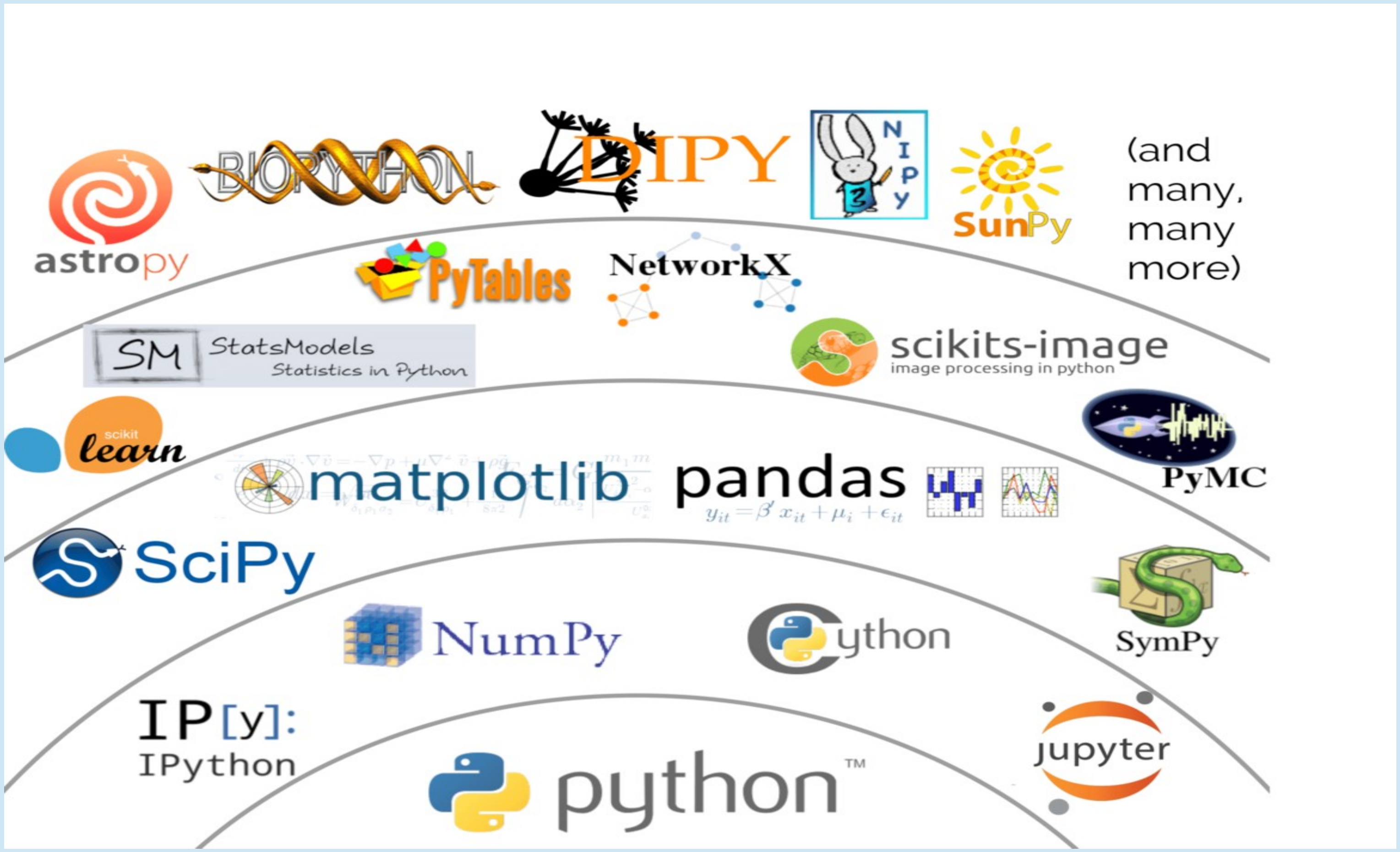
Scikit-Learn Tutorial

Object Oriented Programming and Scripting in Python

The Python Eco-System



Scikit-learn: basic information

- The most relevant activities …
  - classification
  - clustering
  - regression
  - dimensionality reduction

Scikit-learn: basic information

- Classification
  - is the task of identifying the class label of an instance
  - problems with two class labels are called “binary”

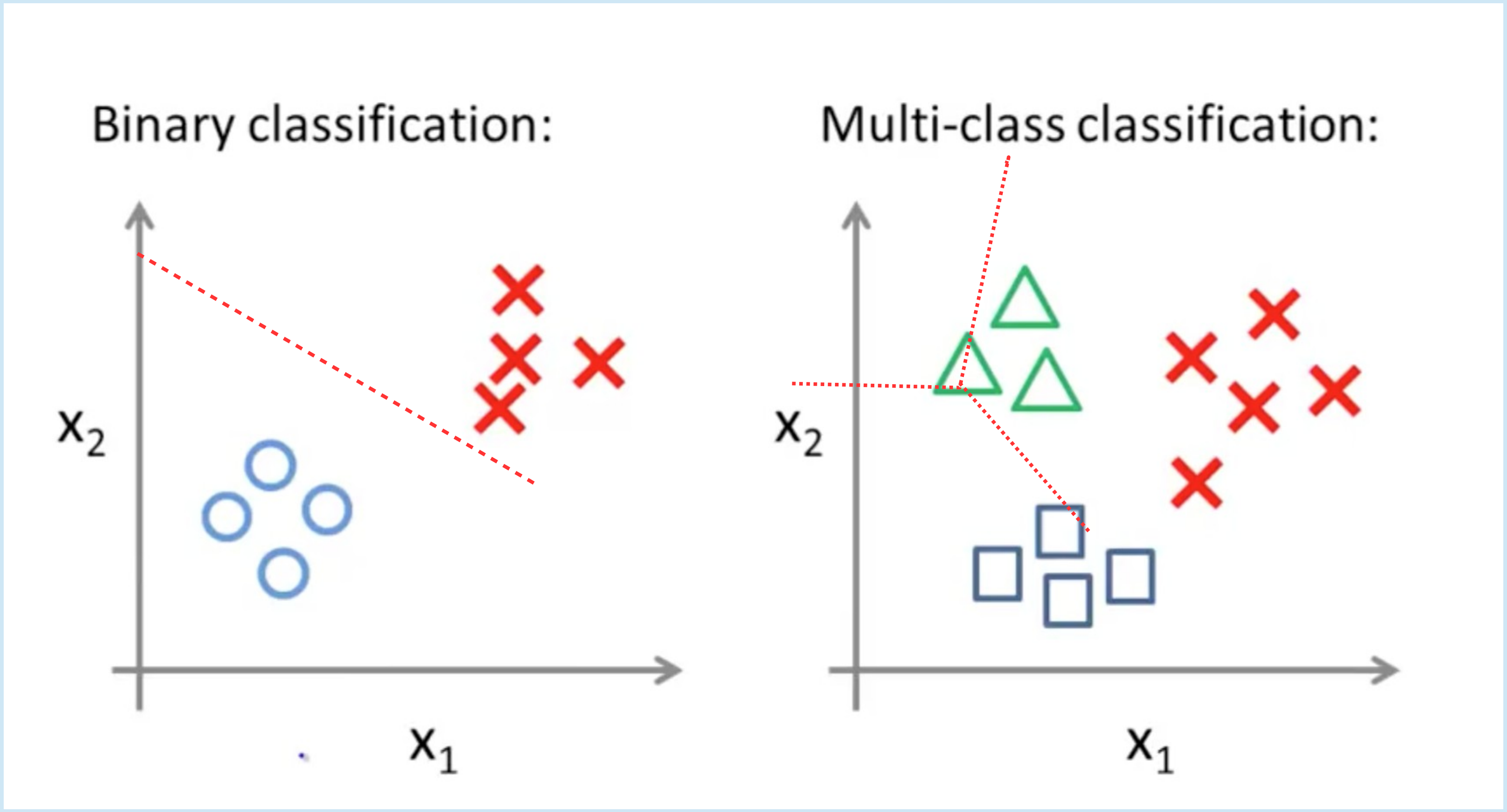
    with more class labels you may have a multiclass or a multilabel problem
  - some examples are needed to train the classifier

Scikit-learn: basic information

- Classification
  - an instance is a 1D array of values
  - each column of the 1D array is a feature
  - the available data are an array of instances, hence data is a 2D array
  - one instance per row and one feature per column
  - targets are the corresponding class labels
  - an example is the combination of an instance and its target

