SCIKIT-LEARN

- **Scikit-learn** (formerly **scikits.learn**) is a free software machine learning library for the Python programming language.
- It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, *k*-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.
- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license (BSD授權條款：Berkeley Software Distribution license)

SCIKIT-LEARN : Features and Feature Extraction

1. Know how to **extract features** from real-world data in order to perform machine learning tasks.

- Feature extraction involves reducing the amount of resources required to describe a large set of data.

- When performing analysis of complex data one of the major problems stems from the number of variables involved.

- Analysis with a large number of variables generally requires a large amount of memory and computation power, also it may cause a classification algorithm to overfit to training samples and generalize poorly to new samples.

- Feature extraction is a general term for methods of constructing combinations of the variables to get around these problems while still describing the data with sufficient accuracy.
  - Feature extraction - wiki

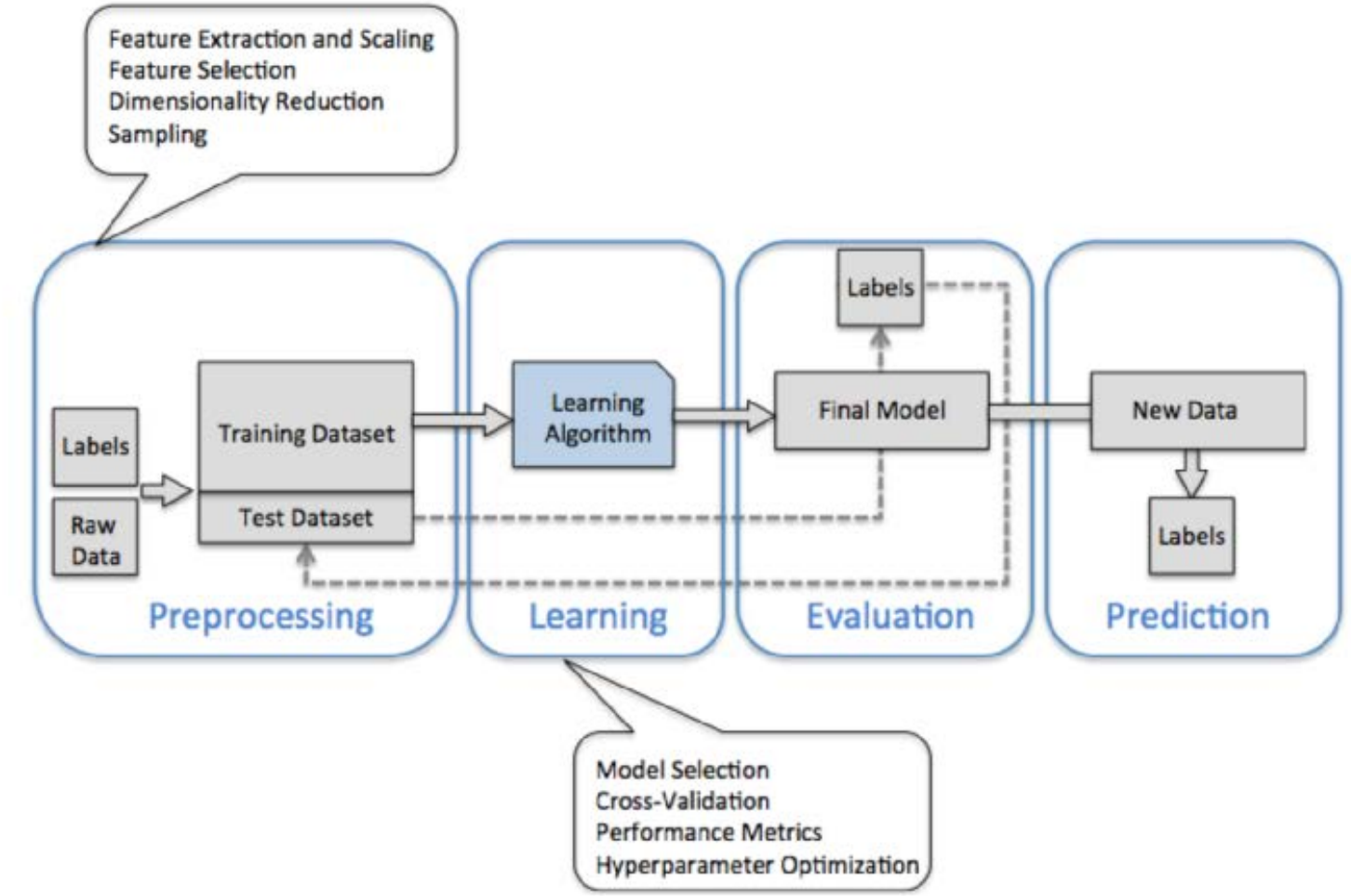
Features and Feature Extraction

2. Know the basic categories of **supervised learning**, including **classification** and **regression** problems.
3. Know the basic categories of **unsupervised learning**, including **dimensionality reduction** and **clustering**.
4. Understand the distinction between **linearly separable** and **non-linearly separable** data.

"Machine Learning is about building programs with tunable parameters (typically an array of floating point values) that are adjusted automatically so as to improve their behavior by adapting to previously seen data."

Features and Feature Extraction

- The diagram shown below is a typical workflow diagram for using machine learning.



Features and feature extraction

1. **Preprocessing** - getting data into shape
   - Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm.
- The preprocessing of the data is one of the most crucial steps in any machine learning application.
  - If we take the Iris flower data set in the next section, we could think of the raw data as a series of flower images from which we want to extract meaningful features.
- Useful features could be the color, the hue, the intensity of the flowers, the height, and the flower lengths and widths.
- Some of the selected features may be highly correlated and therefore redundant to a certain degree.
- In those cases, dimensionality reduction techniques are useful for compressing the features onto a lower dimensional subspace.
  - Reducing the dimensionality of our feature space has the advantage that less storage space is required, and the learning algorithm can run much faster.

Features and feature extraction

2. **Training and selecting a predictive model**
3. **Evaluating models and predicting unseen data instances**
   - After we have selected a model that has been fitted on the training data set, we can use the test data set to estimate how well it performs on this unseen data to estimate the generalization error.
   - If we are satisfied with its performance, we can now use this model to predict new, future data.
   - The parameters for the previously mentioned procedures such as feature scaling and dimensionality reduction are solely obtained from the training data set, and the same parameters are later reapplied to transform the test data set, as well as any new data samples.

