Lecture 7: A Tour of Machine Learning Classifiers Using Scikit-learn

Dr. Huiping Cao

Choosing classification algorithms

- No single classifier works best across all possible scenarios.
- The performance of a classifier (computational performance and predictive power) depends heavily on the underlying data.
- It is always recommended that you **compare the performance of at least several different algorithms.**

Major steps for training classifiers

- Selecting features and training samples
- Choosing a performance metric
- Choosing a classifier and optimization algorithm
- Evaluating the performance of the model
- Tuning the algorithm

The scikit-learn library

- 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.
- **Example**: train a perceptron model similar to the one that we implemented.
  - Dataset: Iris

Step 1: loading data to memory

- We can read the iris dataset using Pandas read_csv function.
- The Iris dataset is also available via scikit-learn.
- The **sklearn.datasets** package embeds some small toy datasets
- Datasets loaded by Scikit-Learn generally have a similar dictionary structure including the following attributes
  - ‘**data**’, the data to learn
  - ‘**target**’, the classification labels
  - ‘**target_names**’, the meaning of the labels
  - ‘**feature_names**’, the meaning of the features
  - ‘**DESCR**’, the full description of the dataset

```
>>> from sklearn import datasets
>>> import numpy as np

>>> iris = datasets.load_iris()
>>> iris.data
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
          …
       [5.9, 3. , 5.1, 1.8]])
>>> type(iris.data)
<class 'numpy.ndarray'>
```

```
>>> iris.target
array([0, 0, 0, 0, 0 …. 1, … 2])
>>> type(iris.target)
<class 'numpy.ndarray’>
>>> iris.target_names
array(['setosa', 'versicolor',
'virginica'], dtype='<U10’)
>>> iris.feature_names
['sepal length (cm)', 'sepal width
(cm)', 'petal length (cm)', 'petal width
(cm)']
```

`keys` function

```
>>> from sklearn import datasets
>>> import numpy as np


>>> iris = datasets.load_iris()
>>> print('Iris keys:', list(iris.keys()))
Iris keys: ['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> print('Iris features:', iris.feature_names)
Iris features: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
>>> iris.data.shape
(150, 4)
```

```
>>> digits = datasets.load_digits()
>>> print('digits keys:', list(digits.keys()))
digits keys: ['data', 'target', 'target_names', 'images', 'DESCR']
>>> print('digits target_names:', digits.target_names)
digits target_names: [0 1 2 3 4 5 6 7 8 9]
>>> digits.data.shape
(1797, 64)
```

Other toy datasets (version 0.22.1)

- datasets.load_boston([return_X_y]) Load and return the boston house-prices dataset (regression).
- datasets.load_breast_cancer([return_X_y]) Load and return the breast cancer wisconsin dataset (classification).
- datasets.load_diabetes([return_X_y]) Load and return the diabetes dataset (regression).
- datasets.load_digits([n_class, return_X_y]) Load and return the digits dataset (classification).
- datasets.load_files(container_path[, …]) Load text files with categories as subfolder names.
- …
- Link: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets

Larger datasets - **fetch_openml**

- Fetch dataset by name or dataset id
- Datasets are uniquely identified by either an integer ID or by a combination of name and version (i.e. there might be multiple versions of the 'iris' dataset). Please give either name or data_id (not both). In case a name is given, a version can also be provided.

```
>>> from sklearn.datasets import fetch_openml
>>> mnist_data = fetch_openml('mnist_784', version=1, cache=True)
>>> print('digits keys:', list(mnist_data.keys()))
digits keys: ['data', 'target', 'feature_names', 'DESCR', 'details', 'categories', 'url']
>>> mnist_data.data.shape
(70000, 784)
```

Step 2: selecting features

- Assume that we only want to use two features (petal length and petal width). In the X matrix, the data for these two features are at columns 2 and 3.
- The target values are already stored as integers (0, 1, 2).
  - Using integer labels is a recommended approach to avoid technical glitches and improve computational performance due to a smaller memory footprint.

```
>>> X = iris.data[:,[2,3]]
>>> y = iris.target
>>> print(np.unique(y))
[0 1 2]
```

Step 3: splitting training and test datasets

- To train a model and conduct prediction, we need to split the dataset to separate training and test datasets. We use the **train_test_split** function from scikit-learn's **model_selection** module.

```
>>> 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)
```

- 30% test data
- random seed
- Stratify splitting

Step 3: splitting training and test datasets

- Verification
  - **np.bincount** function: Count number of occurrences of each value in an array of non-negative integers.

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=1, stratify=y)

>>> 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]
```

Step 4: feature scaling

- Many machine learning algorithms require feature scaling for optimal performance. We implemented such scaling for Adaline algorithm.
- We can standardize the features using **StandardScaler** from scikit-learn's **preprocessing** module.
- **fit** method estimates the parameters μ and σ.
- **transform** method standardize the training data using the estimated parameters.

```
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)
```

Step 4: feature scaling

- Verification: check the mean values and the standard derivations before and after feature scaling.

```
>>> X_train[:,0].mean()
3.7895238095238097
>>> X_train[:,1].mean()
1.197142857142857
>>> X_train[:,0].std()
1.7929982237541362
>>> X_train[:,1].std()
0.7627590448786679
```

```
>>> X_train_std[:,0].mean()
1.1207965772406342e-16
>>> X_train_std[:,1].mean()
6.97854472621527e-17
>>> X_train_std[:,0].std()
1.0
>>> X_train_std[:,1].std()
0.9999999999999997
```

Step 4: feature scaling

- We use the **same scaling parameters** to standardize the test set so that both the values in the training and test dataset are comparable to each other.
- `X_test_std = sc.transform(X_test)`

Step 5: Train classifier

- Use the perceptron class in the module **linear_model**.
- Class parameters:
  - **eta0** is equivalent to the learning rate *eta* in our algorithm.
  - **max_iter** parameter defines the number of epochs.
  - **random_state** parameter to ensure the reproducibility of the initial shuffling of the training dataset after each epoch.
- Most algorithms in scikit-learn already support multi-class classification by default via the **One-versus-Rest (OvR)** method.

```
from sklearn.linear_model import Perceptron

ppn = Perceptron(max_iter=40, eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)
```

Step 6: Make predictions

- Misclassification error

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

- Classification accuracy = 1 - error
- scikit-learn library also implements a large variety of performance metrics (accuracy, f1, precision, recall, AUC, etc.). Those are available via the **metrics** module.

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

Step 6: Make predictions

- Each classifier in scikit-learn has a **score** method, which computes a classifier’s prediction accuracy by combining the **predict** call with **accuracy_score**

```
from sklearn.metrics import accuracy_score
y_pred = ppn.predict(X_test_std)
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

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