Classification

- Binary vs. multiclass classification



Scikit-learn: basic information

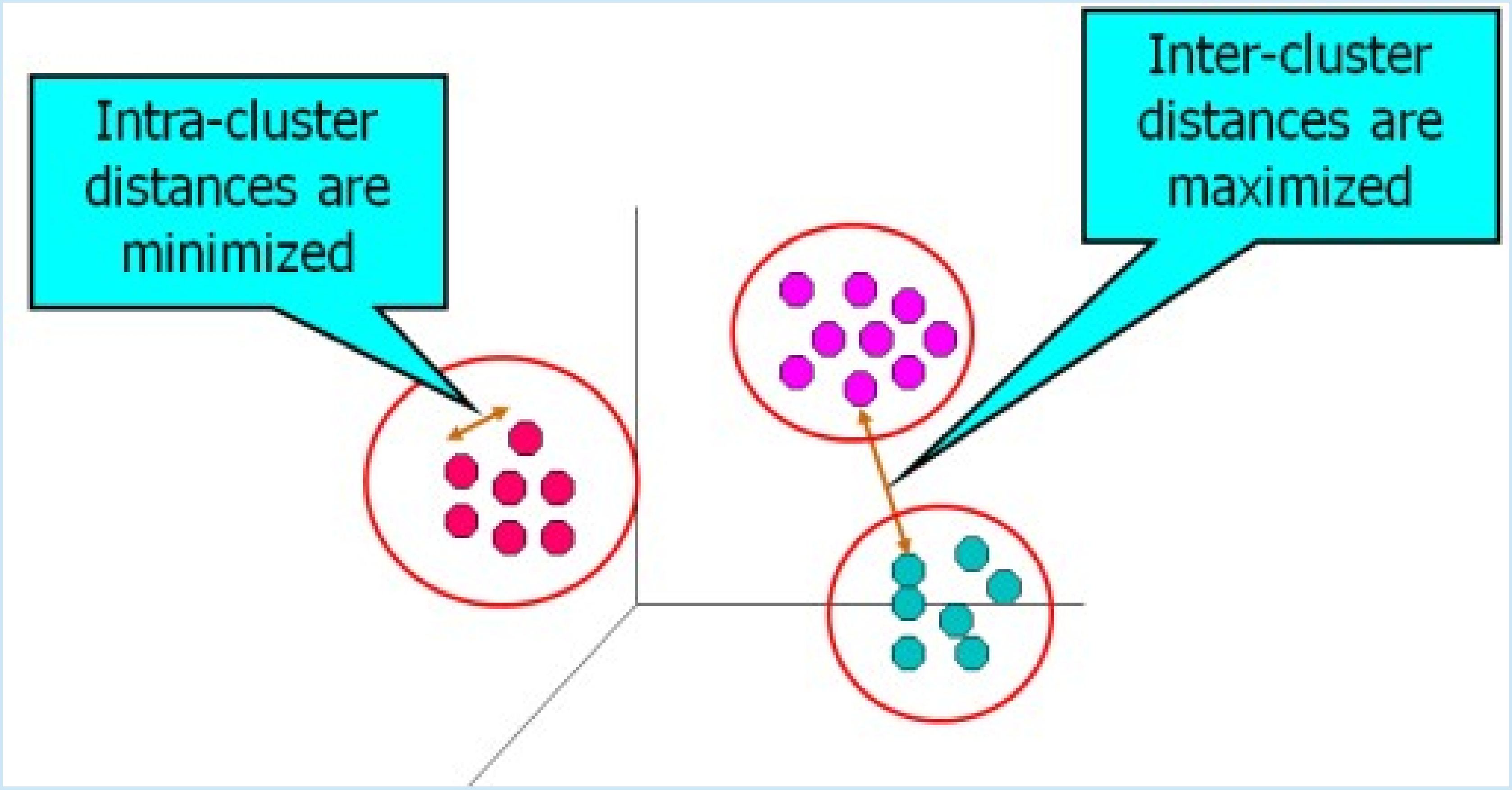
- Classification – an example
  - iris (toy) dataset
    - number of examples: 150
    - number of features: 4 (all real-valued)
    - an instance: [5.1, 3.5, 1.4, 0.2]
    - target names: ['setosa', 'versicolor' 'virginica']
    - an example: [5.1, 3.5, 1.4, 0.2] → setosa

Scikit-learn: basic information

- Clustering
  - Only data are available (i.e., no targets are given)
  - Available data must be partitioned into clusters, based on their “similarity”
  - Most often, similarity means that if two instances are close in the multidimensional space generated by the features, then they should be considered as part of the same cluster

Scikit-learn: basic information

- Clustering in a 3D feature space ...

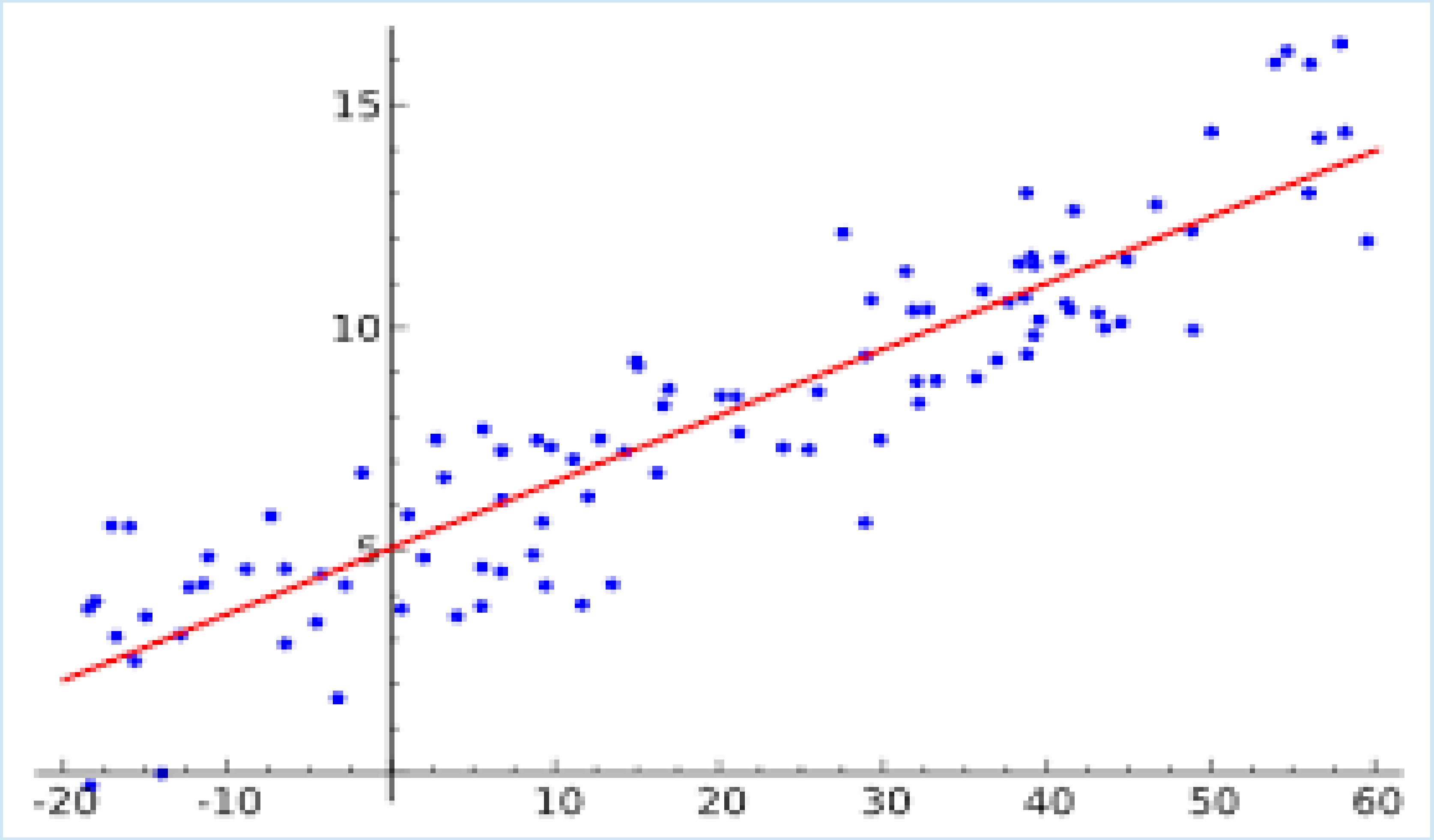

Intra-cluster distances are minimized
Inter-cluster distances are maximized

Scikit-learn: basic information

- Regression
  - aims at estimating the relationships between a dependent variable (i.e., the outcome variable) and one or more independent variables (i.e., the features, also called predictors or covariates)
  - the simplest setting consists of trying to identify the function that generates the outcomes starting from the given feature values

Scikit-learn: basic information

- Regression between a single feature and the outcome ...



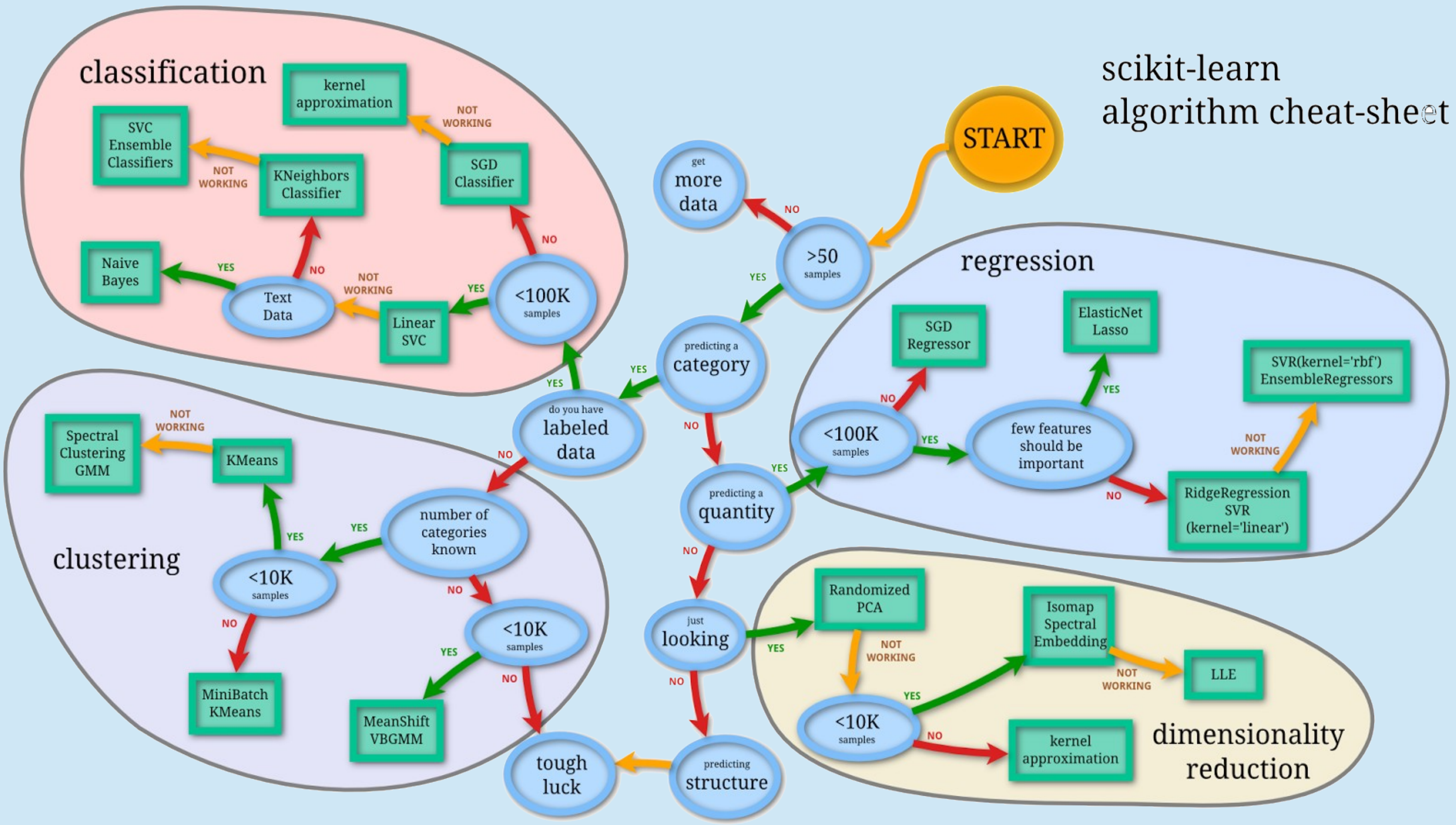
Scikit-learn: basic information

- Dimensionality reduction
  - Sometimes machine learning problems (in particular for classification problems) suffer from the so-called “curse of dimensionality”
  - This issue arises when the number of features is much greater than the number of examples
  - In this case the dimension of the input space is typically reduced with a proper preprocessing technique (i.e., feature selection or feature reduction)
    - Feature selection preserves some features (deemed useful) while dropping the others
    - Feature reduction generates an alternative feature space, whose dimension is reduced with respect to the original one