Python Machine Learning

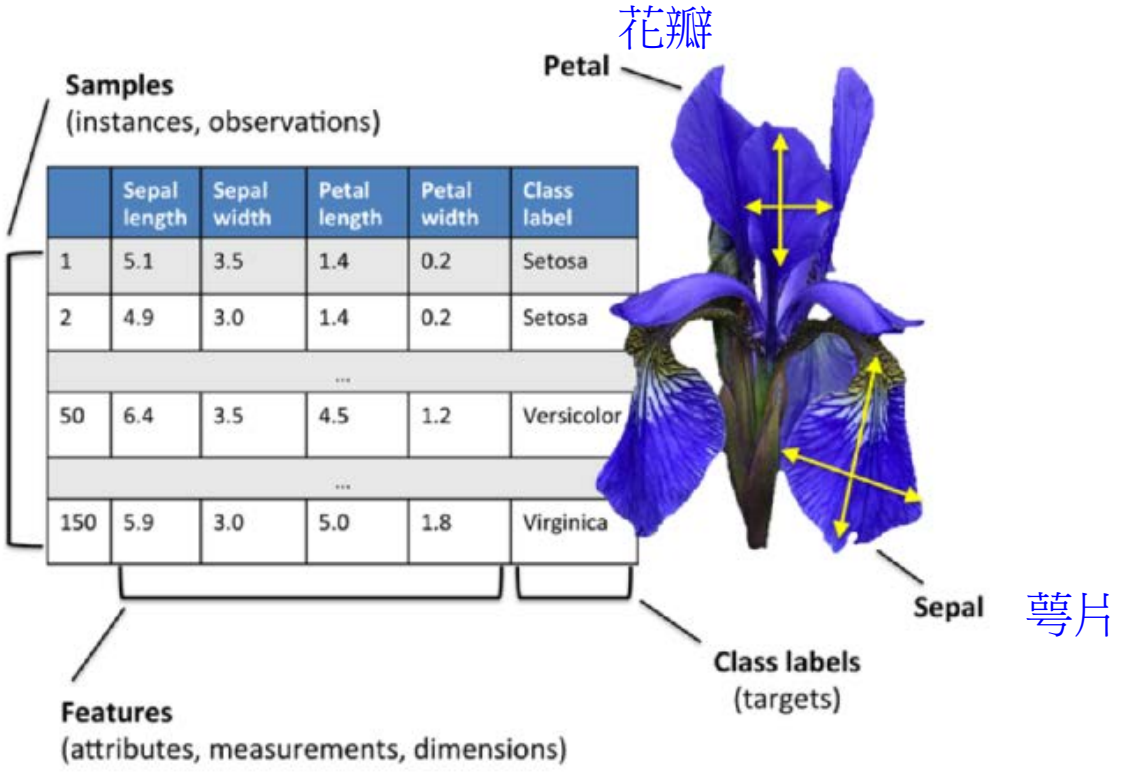
- The version numbers of the major Python packages that were used throughout this tutorial are listed below:
  - NumPy 1.9.1
  - SciPy 0.14.0
  - scikit-learn 0.15.2
  - matplotlib 1.4.0
  - pandas 0.15.2

scikit-learn & numpy

- The scikit-learn API 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 such as preprocessing data, fine-tuning, and evaluating our models.
- Most machine learning algorithms implemented in scikit-learn expect a numpy array as input X that has **(n_samples, n_features)** shape.
  - n_samples: the number of samples.
  - n_features: the number of features or distinct traits that can be used to describe each item in a quantitative manner.

iris flower data set

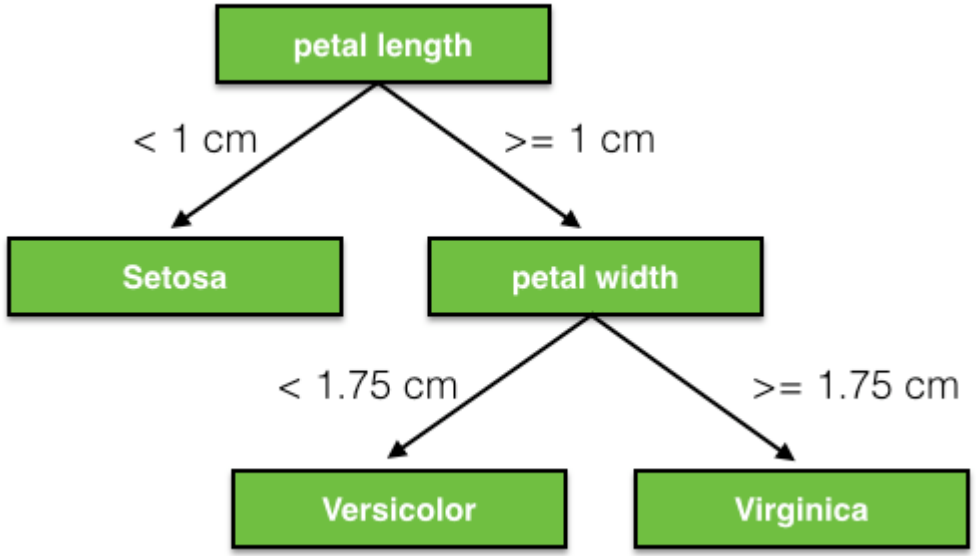
- The data set of this tutorial consists of **50 samples** from each of **three species** of Iris (*setosa*, *virginica* and *versicolor*).

