Metodi di Apprendimento Automatico per la Fisica

Lesson 3

Dr. Autilia Vitiello

Department of Physics "Ettore Pancini"
University of Naples Federico II

2021/2022

Table of contents

Introduction to Classification in Python

Choose a Classification algorithm

- " No Free Lunch' theorem: no single classifier works best across all possible scenarios.
- It is always recommended that you compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem.
- The performance of a classifier, computational power as well as predictive power, depends heavily on the underlying data that are available for learning.
- The five main steps that are involved in training a machine learning algorithm can be summarized as follows:
  1. Selection of features.
  2. Choosing a performance metric.
  3. Choosing a classifier and optimization algorithm.
  4. Evaluating the performance of the model.
  5. Tuning the algorithm.

- We will take a look at the SciKit-Learn API, which combines a user-friendly interface with a highly optimized implementation of several classification algorithms.
- The SciKit-Learn library offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models.
- To get started with the SciKit-Learn library, we will train a perceptron model.
- We will use the already familiar Iris dataset. The Iris dataset is already available via SciKit-Learn, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms.

- We will assign the petal length and petal width of the 150 flower samples to the feature matrix X and the corresponding class labels of the flower species to the vector y:

```
from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> print('Class labels:', np.unique(y))
Class labels: [0 1 2]
```

- To evaluate how well a trained model performs on unseen data, we will further split the dataset into separate training and test datasets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=1, stratify=y)
```

- Note that the `train_test_split` function shuffles the training sets internally before splitting. Via the `random_state` parameter, we provided a fixed random seed for the internal pseudo-random number generator that is used for shuffling the datasets prior to splitting. Using such a fixed `random_state` ensures that our results are reproducible.

- We took advantage of the built-in support for stratification via `stratify=y`. In this context, stratification means that the `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset. We can use NumPy's `bincount` function, which counts the number of occurrences of each value in an array, to verify that this is indeed the case:

```
>>> print('Labels counts in y:', np.bincount(y))
Labels counts in y: [50 50 50]
>>> print('Labels counts in y_train:', np.bincount(y_train))
Labels counts in y_train: [35 35 35]
>>> print('Labels counts in y_test:', np.bincount(y_test))
Labels counts in y_test: [15 15 15]
```

- The min-max scaling procedure is implemented in scikit-learn and can be used as follows:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

- Similar to the `MinMaxScaler` class, scikit-learn also implements a class for standardization:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

- Many machine learning and optimization algorithms also require feature scaling for optimal performance. Here, we will standardize the features using the `StandardScale` class from scikit-learn's `preprocessing` module:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

- Many machine learning and optimization algorithms also require feature scaling for optimal performance. Here, we will standardize the features using the `StandardScale` class from scikit-learn's `preprocessing` module:

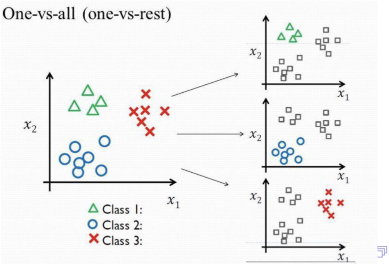
```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

- Using the `fit` method, `StandardScaler` estimated the parameters μ (sample mean) and σ (standard deviation) for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters μ and σ.

- Most algorithms in scikit-learn already support multiclass classification by default via the One-versus-Rest (OvR) method.

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(n_iter=40, eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

- The One-vs-Rest strategy splits a multi-class classification into one binary classification problem per class.



- Having trained a model in scikit-learn, we can make predictions via the `predict` method as follows:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified samples: %d' % (y_test !=
    y_pred).sum())
   Misclassified samples: 3
```

- The scikit-learn library also implements a large variety of different performance metrics that are available via the `metrics` module. For example, we can calculate the classification accuracy of the perceptron on the test set as follows:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
Accuracy: 0.93
```

- Alternatively, each classifier in scikit-learn has a `score` method, which computes a classifier's prediction accuracy by combining the `predict` call with `accuracy_score` as shown here:

```
>>> print('Accuracy: %.2f' % ppn.score(X_test_std, y_test))
Accuracy: 0.93
```

- We can use the following function, `plot_decision_regions`, to plot the decision regions of our newly trained perceptron model and highlight the samples from the test dataset via small circles:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


def plot_decision_regions(X, y, classifier,
    test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
```

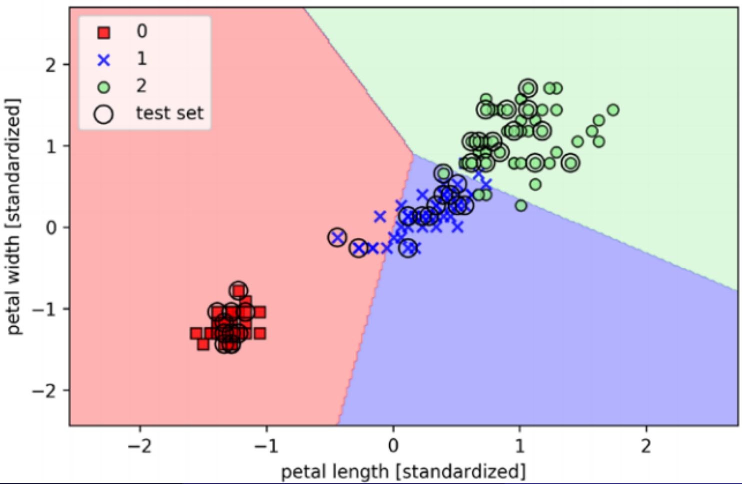
```
 # plot the decision surface
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max,
    resolution), np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(),
    xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
        alpha=0.8, c=colors[idx], marker=markers[idx],
        label=cl, edgecolor='black')
```

```
# highlight test samples
if test_idx:
    # plot all samples
    X_test, y_test = X[test_idx, :], y[test_idx]
    plt.scatter(X_test[:, 0], X_test[:, 1], c='',
        edgecolor='black', alpha=1.0,
    linewidth=1, marker='o', s=100, label='test set')
```

- With the slight modification that we made to the `plot_decision_regions` function, we can now specify the indices of the samples that we want to mark on the resulting plots. The code is as follows:

```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
        y=y_combined, classifier=ppn, test_idx=range(105,
            150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

- As we can see in the resulting plot, the three flower classes cannot be perfectly separated by a linear decision boundary:



First Steps with SciKit-Learn

Training Adaline with mlxtend

- Adaline is not implemented in scikit-learn, but, you can use the module named **mlxtend**, a python module that extends the scikit-learn functionalities:

```
>>> from mlxtend.classifier import Adaline
>>> ada = Adaline(epochs=10, eta=0.01, minibatches=1,
    random_seed=1)
>>> ada.fit(X_std, y)
```

- If the parameter `minibatches` is set to `len(y)`, Adaline is trained using the stochastic gradient descent.