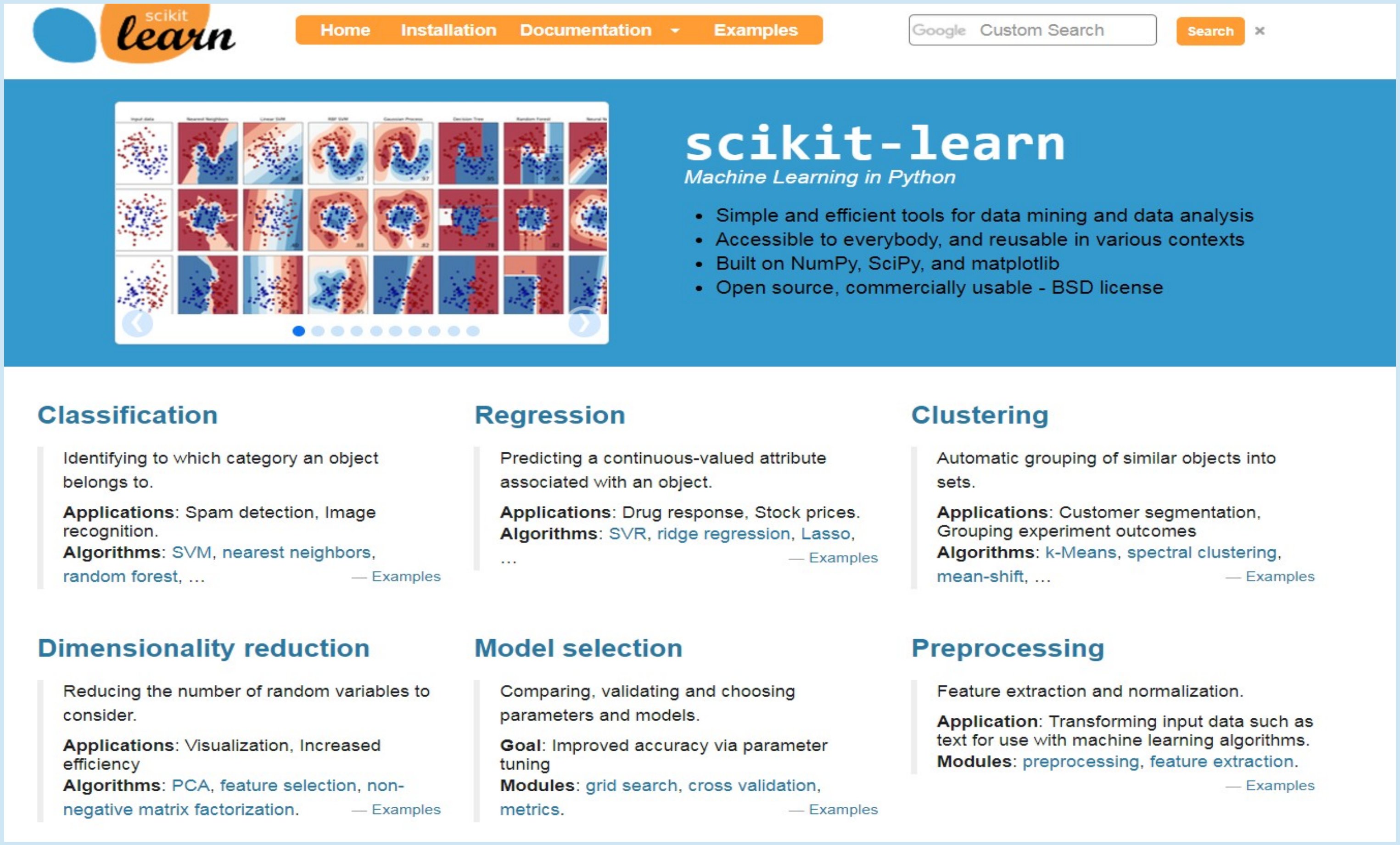
Scikit-learn: basic information

- Dimensionality reduction
  - Principal component analysis (PCA / Kernel PCA)
  - Discriminant analysis (LDA / GDA)
  - Canonical correlation analysis (CCA)
  - ecc.

SKLearn: Which task ?



The main library for data mining



scikit-learn

Machine Learning in Python

- 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

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, … — Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, … — Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, … — Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: PCA, feature selection, non-negative matrix factorization. — Examples

Model selection

Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning
Modules: grid search, cross validation, metrics. — Examples

Preprocessing

Feature extraction and normalization.

Application: Transforming input data such as text for use with machine learning algorithms.
Modules: preprocessing, feature extraction. — Examples

Scikit-learn: basic information

- Machine Learning library
- Designed to inter-operate with NumPy and SciPy
- Focus on:
  - Classification
  - Clustering
  - Regression
  - Dimensionality reduction
  - … etc. ...
- Website: http://scikit-learn.org/
- How to import it in Python

```
>>> import sklearn
```

What is Machine Learning?

- ML is the field of computer science that deals with the study and the development of systems that can learn from data
- Useful definitions
  - Model: the collection of parameters you are trying to fit
  - Data: what you are using to fit the model
  - Target: the value you are trying to predict with your model
  - Features: attributes of your data that will be used in prediction
  - Methods: algorithms that will use your data to fit a model

Learning problem

- A learning problem considers a set of samples of data and then tries to predict properties of unknown data
- Supervised learning
  - If the systems learns from already labeled data (training set) and tries to predict the class of the unknown samples (test set), the task is **classification**
  - If the desired output consists of one or more continuous variables, then the task is **regression**

Learning problem

- Unsupervised learning
  - All samples are unlabeled; hence we can only investigate "hidden" properties of the given samples
- Typical tasks ...
  - Find groups of similar samples (**clustering**)
  - Determine the data distribution (**density estimation**)
  - Project data from high-dimensional space to a low-dimensional space (**dimensionality reduction**)

Datasets: toy data

- Scikit-learn comes with a few standard datasets
- One may use specific functions to download any of them:
  - `load_boston() #Load and return the boston house-prices dataset (regression)`
  - `load_iris() #Load and return the iris dataset (classification)`
  - `load_diabetes() #Load and return the diabetes dataset (regression)`
  - `load_digits([n_class]) #Load and return the digits dataset (classification)`
  - `load_linnerud() #Load and return the linnerud dataset (multivariate regression)`

Datasets: toy data

- A dataset object contains the following fields (*n* be the number of samples and *m* the number of features):
  - **data**: a 2D-array [with shape (n, m)]
  - **feature_names**: a list containing the names of the features [with size m]
  - **target**: an array containing, for each sample, the corresponding class value (for supervised learning) [with size n]
  - **target_names**: an array containing the class names (for supervised learning) [with size at least 2]

Datasets: toy data

- Example of dataset import (iris):

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> dir(iris)
['DESCR', 'data', 'feature_names', 'filename',
'target', 'target_names']
```

Datasets: toy data

- Example of dataset import (iris):

```
>>> from sklearn import datasets
>>> 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], ...
>>> iris.feature_names
    ['sepal length (cm)', 'sepal width (cm)',
     'petal length (cm)', 'petal width (cm)']
>>> iris.target
    array([0, 0, 0, 0, 0, 0, 0, 0, …
```

Datasets: toy data

- Example of dataset import (iris):

```
>>> iris.target
    array([0, 0, 0, 0, 0, 0, 0, 0, …
>>> iris.target_names
   array(['setosa', 'versicolor', 'virginica'],
         dtype='|S10')
```

Datasets: sample images

- Scikit-learn also provides a couple of 2D sample images

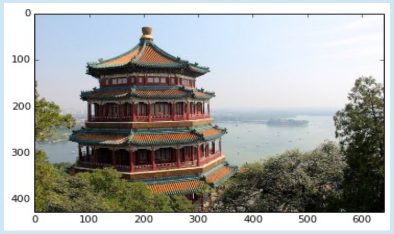
  -
    ```
    #Load the couple of sample images
    load_sample_images()
    ```

  -
    ```
    #Load the numpy array of a single sample image
    load_sample_image(image_name)
    ```

Datasets: sample images

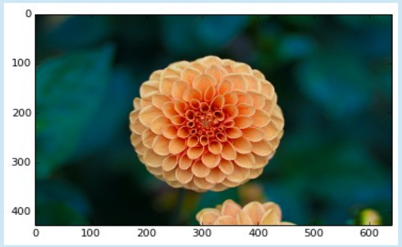
- Examples

Load two images (and show the first one)



```
>>> from sklearn import datasets as ds
>>> from matplotlib import pyplot as pl
>>> images = ds.load_sample_images()
>>> image1, image2 = images.images
>>> pl.imshow(image1)
```

Load a single image (i.e., 'flower.jpg')



```
>>> im = ds.load_sample_image('flower.jpg')
>>> pl.imshow(im)
```

Datasets: Olivetti faces data

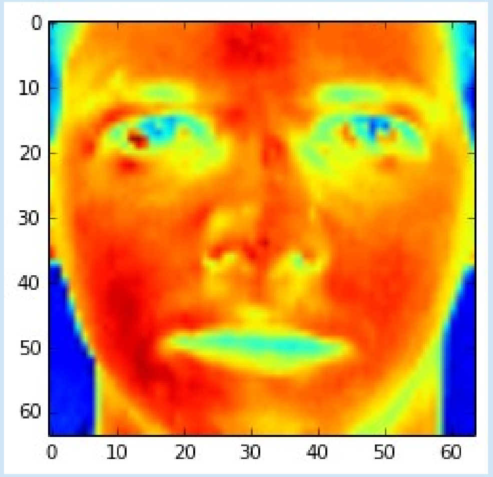
- Set of face images

```
>>> #Load the face dataset (40 subjects, 10
image per subject)
>>> from sklearn import datasets as ds
>>> faces = ds.fetch_olivetti_faces()
```

  - **data**: numpy array of shape (400, 4096), each row being a flatted face image of 64 x 64 pixels
  - **images**: numpy array of shape (400, 64, 64), each row being a face image one subject of the dataset
  - **target**: numpy monodimensional array, each element being a label, ranging from 0 to 39 and corresponding to the subject ID
  - **DESCR**: description of the dataset

Datasets: Olivetti faces data