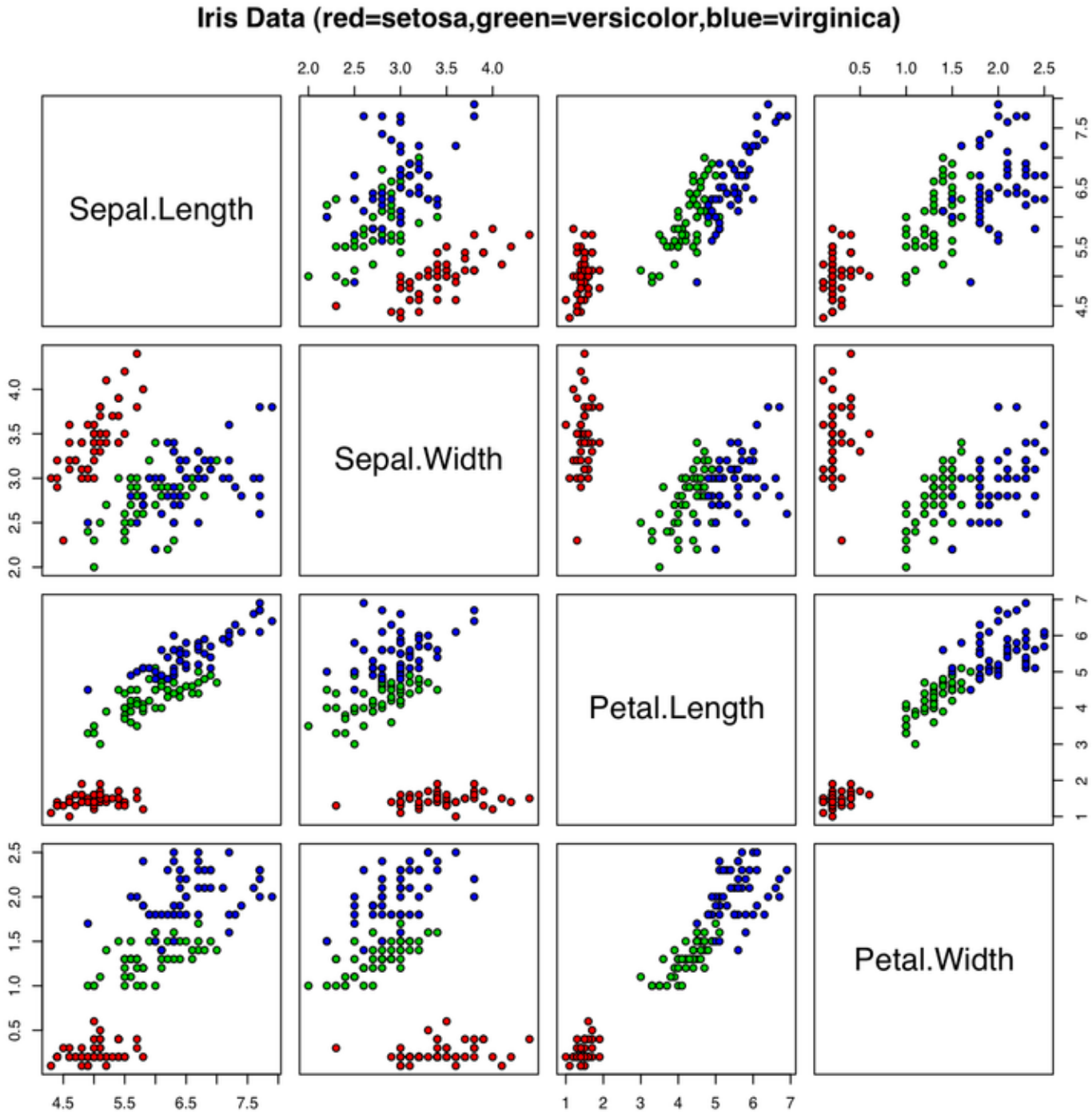


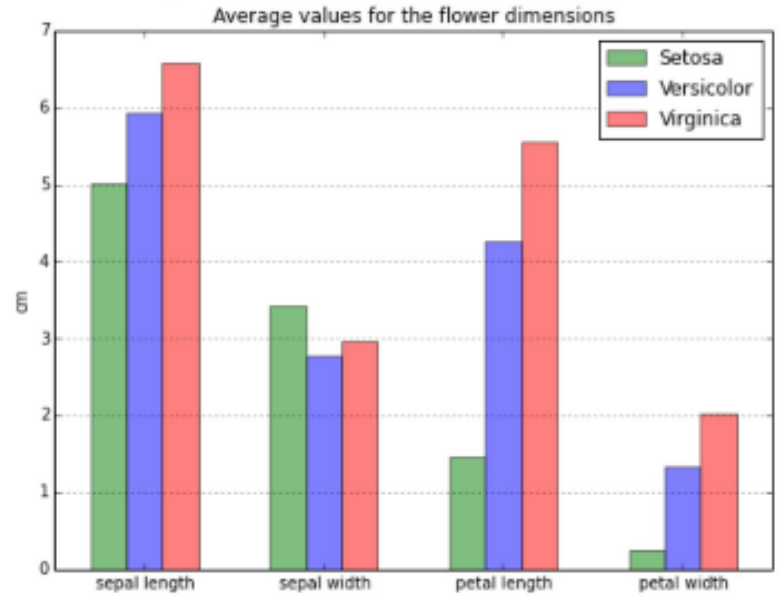
Picture credit: Python Machine Learning by Sebastian Raschka, 2015

