_index:start_index + n[i], :] = i
        start_index += n[i]

    return x, y
Exercise 2: Solution

- This is still not considering different covariance matrices per class

```python
import numpy as np

def make_gaussian_dataset(n, mu):
    [...]  

# generate data
xn, yn = make_gaussian_dataset([10, 5, 2], [[-1, -1], [+1, -1], [-1, +1]])

print 'xn: ', xn
print 'yn: ', yn
```

- How to extend it to use a different covariance matrix per class? `make_gaussian_dataset(n, mu, cov)`?
Exercise 3

Define a function that plots a dataset using a different color for each class:

\[ \text{plot
dataset} \ (x, \ y, \ \text{feat}_1=0, \ \text{feat}_2=1) \]

Hints:
import matplotlib.pyplot as plt
plt.scatter(x1, x2, color='r')
plots the point \((x1, x2)\) as a red point
Colors are: ['k', 'b', 'r', 'g', 'c', 'm', 'y']

bool_class0=(y==0)  # select samples belonging to class 0

Other useful functions:
plt.xlabel(), plt.ylabel(), plt.legend(), plt.show()
import matplotlib.pyplot as plt

def plot_dataset(x, y, feat_1=0, feat_2=1):
    n_classes = len(np.unique(y))
    colors = ['r', 'b', 'k', 'g', 'c', 'm', 'y']

    for y0 in xrange(n_classes):
        x0 = x[y == y0, feat_1]
        x1 = x[y == y0, feat_2]
        plt.scatter(x0, x1, c=colors[y0], label='class ' + str(y0))

    plt.legend()
    plt.xlabel('feature x' + str(feat_1))
    plt.ylabel('feature x' + str(feat_2))

    return
Exercise 3: Solution

# generate data
xn, yn = make_gaussian_dataset([50, 50, 50], [[-5, -5], [+5, -5], [-5, +5]])

plot_dataset(xn, yn, 0, 1)
What’s next? Learning and Classification

• Now we can sample data and visualize it in two dimensions

• The goal of the next exercises is to implement a simple classifier
  – The Nearest Mean Classifier (NMC)

• We will implement its learning and classification procedures
Ex. 4: NMC – Learning & Classification

• During the learning phase, NMC is given a training set consisting of pairs \((x, y)\) of samples along with their labels.

• For each class \(y_0\) in \(y\):
  – NMC estimates the mean of the samples in class \(y_0\)
  – Stores the mean vector (centroid)

• During classification, NMC assigns the current test sample \(x\) to the class whose mean vector (centroid) is the closest one to \(x\).

• Implement the functions:
  – \(\text{centroids} = \text{fit}(x, y)\), corresponding to the learning phase, and
  – \(\text{y_pred}, \text{distances} = \text{predict}(x, \text{centroids})\), corresponding to the classification phase, where \(\text{y_pred}\) is the label of the predicted class, and \(\text{distances}\) are the distance values w.r.t. the centroids.
import numpy as np

def fit(x, y):
    n_classes = np.unique(y).size
    n_features = x.shape[1]

    centroids = np.zeros(shape=(n_classes, n_features))
    for k in xrange(n_classes):
        centroids[k] = x[y == k, :].mean(axis=0)
    return centroids

def predict(x, centroids):
    n_samples = x.shape[0]
    n_classes = centroids.shape[0]
    distances = np.zeros(shape=(n_samples, n_classes))

    for k in xrange(n_classes):
        distances[:, k] = np.linalg.norm(x-centroids[k, :], axis=1)
    y_pred = np.argmin(distances, axis=1)

    return y_pred, distances
Let’s create a class

class CNearestMeanClassifier(object):
    """Class implementing a nearest mean classifier"""

    def __init__(self):
        self._centroids = None
        return

    def fit(self, x, y):
        n_classes = np.unique(y).size
        n_features = x.shape[1]
        centroids = np.zeros(shape=(n_classes, n_features))
        for k in xrange(n_classes):
            centroids[k] = x[y == k, :].mean(axis=0)
        self._centroids = centroids
        return

    def predict(self, x):
        n_samples = x.shape[0]
        n_classes = self._centroids.shape[0]
        distances = np.zeros(shape=(n_samples, n_classes))
        for k in xrange(n_classes):
            distances[:, k] = np.linalg.norm(x - self._centroids[k, :], axis=1)
        y_pred = np.argmin(distances, axis=1)
        return y_pred, distances
Class Properties

• Python decorator to access class private members
  – See also 'setters'

```python
class CNearestMeanClassifier(object):
    """Class implementing a nearest mean classifier""
    
    def __init__(self):
        self._centroids = None
        return

    @property
    def centroids(self):
        return self._centroids

    [...]
Scikit-learn Classifiers


• NearestCentroid implements our CNearestMeanClassifier
def plot_decision_regions(x, y, classifier, resolution=0.02):
    # setup marker generator and color map
    colors = ('red', 'blue', 'lightgreen', 'black', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = x[:, 0].min() - 1, x[:, 0].max() + 1
    x2_min, x2_max = x[:, 1].min() - 1, x[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                            np.arange(x2_min, x2_max, resolution))
    Z, score = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    plot_dataset(x, y)
    return
Creating our own function library

- Packages allow importing modules and functions
  - Organized in folders, include the import file `__init__.py`
Exercise 5: Solution

```python
from src.prlib import CNearestMeanClassifier, 
    make_gaussian_dataset, plot_decision_regions
import matplotlib.pyplot as plt

x, y = make_gaussian_dataset([[50, 50, 50], [-5, -5], [+5, -5], [-5, +5]])

classifier = CNearestMeanClassifier()
classifier.fit(x, y)

plot_decision_regions(x, y, classifier)

# plot centroids
plt.scatter(classifier.centroids[:, 0], classifier.centroids[:, 1], marker='x', color='k')
plt.show()
```
Ex. 6: Testing performance on unseen data

• To assess classifier performance, one should estimate the classification error on never-before-seen data
  – The training data should not be used to this end, as it provides an optimistic estimate of the real performance!
• Therefore, the correct procedure amounts to:
  1. Sampling a training and a testing set (from the same underlying distribution), e.g., with `make_gaussian_data(n, mu)`
  2. Fitting the classifier on training data
  3. Predicting the class labels of testing data
  4. Evaluating the fraction of wrong labels

```python
x_tr, y_tr = make_gaussian_dataset(n, mu)
x_ts, y_ts = make_gaussian_dataset(n, mu)
clf = CNeareastMeanClassifier()
clf.fit(x_tr, y_tr)
y_pred, dist = clf.predict(x_ts)
error = (y_pred != y_ts).mean()
```

What happens if one changes means and/or covariances of the Gaussian classes? How does the error vary?
Lessons learned

- Visualize data and decision regions
- Implementation of a simple classifier (using a Python class)
- Create packages and dedicated function libraries
- Basic estimation of classifier performance on unseen data

Student challenges:

1. Extend make_gaussian_dataset to handle covariance matrices
2. Implement a k-Nearest Neighbor (kNN) classifier
3. Visualize decision regions of scikit-learn classifiers using
   - Nearest Centroid, and kNN (you may try other algorithms as well!)
4. Estimate performance of each classifier on unseen data

Please e-mail us if you are able to solve any of them!