```
>>> from matplotlib import pyplot as pl
>>> from sklearn import datasets as ds
>>> faces = ds.fetch_olivetti_faces()
>>> pl.imshow(faces.images[0])
>>> print(faces.target[0])
    0
```



Datasets: 20 newsgroups

- Set of 18000 textual posts on 20 topics, splitted in training and test set

```
>>> #Load a list of raw text posts
>>> from sklearn import datasets as ds
>>> subset = 'train/test'
>>> news = fetch_20newsgroups(subset=subset)
```

- **data**: list of raw text posts.
- **filenames**: list of file where data are stored
- **target**: numpy monodimensional array containing the integer index of the associated category
- **target_names**: labels of the categories

Datasets: 20 newsgroups

- Set of 18000 textual posts on 20 topics, splitted in training and test set

```
>>> #Load a list of raw text posts
>>> from sklearn import datasets as ds
>>> subset = 'train/test'
>>> news = ds.fetch_20newsgroups(subset=subset)
>>> #Returns ready-to-use features
>>> vnews = ds.fetch_20newsgroups_vectorized()
```

Datasets: 20 newsgroups

- It is possible to load only a sub-selection of the categories, as follows:

```
>>> cats = ['alt.atheism', 'sci.space']
>>> text = ds.fetch_20newsgroups(subset='train',
                                  categories=cats)
>>> list(text.target_names)
    ['alt.atheism', 'sci.space']
>>> text.filenames.shape
    (1073,)
```

Datasets transformation

- Feature extraction

  Extraction of features from datasets, in a format supported by machine learning algorithms

Note Feature extraction differs from feature selection: the former consists of transforming arbitrary data, such as text or images, into numerical features usable for machine learning. The latter is a technique focused on identifying a useful subset of features (typically used in presence of many features).

Feature extraction

- Loading from dictionary

  The class `DictVectorizer` is used to convert features represented as lists of dict objects to the NumPy/SciPy representation used by scikit-learn estimators
- Useful methods of the DictVectorizer class (X being dicts or mappings of feature objects):
  - `fit_transform(X)`: returns feature vectors (array or sparse matrix)
  - `get_feature_names()`: returns a list of feature name, ordered by indexes
  - `inverse_transform(Y)`: transform array or sparse matrix back to feature mappings

Feature extraction

- Loading from dictionary

```
>>> measurements = [
...   {'city': 'Dubai', 'temperature': 33.},
...   {'city': 'London', 'temperature': 12.},
...   {'city': 'San Fransisco', 'temperature':
18.}
... ]
```

Feature extraction

- Loading from dictionary

```
>>> from sklearn.feature_extraction import
DictVectorizer
>>> v = DictVectorizer()
>>> v.fit_transform(measurements).toarray()
    array([[  1.,    0.,    0.,   33.],
           [  0.,    1.,    0.,   12.],
           [  0.,    0.,    1.,   18.]])
>>> v.get_feature_names()
    ['city=Dubai', 'city=London',
     'city=San Fransisco', 'temperature']
```

Text feature extraction

- Bag of Words

  Documents are described by word occurrences while ignoring the position of the words in the document.

  - Tokenization of strings and indexing of each possible token (e.g., white-spaces and punctuation as token separators).
  - Count of the occurrences of tokens in each document.
  - Normalization and weighting (tokens that occur in the majority of samples / documents are less important).
- Each token occurrence frequency is a feature.
- A corpus can be represented by a matrix with one row per document and one column per token (e.g., per word).

Text feature extraction

- CountVectorizer

  Turns a collection of text documents into numerical feature vectors.

```
>>> from sklearn.feature_extraction.text import
CountVectorizer
>>> c = CountVectorizer()
```

Text feature extraction

- CountVectorizer

```
>>> c = CountVectorizer()
>>> c
    CountVectorizer(analyzer=u'word',
    binary=False, charset=None,
    charset_error=None, decode_error=u'strict',
    dtype=<type 'numpy.int64'>,
    encoding=u'utf-8', input=u'content',
    lowercase=True, max_df=1.0, max_features=None,
    min_df=1, ngram_range=(1, 1),
    preprocessor=None, stop_words=None,
    strip_accents=None,
    token_pattern=u'(u?)\\b\\w\\w+\\b',
    tokenizer=None, vocabulary=None)
```

Text feature extraction

- Tf-Idf term weighting

  In a large text corpus, frequent words carry very little meaningful information about the actual contents of the document (e.g. “the”, “a”, “is” in English).
- In order to re-weight the count features into floating point values suitable for usage by a classifier it is very common to use the **tf–idf transform**.
- **Tf** means term-frequency while **idf** means inverse document-frequency.
- This normalization is performed by the `TfidfTransformer` class.

Text feature extraction

```
>>> from sklearn.feature_extraction.text
        import TfidfTransformer
>>> transformer = TfidfTransformer()
>>> counts = [[3, 0, 1],
...           [2, 0, 0],
...           [3, 0, 0],
...           [4, 0, 0],
...           [3, 2, 0],
...           [3, 0, 2]]
...
>>> tfidf = transformer.fit_transform(counts)
>>> tfidf
  <6x3 sparse matrix of type '<...
  'numpy.float64'>' with 9 stored elements in
  Compressed Sparse ... format>
```

Text feature extraction

```
>>> tfidf.toarray()
array([[ 0.85...,  0.  ...,  0.52...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.55...,  0.83...,  0.  ...],
       [ 0.63...,  0.  ...,  0.77...]])
```

Text feature extraction

- As tf–idf is very often used for text features, there is also another class called `TfidfVectorizer` that combines all the options of CountVectorizer and TfidfTransformer in a single model

```
>>> from sklearn.feature_extraction.text
import TfidfVectorizer
>>> vectorizer = TfidfVectorizer()
>>> vectorizer.fit_transform(corpus)

    <4x9 sparse matrix of type '<...
    'numpy.float64'>' with 19 stored elements
    in Compressed Sparse ... format>
```

Dataset transformation

- Preprocessing

  The preprocessing package provides common utility functions and transformer classes to change raw feature vectors into more suitable representations.
- **Standardization:** several estimators require data with Gaussian distribution with zero mean and unit variance.
- The function scale provides a quick and easy way to perform this operation on a single array-like dataset.

Preprocessing: gaussian scaling

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X = np.array([[ 1., -1.,  2.],
...               [ 2.,  0.,  0.],
...               [ 0.,  1., -1.]])
>>> X_scaled = preprocessing.scale(X)
>>> X_scaled.mean(axis = 0)
    array([ 0.,  0.,  0.])
>>> X_scaled.std(axis = 0)
    array([ 1.,  1.,  1.])
```

Preprocessing: normalization

- **Normalization**: process of scaling individual samples to have unit norm.
- The function **normalize** provides a quick and easy way to perform this operation on a single array-like dataset, either using the **L1** or **L2** norms:

```
>>> X = ( [ [ 1., -1.,  2.],
...       [ 2.,  0.,  0.],
...       [ 0.,  1., -1.] ] )
>>> X_normalized = preprocessing.normalize(X, norm = 'l2')
>>> X_normalized
        array( [ [ 0.40..., -0.40...,  0.81...],
                 [ 1.  ...,  0.  ...,  0.  ...],
                 [ 0.  ...,  0.70..., -0.70...] ] )
```

Preprocessing: binarization

- **Feature binarization**: process of thresholding numerical features to get Boolean values
- The class **Binarizer** provides way to perform this operation on a single array-like dataset:

```
>>> X = ([[ 1., -1.,  2.],
...       [ 2.,  0.,  0.],
...       [ 0.,  1., -1.]])
>>> from preprocessing import Binarizer
>>> b = Binarizer(threshold = 0.0)
>>> X_binarized = b.transform(X)
>>> X_binarized
      array([[ 1.,  0.,  1.],
             [ 1.,  0.,  0.],
             [ 0.,  1.,  0.]])
```

Preprocessing: dimensionality reduction

- If the number of features is high, it may be useful to reduce it with an unsupervised step prior to supervised steps.
- **Principal Component Analysis (PCA)**: linear dimensionality reduction using Singular Value Decomposition, keeping only the most significant vectors to project the data to a lower dimensional space.

Preprocessing: dimensionality reduction

```
>>> from sklearn.decomposition import PCA
>>> X = np.array( [ [-1, -1, 1], [-2, -1, 3],
                    [-3, -2, -1], [1, 1, 4],
                    [2, 1, -2], [3, 2, 0] ] )
>>> pca = PCA(n_components = 2)
>>> pca.fit_transform(X)
    array([[-1.37906599, -0.19483452],
           [-2.67976904,  1.58289587],
           [-3.05951071, -2.64246464],
           [ 0.57960659,  3.41925573],
           [ 2.83966112, -2.22778034],
           [ 3.69907808,  0.06292795]],
    dtype=float32)
```

Feature selection

- Feature selection

  Process of selecting a subset of relevant features, either to improve estimators' accuracy scores or to boost their performance on very high-dimensional datasets
- SkLearn provides a module containing the main algorithms and utilities for feature selection tasks (feature_selection module)

Feature selection

- Functions of feature_selection:
  - SelectPercentile([...])

    select features according to a percentile of the highest scores.
  - SelectKBest([score_func, k])

    select features according to the *k* highest scores.
  - RFE(estimator[, …])

    feature ranking with recursive feature elimination.
  - VarianceThreshold([threshold])

    feature selector that removes all low-variance features.
  - feature_selection.chi2(X, y)

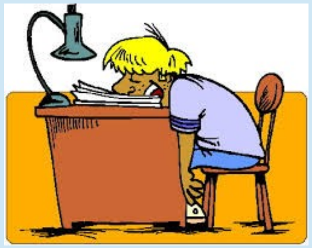
    compute chi-squared statistic for each class/feature combination.
  - ...