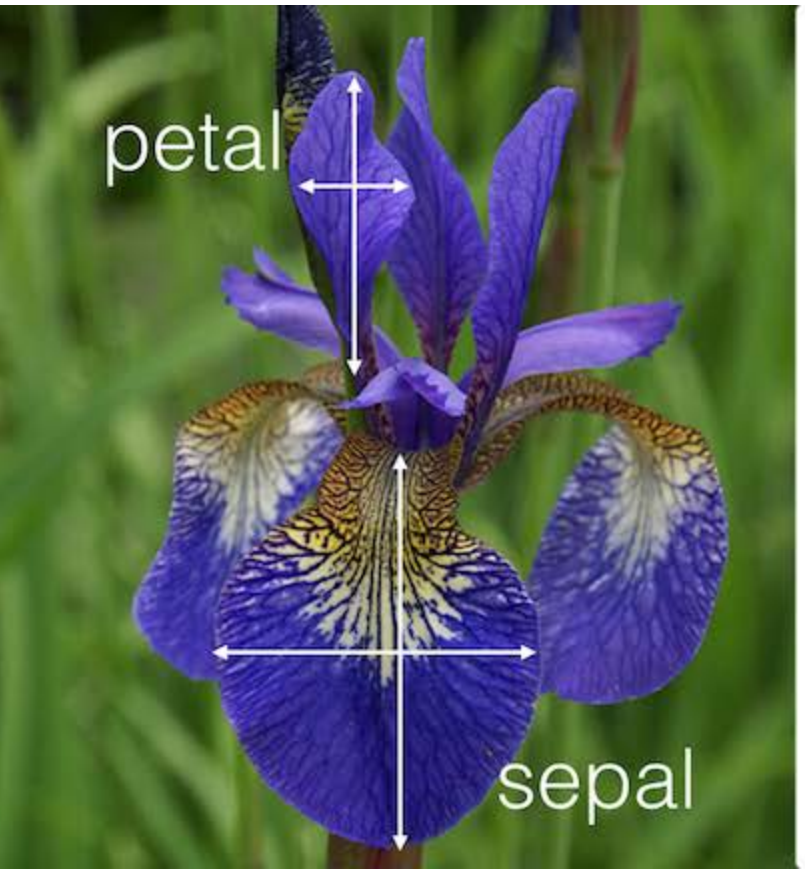
iris flower data set

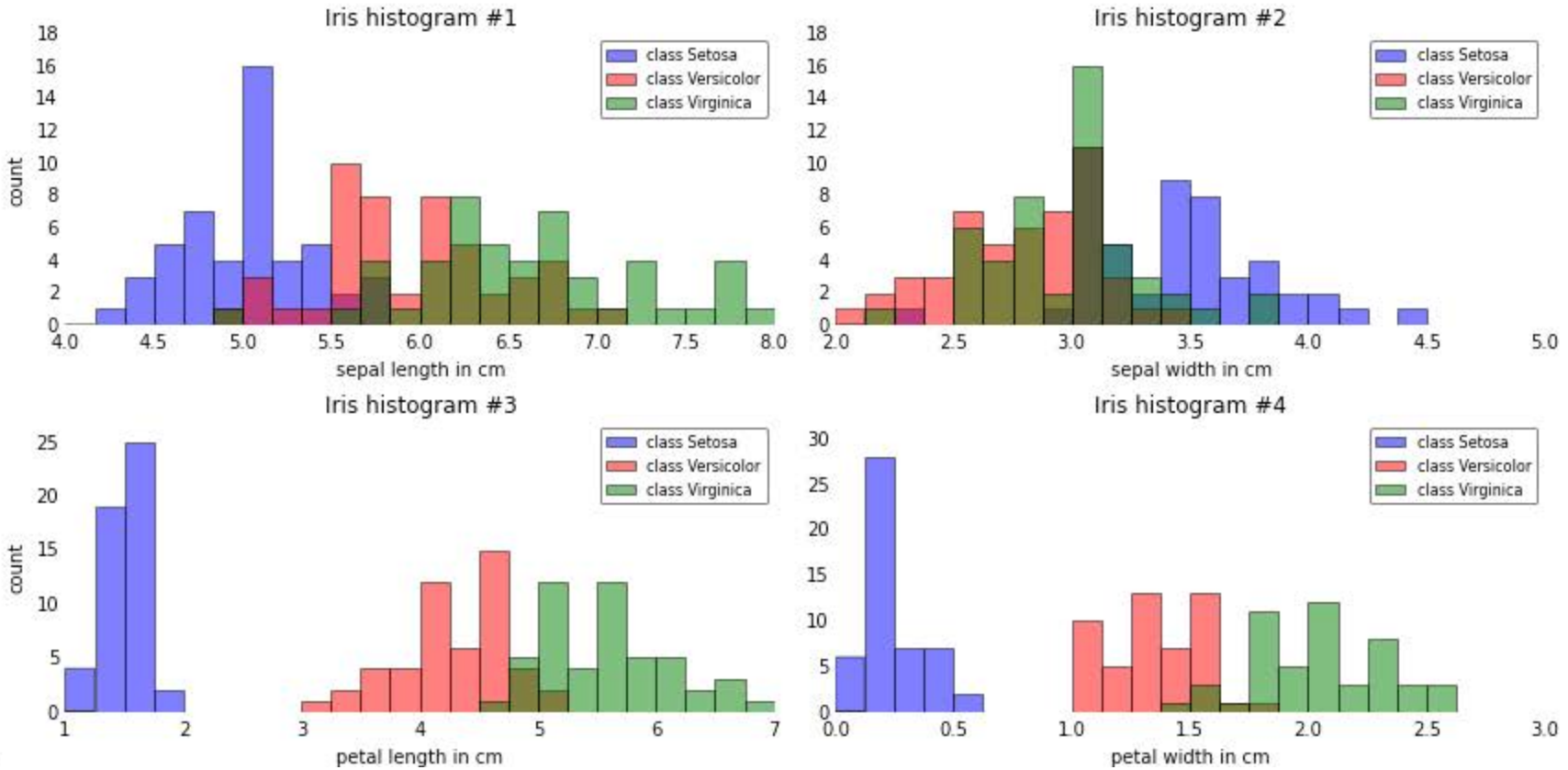
- **Four features** were measured from each sample: the length and the width of the sepals and petals, in centimeters.
- Based on the combination of these four features, Fisher developed a linear discriminant model (線性判別分析) to distinguish the species from each other." - Iris flower data set




Iris Data (red=setosa,green=versicolor,blue=virginica)
Sepal.Length
Sepal.Width
Petal.Length
Petal.Width


Average values for the flower dimensions
Setosa
Versicolor
Virginica
cm
sepal length
sepal width
petal length
petal width


petal
sepal


Iris histogram #1
count
sepal length in cm
class Setosa
class Versicolor
class Virginica
Iris histogram #2
sepal width in cm
Iris histogram #3
petal length in cm
Iris histogram #4
petal width in cm

iris data set & scikit-learn

- **scikit-learn** loads data from CSV file into numpy arrays:

```
In [74]: from sklearn.datasets import load_iris

In [75]: iris = load_iris()
```

- The **data attribute** of the dataset stores the features of each sample flower:

```
In [76]: iris.data
Out[76]:
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       ...,
       [ 6.5,  3. ,  5.2,  2. ],
       [ 6.2,  3.4,  5.4,  2.3],
       [ 5.9,  3. ,  5.1,  1.8]])

In [77]: n_samples, n_features = iris.data.shape

In [78]: n_samples
Out[78]: 150L

In [79]: n_features
Out[79]: 4L
In [80]: iris.data.shape
Out[80]: (150L, 4L)
```

iris data set & scikit-learn

- The **target attribute** of the dataset stores the information about the class of each sample:

```
In [81]: len(iris.target)
Out[81]: 150

In [82]: iris.target
Out[82]: array([0, 0, 0, ..., 2, 2, 2])
```

```
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, 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])
```

- The **target_names** attribute stores the names of the classes:

```
In [83]: iris.target_names
Out[83]:
array(['setosa', 'versicolor', 'virginica'],
      dtype='|S10')
```

string

iris data set & scikit-learn

- If we just want a portion of dataset, for example, "Petal length" and "Petal width", we can extract like this:

```
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       ...,
       [ 6.5,  3. ,  5.2,  2. ],
       [ 6.2,  3.4,  5.4,  2.3],
       [ 5.9,  3. ,  5.1,  1.8]])
```

```
In [90]: from sklearn import datasets

In [91]: import numpy as np

In [92]: iris = datasets.load_iris()

In [93]: X = iris.data[:, [2, 3]]

In [94]: X
Out[94]:
array([[ 1.4,  0.2],
       [ 1.4,  0.2],
       [ 1.3,  0.2],
       ...,
       [ 5.2,  2. ],
       [ 5.4,  2.3],
       [ 5.1,  1.8]])
```

- If we do **np.unique(y)** to return the different class labels stored in **iris.target**, we can see Iris flower class names, Iris-Setosa, Iris-Versicolor, and Iris-Virginica, which are stored as integers (0 , 1 , 2):

```
In [95]: y = iris.target

In [96]: np.unique(y)
Out[96]: array([0, 1, 2])
```

Dataset split and scaling

- The data set consists of **50 samples** from each of **three species** of Iris (setosa, virginica and versicolor).
- **Four features** were measured from each sample: the length and the width of the sepals and petals, in centimeters.
  - We only use two features from the Iris flower dataset in this section.

```
In [1]: from sklearn import datasets

In [2]: import numpy as np

In [3]: iris = datasets.load_iris()

In [4]: X = iris.data[:,[2,3]]

In [5]: y = iris.target
```

- To evaluate how well a trained model is performing on unseen data, we further split the dataset into separate **training** and **test** datasets:

```
from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

Dataset split and scaling

- We randomly split the X and y arrays into 30 percent test data (45 samples) and 70 percent training data (105 samples).
- We also want to do feature scaling for optimal performance of our algorithm using the StandardScaler class from scikit-learn's **preprocessing module**:

```
In [8]: from sklearn.preprocessing import StandardScaler

In [9]: sc = StandardScaler()

In [10]: sc.fit(X_train)
Out[10]: StandardScaler(copy=True, with_mean=True, with_std=True)

In [11]: X_train_std = sc.transform(X_train)

In [12]: X_test_std = sc.transform(X_test)
```

- As we can see from the code, we initialized a new **StandardScaler** object.
- 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.
- Here we used 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.

Scikit Perceptron model

- Now that we have standardized the training data, we can train a **perceptron** model.
- Most of the algorithms in scikit-learn support **multiclass** classification by default via the **One-vs.-Rest (OvR)** method. It allows us to feed the three flower classes to the perceptron all at once.
- The code looks like the following:

```
In [45]: from sklearn.linear_model import Perceptron

In [46]: ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)

In [47]: ppn.fit(X_train_std, y_train)
Out[47]:
Perceptron(alpha=0.0001, class_weight=None, eta0=0.1, fit_intercept=True,
      n_iter=40, n_jobs=1, penalty=None, random_state=0, shuffle=True,
      verbose=0, warm_start=False)
```

sklearn.multiclass.OneVsRestClassifier

class sklearn.multiclass. **OneVsRestClassifier** (*estimator*, *n_jobs=1*) [source]

One-vs-the-rest (OvR) multiclass/multilabel strategy

Also known as one-vs-all, this strategy consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes. In addition to its computational efficiency (only n_classes classifiers are needed), one advantage of this approach is its interpretability. Since each class is represented by one and one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most commonly used strategy for multiclass classification and is a fair default choice.

This strategy can also be used for multilabel learning, where a classifier is used to predict multiple labels for instance, by fitting on a 2-d matrix in which cell [i, j] is 1 if sample i has label j and 0 otherwise.

In the multilabel learning literature, OvR is also known as the binary relevance method.

Read more in the User Guide.

Parameters:

estimator : estimator object

An estimator object implementing fit and one of decision_function or predict_proba.

n_jobs : int, optional, default: 1

The number of jobs to use for the computation. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

Attributes:

estimators_ : list of n_classes estimators

Estimators used for predictions.

classes_ : array, shape = [n_classes]

Scikit Perceptron model

- After loading the **Perceptron** class from the **linear_model** module, we initialized a new Perceptron object and trained the model via the **fit** method.
- Here, the model parameter eta0 is the learning rate η.
- To find an proper learning rate requires some experimentation.
  - If the learning rate is too large, the algorithm will overshoot the global cost minimum.
  - if the learning rate is too small, the algorithm requires more epochs until convergence, which can make the learning slow, especially for large datasets.
- Also, we used the **random_state** parameter for reproducibility of the initial shuffling of the training dataset after each epoch.

Making predictions

- Now that we've trained a model in scikit-learn, we can make predictions via the **predict** method.

```
In [25]: y_pred = ppn.predict(X_test_std)

In [26]: print('Misclassified samples: %d' % (y_test != y_pred).sum())
Misclassified samples: 4
```

- Here, **y_test** are the true class labels and **y_pred** are the class labels that we predicted.
- We see that the perceptron misclassifies 4 out of the 45 flower samples. Thus, the **misclassification** error on the test dataset is 0.08889 or 9% (4 / 45 ~ 0.088889).
- Scikit-learn 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:

```
In [27]: from sklearn.metrics import accuracy_score

In [28]: print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
Accuracy: 0.91
```

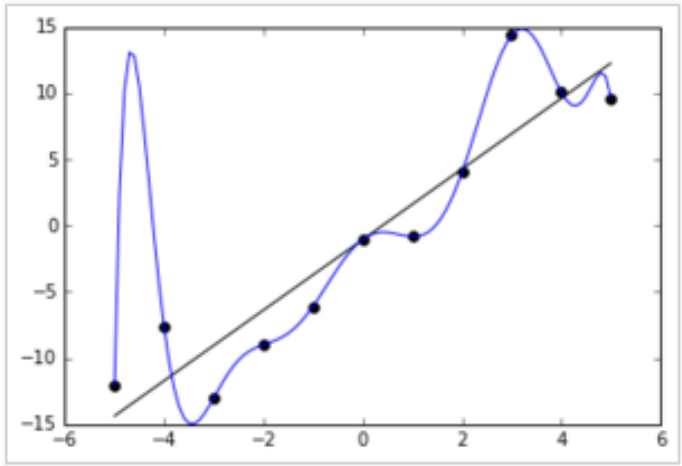
Overfitting

- Let's learn a terminology of machine learning: **overfitting**.
- In statistics and machine learning, one of the most common tasks is to fit a "model" to a set of training data, so as to be able to make reliable predictions on general untrained data.

- In **overfitting**, a statistical model describes random error or noise instead of the underlying relationship.

Overfitting

- Overfitting occurs when a model is excessively complex, such as having too many parameters relative to the number of observations.
- A model that has been overfit has **poor predictive performance**, as it overreacts to minor fluctuations in the training data.



Noisy (roughly linear) data is fitted to both linear and polynomial functions. Although the polynomial function is a perfect fit, the linear version can be expected to generalize better. In other words, if the two functions were used to extrapolate the data beyond the fit data, the linear function would make better predictions.

From Overfitting - wiki

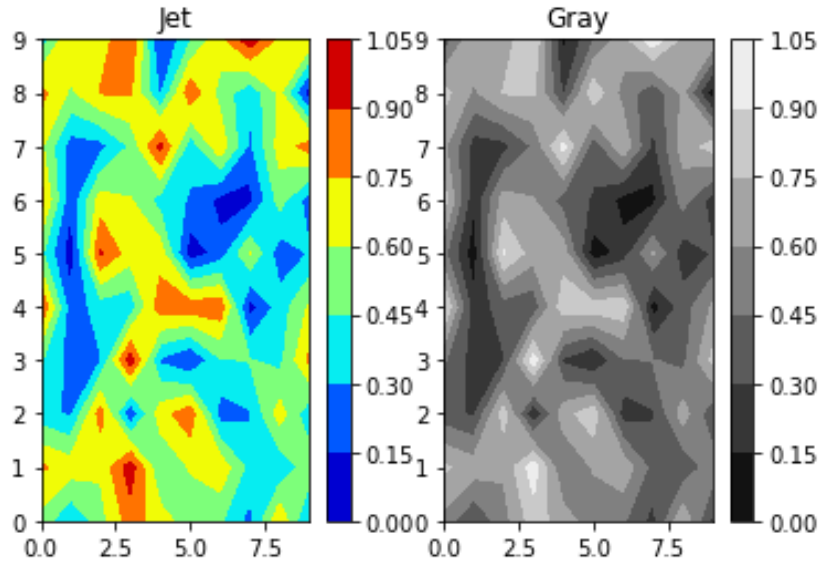
Decision region

- Now, we want to plot decision regions of our trained perceptron model and visualize how well it separates the different flower samples.

contour()

- Axes.contourf(**args*, ***kwargs*)
  - Plot contours.
  - contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

```
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

data = np.random.rand(10,10)

plt.subplot(1,2,1)
con = plt.contourf(data, cmap=cm.jet)
plt.title('Jet')
plt.colorbar()

hax = plt.subplot(1,2,2)
con = plt.contourf(data, cmap=cm.gray)
plt.title('Gray')
plt.colorbar()
```



Find the unique elements of an array.