Feature selection

- Removing features with low variance:

```
>>> from sklearn.feature_selection import VarianceThreshold as VTh
>>> X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
>>> sel = VTh(threshold = .2)
>>> sel.fit_transform(X)
    array([[0, 1],
           [1, 0],
           [0, 0],
           [1, 1],
           [1, 0],
           [1, 1]])
```

End of part I

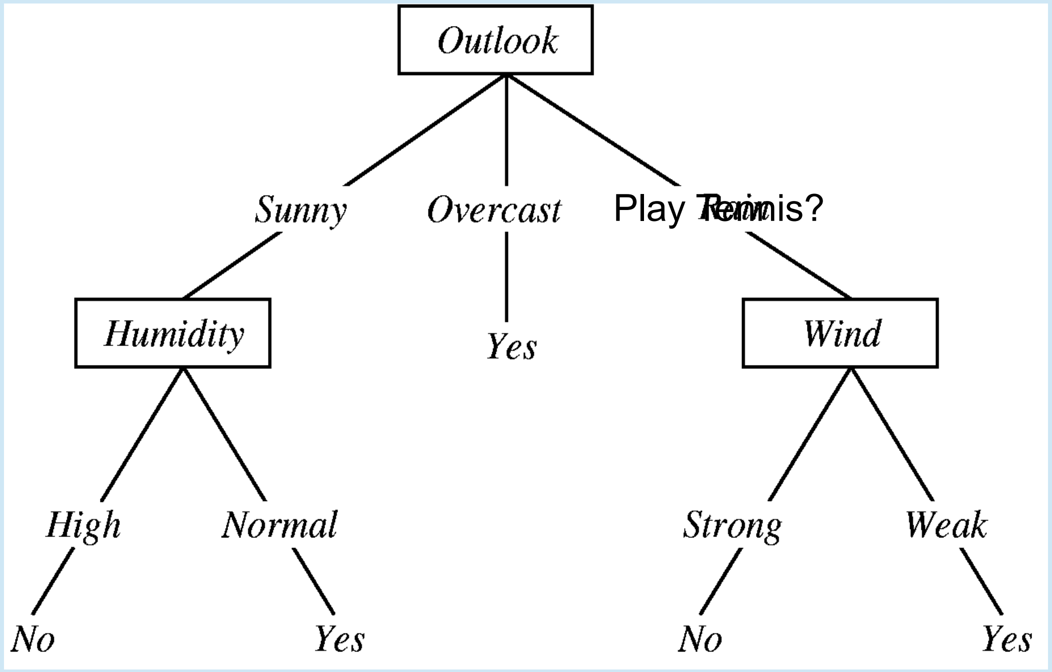


Classification

- In machine learning, classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known.
- Classification is considered an instance of supervised learning, i.e., learning where a training set of correctly identified observations is available.

Decision Trees

- Decision Trees (DTs) are used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. The deeper the tree, the more complex the decision rules and the fitter the model.



Decision Tree Classifier

- DecisionTreeClassifier is a class capable of performing multi-class classification on a dataset.
- A DecisionTreeClassifier takes as input two arrays:
    - an array **X** of shape (n_samples, n_features) holding the training samples, and
    - an array **Y** of integer values, of size n_samples, holding the class labels for the training samples

Decision Tree Classifier

- Example 1

```
>>> from sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, Y) #data fitting
```

- After fitting, the model can then be used to predict new values:

```
>>> clf.predict([[2., 2.]])
    array([1])
```

Decision Tree Classifier

- Example 2

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
>>> iris = load_iris()
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(iris.data, iris.target)
>>> clf.predict(iris.data[0, :])
    array([0])
```

Nearest Neighbors Classification

- A neirest neighbor classifier finds a predefined number of training samples closest in distance to the new point, and predicts the label from these
- The number of samples to be considered for classification can be a user-defined constant (in k-nearest neighbor classifiers), or may vary depending on the local density of points (in radius-based neighbor classifiers)

Nearest Neighbors Classification

- Scikit-learn implements two different nearest neighbors classifiers:
  - `KNeighborsClassifier` implements the kNN classifier
  - `RadiusNeighborsClassifier` implements the radius-based neighbor classifier

```
>>> from sklearn import neighbors
```

Nearest Neighbors Classification

- KNN classifier class

  `KNeighborsClassifier(n_neighbors, weights = weights)`
- `n_neighbors`: the value of *k*
- `weights`: the weighting function for each neighbor:
  - 'uniform': assigns uniform weights to each neighbor
  - 'distance': weights are proportional to the inverse of the distance from the query point
  - alternatively, a user-defined function of the distance can be supplied

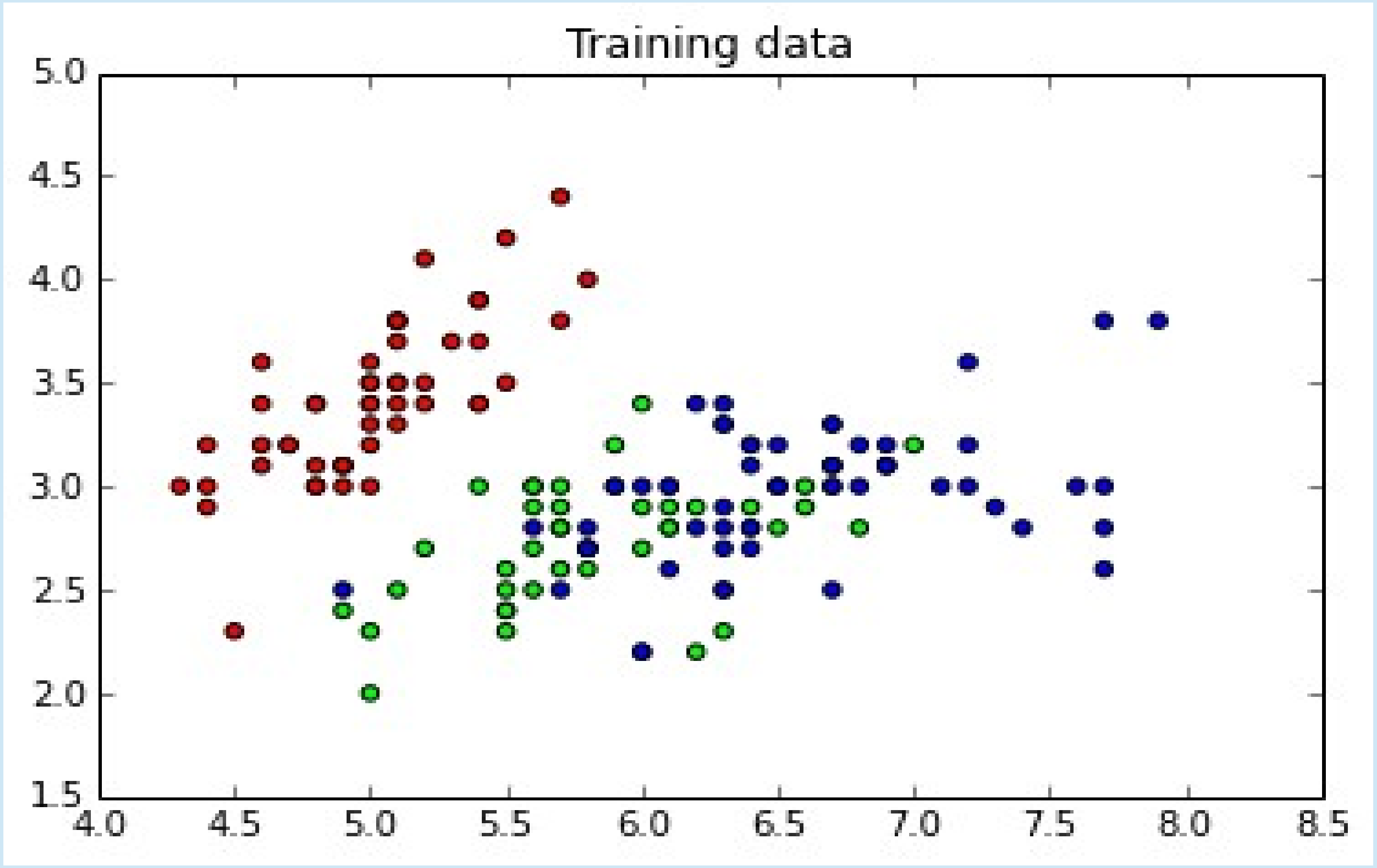
Nearest Neighbors Classification

- Example: iris dataset

```
>>> from matplotlib.colors import ListedColormap
>>> from sklearn import neighbors, datasets
>>> iris = datasets.load_iris()
>>> X = iris.data[:, :2]  #Taking the first 2 features
>>> y = iris.target
>>> colors = ['#FF0000','#00FF00','#0000FF']
>>> cmap_bold = ListedColormap(colors)
>>> F1, F2 = X[:, 0], X[:, 1]
>>> matplotlib.scatter(F1,F2, c = y, cmap = cmap_bold)
>>> matplotlib.title('Training data')
>>> matplotlib.show()
```

Nearest Neighbors Classification

- Example: iris dataset



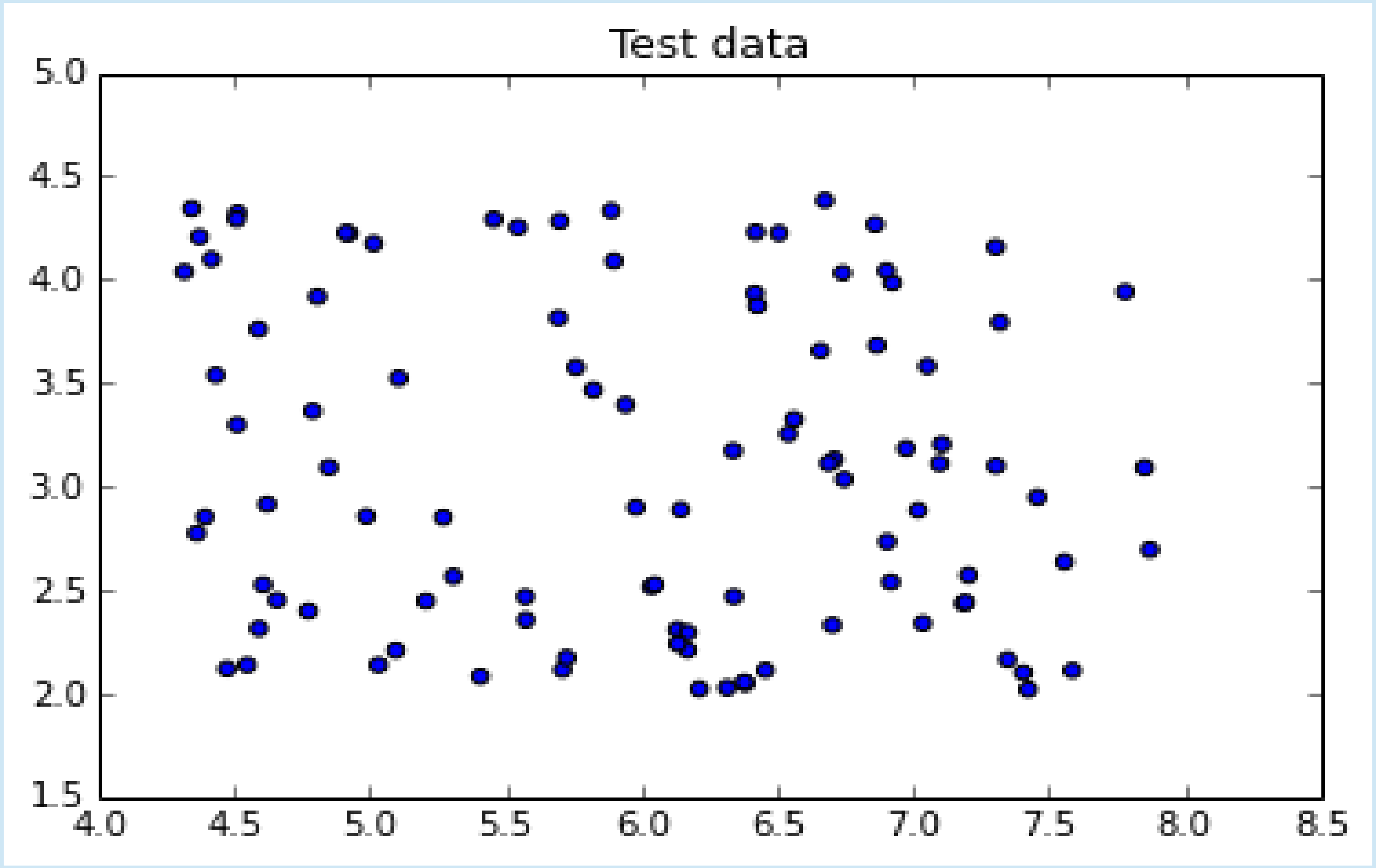
Nearest Neighbors Classification

- Iris dataset