```python
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
import numpy as np


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=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot all samples
    X_test, y_test = X[test_idx, :], y[test_idx]
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
    # highlight test samples
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                    alpha=1.0, linewidth=1, marker='o',
                    s=55, label='test set')

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
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()
```

meshgrid
按照给定的坐標向量建立坐標矩陣。

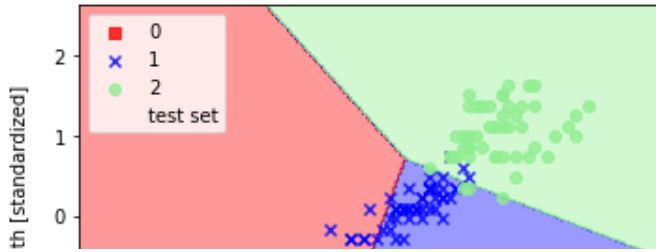
```python
import numpy as np


print np.meshgrid(np.arange(0, 6))
print

x, y = np.meshgrid(np.arange(-1, 2), np.arange(0, 2))
print 'x is:', x
print 'y is:', y
print

print 'points built by (x, y):'
print np.rec.fromarrays([x, y])
```

```
[array([0, 1, 2, 3, 4, 5])]

x is: [[-1  0  1]
 [-1  0  1]]
y is: [[0 0 0]
 [1 1 1]]

points built by (x, y):
[[(-1, 0) (0, 0) (1, 0)]
 [(-1, 1) (0, 1) (1, 1)]]
```

`predict(X)` Perform classification on samples in X.

numpy.reshape(*a, newshape, order='C'*)

Gives a new shape to an array without changing its data.



numpy.unique(*ar, return_index=False, return_inverse=False, return_counts=False*)

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements: th
of the unique array that reconstruct the input array, and the number of times each unique value comes up in the input

petal length [standardized]

numpy.vstack(*tup*)

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by vsplit.

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

numpy.hstack(*tup*)

Stack arrays in sequence horizontally (column wise).

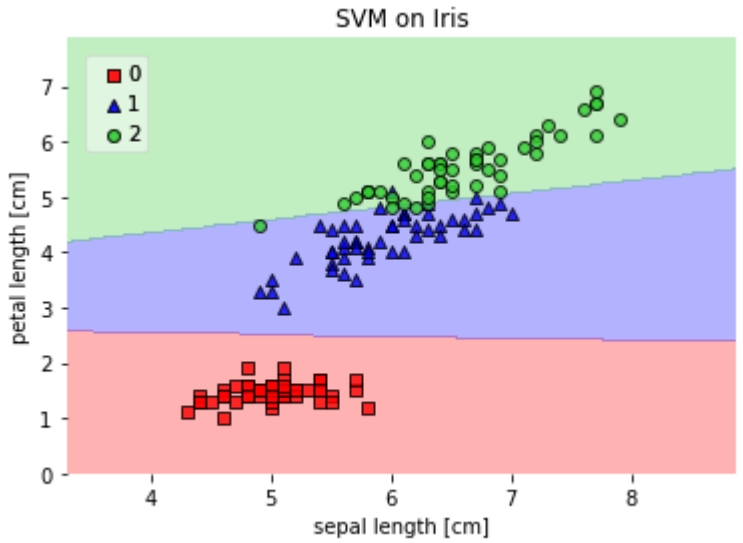
Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by hsplit.

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.hstack((a,b))
array([1, 2, 3, 2, 3, 4])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.hstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

Plotting Decision Regions

A function for plotting decision regions of classifiers in 1 or 2 dimensions.

```
from mlxtend.plotting import plot_decision_regions
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC

# Loading some example data
iris = datasets.load_iris()
X = iris.data[:, [0, 2]]
y = iris.target

# Training a classifier
svm = SVC(C=0.5, kernel='linear')
svm.fit(X, y)


# Plotting decision regions
plot_decision_regions(X, y, clf=svm,
                      res=0.02, legend=2)

# Adding axes annotations
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.title('SVM on Iris')
plt.show()
```



Decision region

- The perceptron algorithm never converges on datasets that aren't perfectly linearly separable, which is why the use of the perceptron algorithm is typically not recommended in practice.
- Later, we will look at more powerful linear classifiers that converge to a cost minimum even if the classes are not perfectly linearly separable.

Handling missing data

- In real-world samples, there are missing one or more values such as the blank spaces in our data table.

- Quite a few computational tools are unable to handle such missing values and might produce unpredictable results.

- So, before we proceed with further analyses, it is critical that we take care of those missing values.

Pandas DataFrame

- To get a better feel for the problem, let's create a simple example using CSV file:
- To easily understand the problem:

```
In [50]: import pandas as pd

In [51]: from io import StringIO

In [52]: csv_data = '''A,B,C,D
    ...: 1.0, 2.0, 3.0, 4.0
    ...: 5.0, 6.0,,8.0
    ...: 0.0,11.0,12.0,'''

In [53]: csv_data = unicode(csv_data)

In [54]: df = pd.read_csv(StringIO(csv_data))

In [55]: df
Out[55]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
1  5.0   6.0   NaN  8.0
2  0.0  11.0  12.0  NaN
```

- The **StringIO()** function allows us to read the string assigned to csv_data into a pandas **DataFrame** via the **read_csv()** function as if it was a regular CSV file on our hard drive.
- Note that the two missing cells were replaced by **NaN**.

Pandas DataFrame

- We can use **isnull()** method to check whether a cell contains a numeric value (False) or if data is missing (True):

```
In [56]: df.isnull()
Out[56]:
       A      B      C      D
0  False  False  False  False
1  False  False   True  False
2  False  False  False   True
```

- For a larger DataFrame, we may want to use the **sum()** method which returns the number of missing values per column:

```
In [57]: df.isnull().sum()
Out[57]:
A    0
B    0
C    1
D    1
dtype: int64
```

- Note that we can always access the underlying **NumPy** array of the DataFrame via the values attribute before we feed it into a scikit-learn estimator:

```
In [58]: df.values
Out[58]:
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [  0.,  11.,  12.,  nan]])
```

Pandas DataFrame

- scikit-learn was developed for working with NumPy arrays, it can sometimes be more convenient to preprocess data using **pandas' DataFrame**.

Eliminating samples/features with missing cells via pandas.DataFrame.dropna()

- We can remove the corresponding features (columns) or samples (rows) from the dataset.
- The rows with missing values can be dropped via the **pandas.DataFrame.dropna()** method:

```
In [59]: df
Out[59]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
1  5.0   6.0   NaN  8.0
2  0.0  11.0  12.0  NaN
```

```
In [62]: df.dropna()
Out[62]:
     A    B    C    D
0  1.0  2.0  3.0  4.0
```

- We can drop columns that have at least one NaN in any row by setting the axis argument to 1:

```
In [63]: df.dropna(axis=1)
Out[63]:
     A     B
0  1.0   2.0
1  5.0   6.0
2  0.0  11.0
```

where axis : {0 or 'index', 1 or 'columns'}.

pandas.DataFrame.dropna()

- The dropna() method has several additional parameters:
- The removal of missing data appears to be a convenient approach, however, it also comes with certain disadvantages:
  - There are chances that we may remove too many, which will make our analysis not reliable.
  - By eliminating too many feature columns, we may run the risk of losing valuable information for our classifier.