```
>>> clf = neighbors.KNeighborsClassifier(15,
weights='uniform') #Classifier instance
>>> clf.fit(X, y) # Classifier fitting
>>> #generation of random test samples
>>> N = 100
>>> from random import uniform
>>> xx = [uniform(F1.min(),F1.max()) for k in range(N)]
>>> yy = [uniform(F2.min(),F2.max()) for k in range(N)]
>>> matplotlib.scatter(xx, yy)
>>> matplotlib.title('Test data')
>>> matplotlib.show()
```

Nearest Neighbors Classification

- Iris dataset



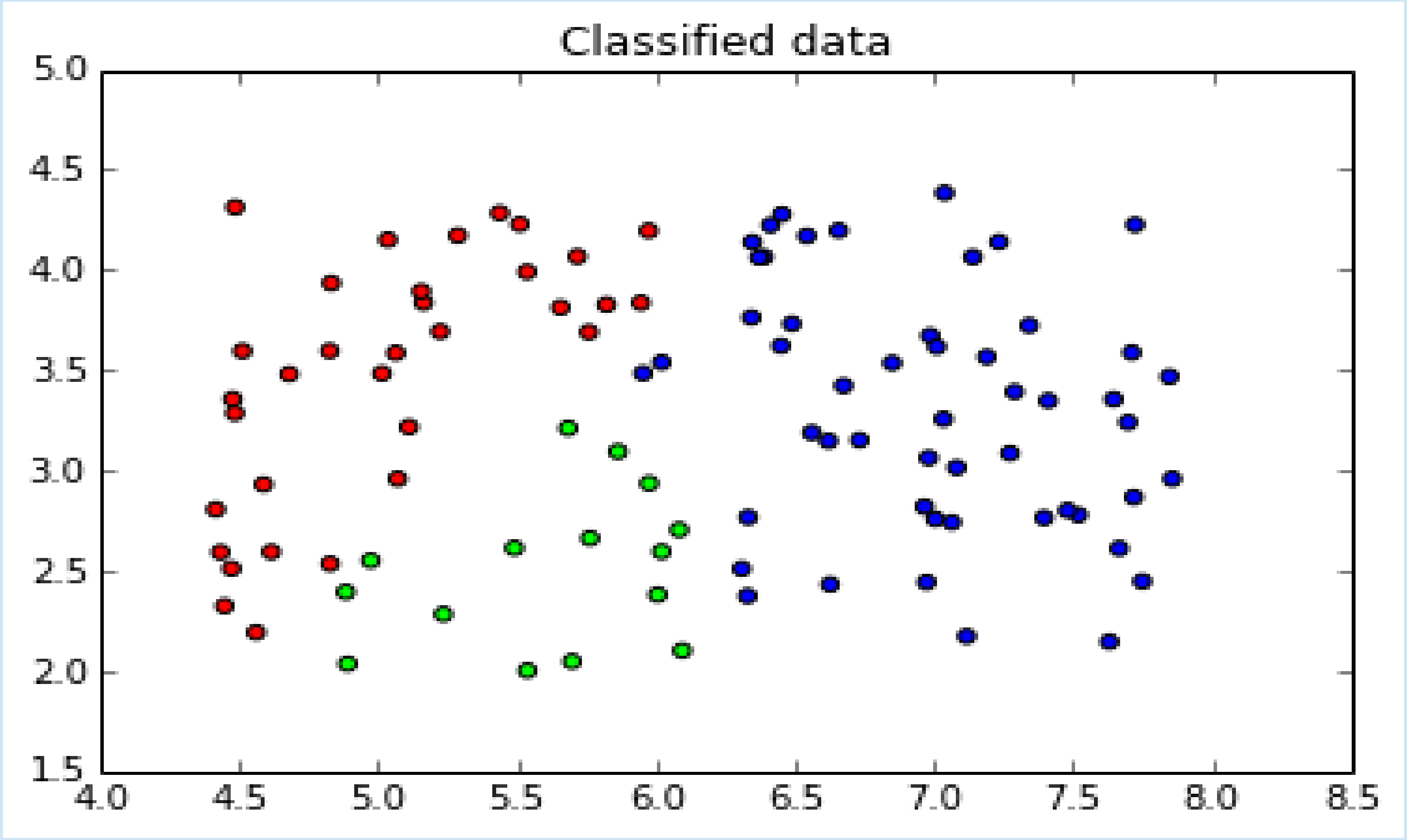
Nearest Neighbors Classification

- Classification

```
>>> points = np.array(zip(xx, yy))#test points
>>> Z = clf.predict(points)#classification
>>> plt.scatter(xx, yy, c=Z, cmap=cmap_bold)
>>> matplotlib.scatter(xx, yy)
>>> matplotlib.title('Classified data')
>>> matplotlib.show()
```

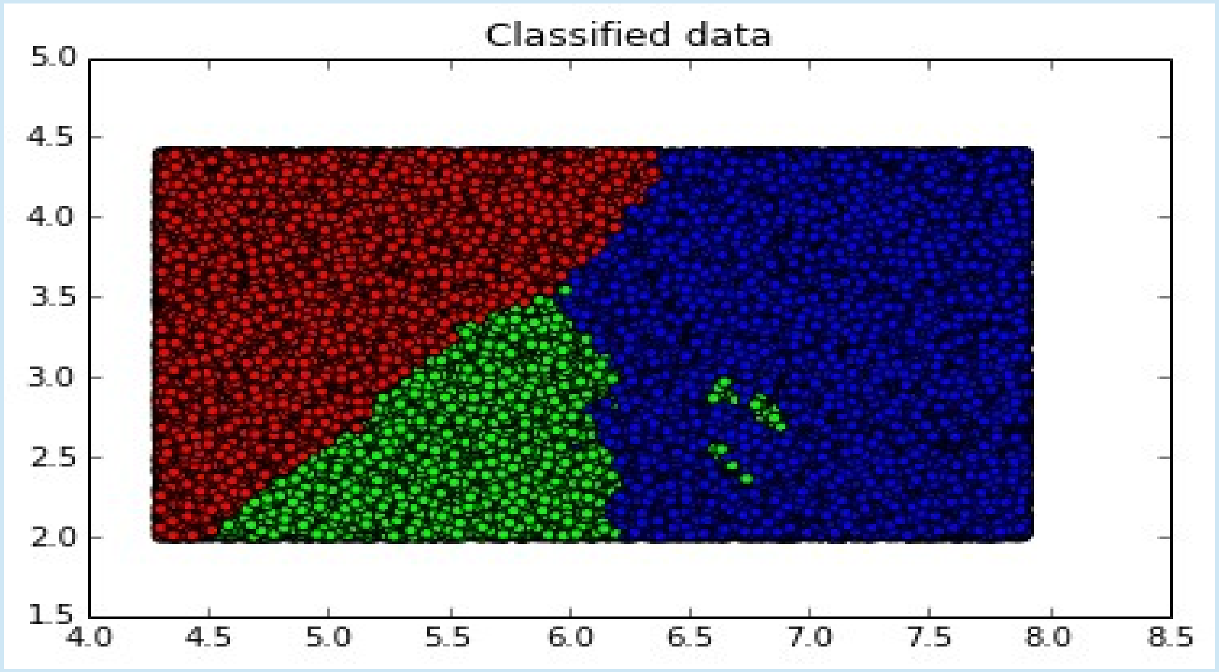
Nearest Neighbors Classification

- Classification



Nearest Neighbors Classification

- Increasing the test points allows to clearly see the classification regions



- For a complete example see

http://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html#example-neighbors-plot-classification-py

Naive Bayes

- Naive Bayes methods are a set of algorithms based on applying the Bayes theorem with the “naive” assumption of independence between every pair of features

Naive Bayes

- Given a class variable y and a dependent feature vector x, Bayes' theorem states the following relationship:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)}$$

- Using the feature independence assumption:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Naive Bayes

- Since $P(x_1, \ldots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

Naive Bayes

- We can use Maximum A Posteriori (MAP) estimation to estimate P(y) and P(x_i | y)
- The former is the relative frequency of class y in the training set
- The various kinds of naive Bayes classifiers differ mainly on the assumptions made regarding the distribution of **P(x_i | y)**

```
>>> from sklearn import naive_bayes
```

Gaussian Naive Bayes

- `GaussianNB` implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma^2_y}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma^2_y}\right)$$

- The parameters s_y and m_y are estimated using maximum likelihood.

Gaussian Naive Bayes

- Example 1

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2],
[1, 1], [2, 1], [3, 2]])
>>> Y = np.array([1, 1, 1, 2, 2, 2])
>>> from sklearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(X, Y)
>>> print(clf.predict([[-0.8, -1]]))
    [1]
```