```
In [64]: df.dropna(how='all')
Out[64]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
1  5.0   6.0   NaN  8.0
2  0.0  11.0  12.0  NaN
```

```
# only drop rows where all columns are NaN
```

```
In [65]: df.dropna(thresh=4)
Out[65]:
     A    B    C    D
0  1.0  2.0  3.0  4.0
```

```
# drop rows that do not have at least 4 non-NaN values
```

```
In [67]: df.dropna(subset=['C'])
Out[67]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
2  0.0  11.0  12.0  NaN
```

```
# only drop rows where NaN appear in specific columns (here: 'C')
```

- We will look into **interpolation techniques** which one of the most commonly used alternatives for dealing with missing data.

Estimating -missing values via interpolation

- Mean imputation is a method replacing the missing values with the mean value of the entire feature column.
- While this method maintains the sample size and is easy to use, the variability in the data is reduced, so the standard deviations and the variance estimates tend to be underestimated.
- We use the **sklearn.preprocessing.Imputer** class:

```
In [59]: df
Out[59]:
     A     B     C    D
0  1.0   2.0   3.0  4.0
1  5.0   6.0   NaN  8.0
2  0.0  11.0  12.0  NaN
```

```
In [71]: from sklearn.preprocessing import Imputer

In [72]: imputer = Imputer(missing_values='NaN',strategy='mean',axis=0)

In [73]: imputer = imputer.fit(df)

In [74]: imputer_data = imputer.transform(df.values)

In [75]: imputer_data
Out[75]:
array([[  1. ,   2. ,   3. ,   4. ],
       [  5. ,   6. ,   7.5,   8. ],
       [  0. ,  11. ,  12. ,   6. ]])
```

Estimating -missing values via interpolation

- Here, we replaced each NaN value by the corresponding mean from each feature column.
- We can use the row means if we change the setting axis=0 to axis=1:

```
In [76]: from sklearn.preprocessing import Imputer

In [77]: imputer = Imputer(missing_values='NaN',strategy='mean',axis=1)

In [78]: imputer = imputer.fit(df)

In [79]: imputer_data = imputer.transform(df.values)

In [80]: imputer_data
Out[80]:
array([[  1.        ,   2.        ,   3.        ,   4.        ],
       [  5.        ,   6.        ,   6.33333333,   8.        ],
       [  0.        ,  11.        ,  12.        ,   7.66666667]])
```

- As for the strategy parameter, there are other options such as **median** or **most_frequent**.

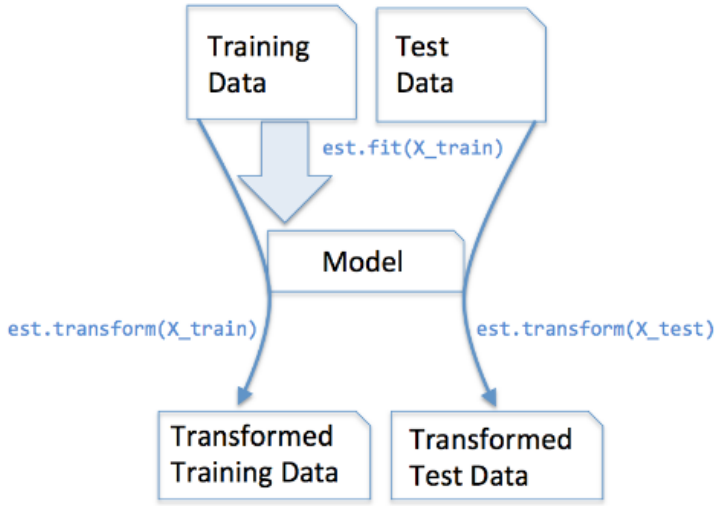
strategy : string, optional (default="mean")

The imputation strategy.

- If "mean", then replace missing values using the mean along the axis.
- If "median", then replace missing values using the median along the axis.
- If "most_frequent", then replace missing using the most frequent value along the axis.

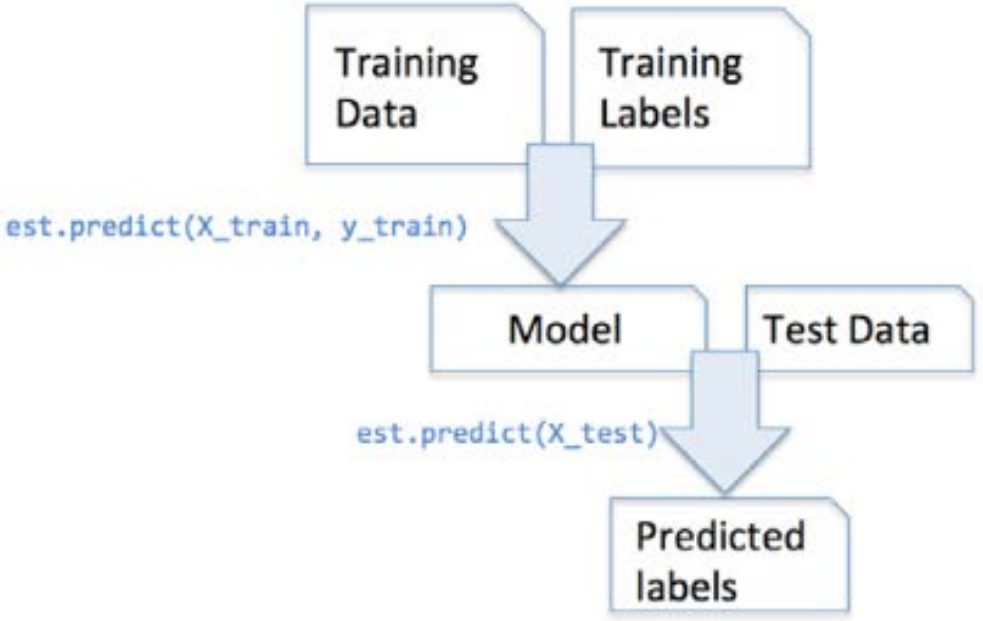
Scikit-learn estimator API

- For **Imputer** class, we used in the previous section belongs to the so-called **transformer** classes in scikit-learn that are used for data transformation.
- There are two methods of estimators:
  1. **fit**: this method is used to learn the parameters from the training data.
  2. **transform**: uses those parameters to transform the data.
- The diagram below shows how a transformer fitted on the training data is used to transform a training dataset as well as a new test dataset:



Scikit-learn estimator API

- In supervised learning tasks, we additionally provide the class labels for fitting the model, which can then be used to make predictions about new data samples via the **predict()** method:



Dealing with categorical data

- Not all data has numerical values. Here are examples of **categorical** data:
  - The blood type of a person: A, B, AB or O.
  - The state that a resident of the United States lives in.
  - T-shirt size. XL > L > M
  - T-shirt color.
- Even among categorical data, we may want to distinguish further between **nominal** and **ordinal** which can be sorted or ordered features.
- Thus, T-shirt size can be an ordinal feature, because we can define an order XL > L > M.

Dealing with categorical data

- Let's create a new categorical data frame:

```
In [83]: import pandas as pd

In [84]: df = pd.DataFrame([
    ...: ['green','M',10.1,'class1'],
    ...: ['red','L',13.5,'class2'],
    ...: ['blue','XL',15.3,'class1']])

In [85]: df
Out[85]:
       0   1     2       3
0  green   M  10.1  class1
1    red   L  13.5  class2
2   blue  XL  15.3  class1
```

```
In [89]: df.columns
Out[89]: RangeIndex(start=0, stop=4, step=1)

In [90]: df.columns=['color','size','price','classlabel']

In [91]: df
Out[91]:
   color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```

- As we can see from the output, the DataFrame contains a nominal feature (color), an ordinal feature (size) as well as a numerical feature (price) column.
- In the last column, the class labels are created.

Ordinal feature mapping

- In order to interpret the ordinal features correctly, we should convert the categorical string values into integers.
- However, since there is no convenient function that can automatically derive the correct order of the labels of our size feature, we have to define the mapping manually.
- Let's assume that we know the difference between features such as XL = L + 1 = M + 2.

	color	size	price	classlabel
0	green	M	10.1	class1
1	red	L	13.5	class2
2	blue	XL	15.3	class1

```
In [92]: size_mapping = {'XL':3, 'L':2, 'M':1}

In [93]: df['size']=df['size'].map(size_mapping)

In [94]: df
Out[94]:
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

```
In [95]: size_mapping.items()
Out[95]: [('M', 1), ('L', 2), ('XL', 3)]
```

Ordinal feature mapping

- If we transform the integer values back to the original string representation, we simply define a reverse-mapping dictionary "inv_size_mapping" that can then be used via the pandas' map method on the transformed feature column similar to the "size_mapping" dictionary that we used previously:

```
In [96]: inv_size_mapping = {v:k for k, v in size_mapping.items()}

In [97]: inv_size_mapping
Out[97]: {1: 'M', 2: 'L', 3: 'XL'}

In [98]: df['size']=df['size'].map(inv_size_mapping)

In [99]: df
Out[99]:
   color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```

Class labels encoding

- Usually, class labels are required to be encoded as integer values.
- While most estimators for classification in scikit-learn convert class labels to integers internally, we may want to provide class labels as integer arrays to avoid any issues.
- To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed in the previous section.
- Since class labels are not ordinal, it doesn't matter which integer number we assign to a particular string-label.

Class labels encoding

- So, we can simply enumerate the class labels starting at 0:

```
np.unique(df['classlabel'])
```

```
array(['class1', 'class2'], dtype=object)
```

```
class_mapping = {label:idx for idx,label in
                 enumerate(np.unique(df['classlabel']))}
class_mapping
```

```
{'class1': 0, 'class2': 1}
```

- Now we want mapping dictionary to transform the class labels into integers:

```
df['classlabel'] = df['classlabel'].map(class_mapping)
df
```

	color	size	price	classlabel
0	green	M	10.1	0
1	red	L	13.5	1
2	blue	XL	15.3	0

Class labels encoding

- As we did for "size" in the previous section, we can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

	color	size	price	classlabel
0	green	M	10.1	0
1	red	L	13.5	1
2	blue	XL	15.3	0

```
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
df
```

	color	size	price	classlabel
0	green	M	10.1	class1
1	red	L	13.5	class2
2	blue	XL	15.3	class1

scikit-learn's LabelEncoder class

- Though we encoded the class labels manually, luckily, there is a convenient **LabelEncoder** class directly implemented in scikit-learn to achieve the same:
- Note that the **fit_transform()** method is just a shortcut for calling fit and transform separately, and we can use the **inverse_transform()** method to transform the integer class labels back into their original string representation:

```
class_label_encoder.inverse_transform(y)
```

```
array(['class1', 'class2', 'class1'], dtype=object)
```

Nominal feature encoding

- So far, we used a simple dictionary-mapping approach to convert the ordinal size feature into integers.
- Because scikit-learn's estimators treat class labels without any order, we used the convenient **LabelEncoder** class to encode the string labels into integers.
- We can use a similar approach to transform the nominal color column of our dataset as well:

```
X = df[['color', 'size', 'price']].values
X
```

```
array([['green', 'M', 10.1],
       ['red', 'L', 13.5],
       ['blue', 'XL', 15.3]], dtype=object)
```

```
color_label_encoder = LabelEncoder()
X[:, 0] = color_label_encoder.fit_transform(X[:, 0])
X
```

```
array([[1, 'M', 10.1],
       [2, 'L', 13.5],
       [0, 'XL', 15.3]], dtype=object)
```

Nominal feature encoding

- We may want to create a new **dummy feature** for each unique value in the nominal feature column.
- In other words, we would convert the color feature into three new features: blue, green, and red. Binary values can then be used to indicate the particular color of a sample; for example, a blue sample can be encoded as blue=1, green=0, red=0. This technique is called **one-hot encoding**.
- In order to perform this transformation, we can use the scikit-learn.preprocessingOneHotEncoder:

```
size_mapping = {'XL': 3, 'L': 2, 'M': 1}
df['size'] = df['size'].map(size_mapping)
X = df[['color', 'size', 'price']].values
X
```

```
array([['green', 1, 10.1],
       ['red', 2, 13.5],
       ['blue', 3, 15.3]], dtype=objec
```

```
color_label_encoder = LabelEncoder()
X[:, 0] = color_label_encoder.fit_transform(X[:, 0])
X
```

```
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

```
from sklearn.preprocessing import OneHotEncoder
one_hot_encoder = OneHotEncoder(categorical_features=[0])
one_hot_encoder
```

```
OneHotEncoder(categorical_features=[0], dtype=<type
'numpy.float64'>,
       handle_unknown='error', n_values='auto', sparse=True
```

```
one_hot_encoder.fit_transform(X).toarray()
```

```
array([[  0. ,   1. ,   0. ,   1. ,  10.1],
       [  0. ,   0. ,   1. ,   2. ,  13.5],
       [  1. ,   0. ,   0. ,   3. ,  15.3]])
```

Nominal feature encoding

- Note that when we initialized the **OneHotEncoder**, we defined the column position of the variable that we want to transform via the categorical_features parameter which is the first column in the feature matrix X.
- The **OneHotEncoder**, by default, returns a sparse matrix when we use the **transform()** method, and we converted the sparse matrix representation into a regular (**dense**) NumPy array for the purposes of visualization via the **toarray()** method.
- Sparse matrices are simply a more efficient way of storing large datasets, and one that is supported by many scikit-learn functions.
  - It is especially useful if it contains a lot of zeros.
- If we want to omit the **toarray()** step, we may initialize the encoder as **OneHotEncoder(...,sparse=False)** to return a regular NumPy array:

```
one_hot_encoder = OneHotEncoder(categorical_features=[0], sparse=False)
```

Nominal feature encoding

- Another way which is more convenient is to create those dummy features via one-hot encoding is to use the **pandas.get_dummies()** method.
- Applied on a **DataFrame**, the **get_dummies()** method will only convert string columns and leave all other columns unchanged:

```
pd.get_dummies(df[['price', 'color', 'size']])
```

	price	size	color_blue	color_green	color_red
0	10.1	1	0	1	0
1	13.5	2	0	0	1
2	15.3	3	1	0	0