Gaussian Naive Bayes

- Example 2

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
>>> data, target = iris.data, iris.target
>>> y_pred = gnb.fit(data,target).predict(data)
>>> nrows, ncols = data.shape
>>> nerrors = (target != y_pred).sum()
>>> frmt = "Number of mislabeled points %d (%d)"
>>> print(frmt % (nrows,nerrors))
    Number of mislabeled points 6 (150)
```

Multinomial Naive Bayes

- `MultinomialNB` implements naive Bayes for multinomially distributed data,
- It is one of the two classic naive Bayes variants used in text classification (where data are typically represented as word vector counts).
- The distribution is parametrized by vectors $q_y=(q_{y1}, \ldots, q_n)$ for each class y, where *n* is the number of features (in text classification, it is the size of the vocabulary) and q_{yi} is the probability $P(x_i \mid y)$ for feature i to appear in a sample belonging to class y.
- The parameters q_y are estimated according to a smoothed version of maximum likelihood

Multinomial Naive Bayes

- Example

```
>>> import numpy as np
>>> X = np.random.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.fit(X, y)
>>> print(clf.predict(X[2]))
[3]
```

Bernoulli Naive Bayes

- `BernoulliNB` implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable. Therefore, this class requires samples to be represented as binary-valued feature vectors; if handed any other kind of data, a BernoulliNB instance may binarize its input (depending on the **binarize** parameter).

Bernoulli Naive Bayes

- Example

```
>>> import numpy as np
>>> X = np.random.randint(2, size=(5, 100))
>>> y = np.array([1, 2, 3, 4, 5])
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(X, y)
>>> print(clf.predict(X[2]))
[3]
```

- If X is not binary, the object should be instantiated with the binarize parameter

```
>>> clf = BernoulliNB(binarize=th)
>>> # th is the threshold for binarization
```

Support Vector Machines

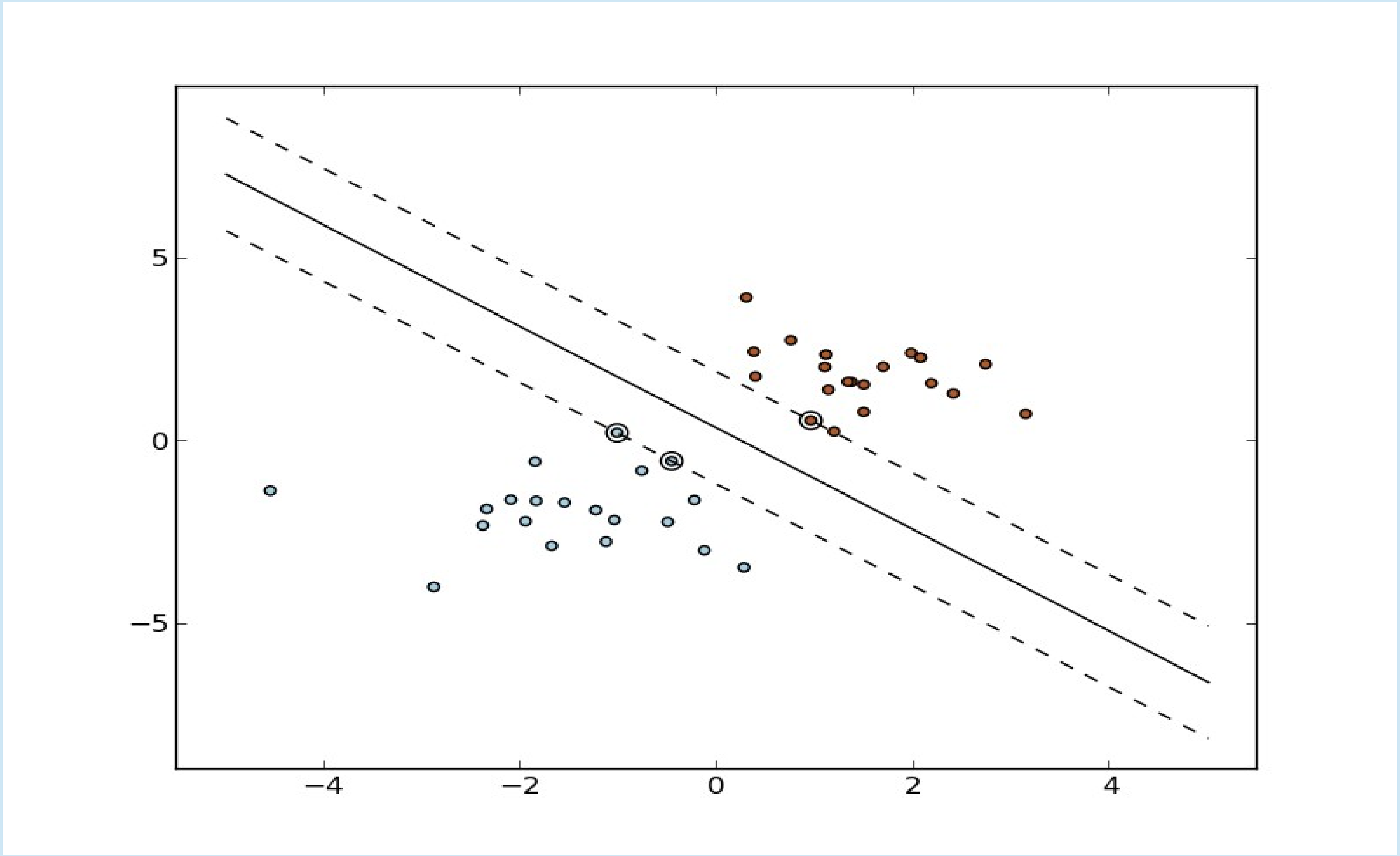
- Support Vector Machines (SVMs) are models with associated learning algorithms that analyze data and recognize patterns. Given a set of training examples, each marked as belonging to a distinct category, an SVM training algorithm builds a model that assigns new examples into one category, making it a non-probabilistic binary linear classifier.

Support Vector Machines

- A SVM constructs a hyper-plane (or a set of hyper-planes) in a high dimensional space. Intuitively, a good separation is achieved by the hyper-plane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

Support Vector Machines

- SVM



Support Vector Machines

- **SVC, NuSVC** and LinearSVC are classes capable of performing multi-class classification on a dataset.
- **SVC** and **NuSVC** are similar methods, but accept slightly different sets of parameters and have different mathematical formulations (see http://scikit-learn.org/stable/modules/svm.html for the documentation). **LinearSVC** is another implementation of Support Vector Classification for the case of a linear kernel.

Support Vector Machines

- Example: binary classification

```
>>> from sklearn import svm
>>> X = [[0, 0], [1, 1]]
>>> y = [0, 1]
>>> clf = svm.SVC()
>>> clf.fit(X, y)
>>> clf.predict([[2., 2.]])
    array([1])
```

Support Vector Machines

- SVMs decision function depends on some subset of the training data, called the support vectors. Some properties of these support vectors can be found with the following methods:

```
>>> #get support vectors
>>> clf.support_vectors_
    array([[ 0.,  0.],
           [ 1.,  1.]])
>>> #get indices of support vectors
>>> clf.support_
    array([0, 1])
>>> #get number of support vectors for each class
>>> clf.n_support_
    array([1, 1])
```

Support Vector Machines

- Multi-class classifications: **SVC** and **NuSVC** implement the "one-against-one" approach for multi- class classification. If ***C*** is the number of classes, then ***C (C - 1)/2*** classifiers are constructed and each one trains data from two classes:

```
>>> #get support vectors
>>> X = [[0], [1], [2], [3]]
>>> Y = [0, 1, 2, 3]
>>> clf = svm.SVC()
>>> clf.fit(X, Y)
>>> dec = clf.decision_function([[0.3]])
>>> dec.shape[1] #4 classes: 4*3/2 = 6
    6
>>> clf.predict([0.3])
    [0]
```

Support Vector Machines

- Multi-class classifications: **LinearSVC** implements “one-vs-the-rest” multi-class strategy, thus training ***C*** models. If there are only two classes, only one model is trained:

```
>>> #get support vectors
>>> X = [[0], [1], [2], [3]]
>>> Y = [0, 1, 2, 3]
>>> clf = svm.SVC()
>>> clf.fit(X, Y)
>>> dec = clf.decision_function([[0.3]])
>>> dec.shape[1]#4 classes: 4 models
    4
>>> clf.predict([0.3])
    [0]
```

Ensemble methods

- Ensemble: combination of the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.

Ensemble methods

- Two families of ensemble methods are usually distinguished:
  - Averaging: the driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.

    Examples: Bagging, Random Forest,...
  - Boosting: base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.

    Examples: AdaBoost, Gradient Tree Boosting, ...

Bagging classifier

- Each classifier is trained on random subsets of the original training set, with replacement (bootstraping) or not.
- BaggingClassifier offers a unique class for performing the bagging algorithms:
    - Pasting: each training set is a random subset of original training set
    - Bagging: each random subset is drawn with replacements of original samples
    - Random Subspacing: each training set is drawn as random subset of features (with bootstraping on feature or not)
    - Random Patches: each classifier is built on subsets of both samples and features

Bagging classifier

- BaggingClassifier parameters:
  - **base_estimator** (optional, default=None) : the base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.
  - **n_estimators** (optional, default=10): the number of base estimators in the ensemble.
  - **max_samples** (optional, default=1.0): the number of samples to draw from X to train each base estimator.
  - **max_features** (optional, default=1.0): the number of features to draw from X to train each base estimator.
  - **bootstrap** (optional; default=True): whether samples are drawn with replacement.
  - **bootstrap_features** (optional; default=False): whether features are drawn with replacement.

Bagging classifier

- **Example:**

```
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.neighbors import
KNeighborsClassifier as KNeighbors
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> bagging = BaggingClassifier(KNeighbors(15))
>>> bagging.fit(X, y)
>>> bagging.predict([[5.4, 1.5], [3.6, 5.1]])
    array([1, 0])
```

Random forest classifier

- The algorithm is based on randomized decision trees. It fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. Each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.

Random forest classifier

- RandomForestClassifier main parameters:
  - **n_estimators** (optional, default=10): the number of trees in the forest.
  - **max_features** (optional, default='auto'): the number of features to consider when looking for the best split (values are integers, float, 'sqrt', 'log2', 'None', 'auto')
  - **bootstrap** (optional; default=True): whether bootstrap samples are used when building trees.

Random forest classifier

- Example:

```
>>> from sklearn.ensemble import
RandomForestClassifier as RandomForest
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> rf = RandomForest(n_estimators = 10)
>>> rf.fit(X, y)
>>> rf.predict([[5.4, 1.5], [3.6, 5.1]])
    array([1, 0])
```

AdaBoost classifier

- Fitting sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data.
- The predictions are combined through a weighted majority vote (or sum) to produce the final prediction.
- The data modifications at each iteration consist of applying weights to each of the training samples. Initially, those weights are all set for simply training a weak learner on the original data.
- For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the re-weighted data (boosting).

AdaBoost classifier

- At a given step, those training examples that were incorrectly predicted at the previous step have their weights increased
- Weights are decreased for those that were predicted correctly
- Examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence training set

Random forest classifier

- Example:

```
>>> from sklearn.ensemble import
AdaBoostClassifier as AdaBoost
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> ada = AdaBoost(n_estimators = 100)
>>> ada.fit(X, y)
>>> ada.predict([[5.4, 1.5], [3.6, 5.1]])
    array([1, 0])
```

Multi-class vs multi-label classification

- **Multiclass**: classification with more than two classes. Each sample is assigned to one (and only one) label.
- **Multilabel**: each sample is assigned at a set of target labels.
- **Multioutput-multiclass** / **multi-task**: a single estimator has to handle several joint classification tasks. This is a generalization of the multi-label classification task, where the set of classification problem is restricted to binary classification, and of the multi-class classification task. The output format is a 2d numpy array.

Multi-class vs multi-label classification

- **sklearn.multiclass**: meta-estimators to solve multiclass and multilabel classification problems by decomposition into binary classification problems. **Warning!** All classifiers in scikit-learn do multiclass classification out-of-the-box.

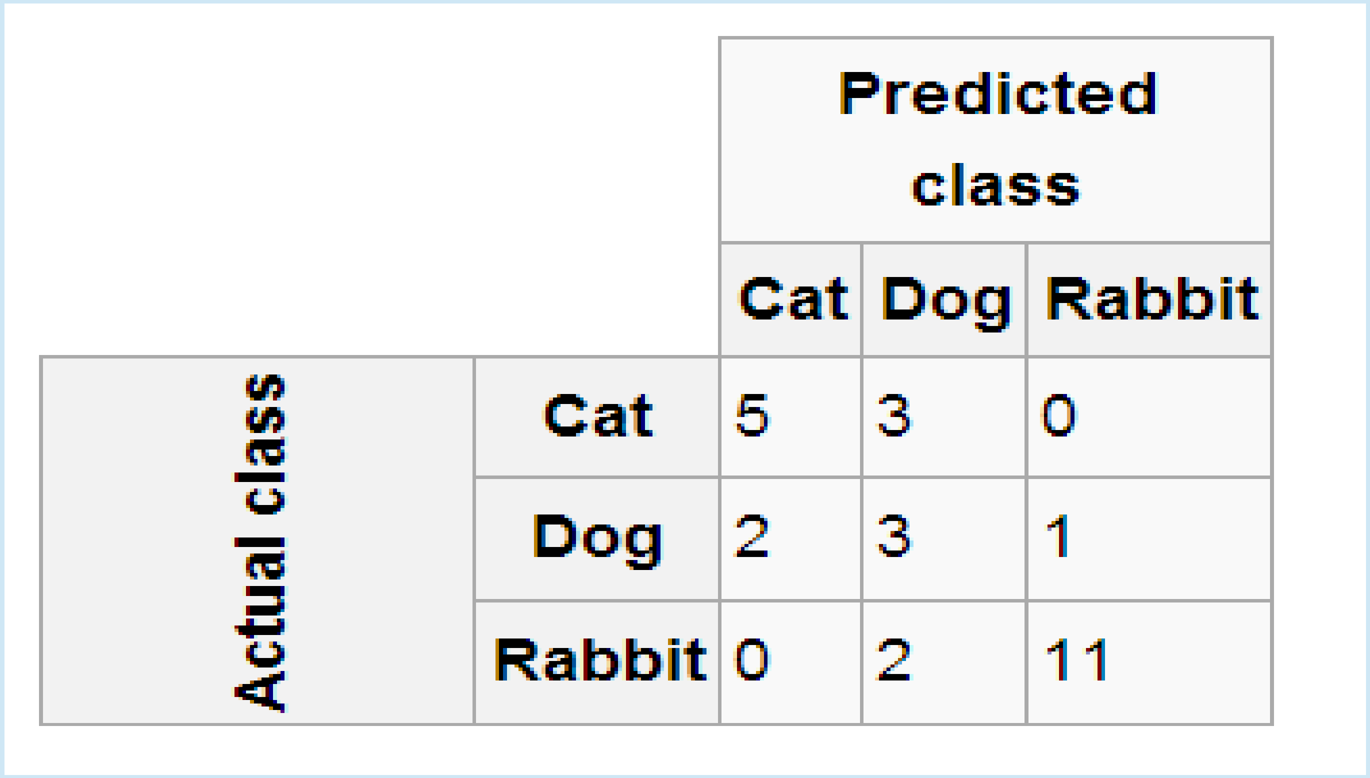
End of part II

Classifier evaluation

- Evaluation is necessary in order to compare different classifiers. How do we measure their performance? The module metrics offers objects and functions aimed at the evaluation for classification, regression, clustering, etc.
- Most well-known metrics relies on the confusion matrix

Classifier evaluation

- Confusion Matrix



		Predicted class		
		Cat	Dog	Rabbit
Actual class	Cat	5	3	0
	Dog	2	3	1
	Rabbit	0	2	11

Each column of the represents the instances in a predicted class, while each row represents the instances in an actual class.

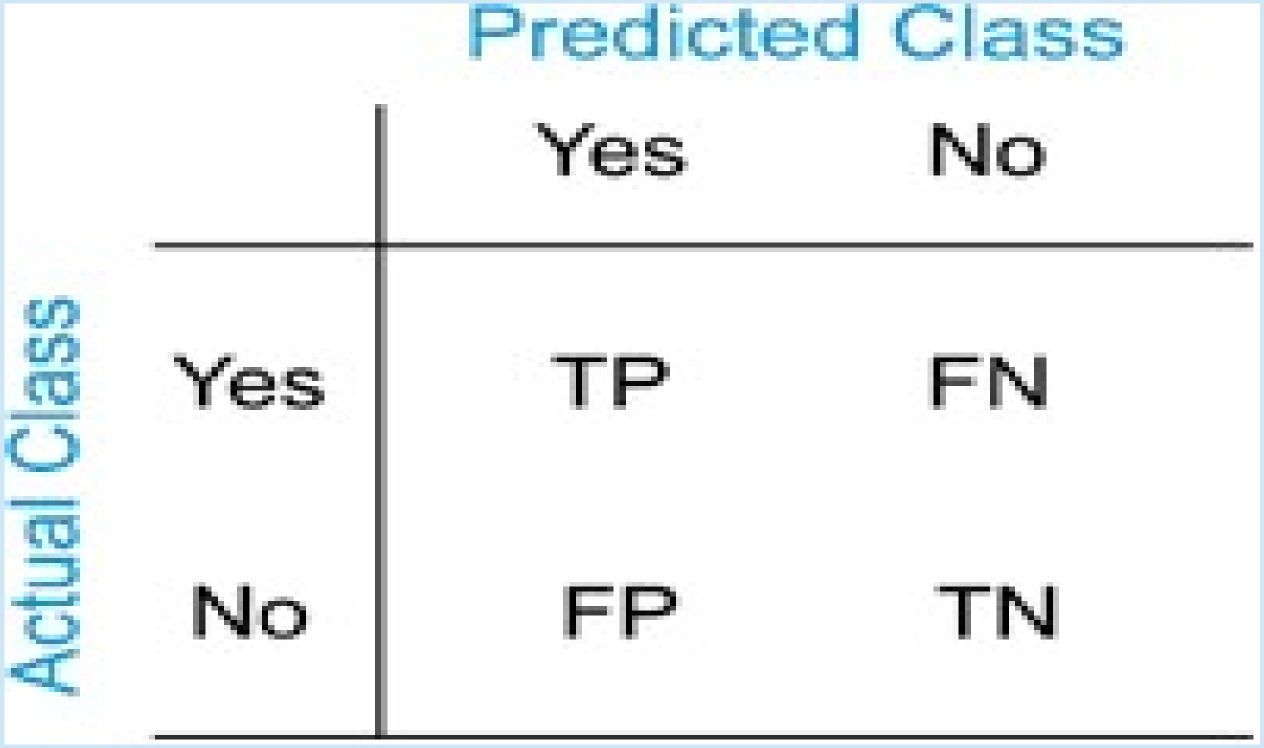
Classifier evaluation: confusion matrix

- Exampe of confusion matrix code:

```
>>> from sklearn.metrics import
confusion_matrix as cMatrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> cMatrix(y_true, y_pred)
    array([[2, 0, 0],
           [0, 0, 1],
           [1, 0, 2]])
```

Classifier evaluation: confusion matrix

- For a given category, the term confusion matrix is referred to a table 2x2 that reports the number of false positives (FP), false negatives (FN), true positives (TP), and true negatives (TN).



		Predicted Class	
		Yes	No
Actual Class	Yes	TP	FN
	No	FP	TN

Classification metrics

- For binary classifiers ...
  - Accuracy $$a=\frac{TP+TN}{TP+FP+FN+TN}$$
  - Precision $$p=\frac{TP}{TP+FP}$$
  - Recall $$r=\frac{TP}{TP+FN}$$
  - F1 score $$f=\frac{2\cdot p\cdot r}{p+r}$$

Classification metrics

- Multiclass classifiers: there are 2 conventional methods for calculating precision and recall (let C be the number of classes, and *i* ranging from 1 to C)
  - Micro-precision $$p=\frac{\sum_i TP_i}{\sum_i (TP_i+FP_i)}$$
  - Micro-recall $$r=\frac{\sum_i TP_i}{\sum_i (TP_i+FN_i)}$$
  - Macro-precision $$P=\frac{\sum_i p_i}{C}$$
  - Macro-recall $$R=\frac{\sum_i r_i}{C}$$

Classification metrics

- Accuracy:

```
>>> from sklearn.metrics import accuracy_score
as accuracy
>>> y_true = [0, 2, 1, 3]
>>> y_pred = [0, 1, 2, 3]
>>> #percentage of correctly classified
samples
>>> accuracy(y_true,y_pred)
    0.5
>>> #number of correctly classified samples
>>> accuracy(y_true,y_pred,normalize='False')
    0.5
```

Classification metrics

- Precision: the parameter `average` determines the type of averaging performed on the data. It can assume the values
  - None: the scores for each class are returned
  - micro: micro-precision is computed
  - macro: macro-precision is computed
  - weighted: the default value; the terms in the macro-precision are weighted by the number of correct instances for each class
  - samples: calculates metrics for each instance, and finds their average (only meaningful for multilabel classification)

Classification metrics

- Precision

```
>>> from sklearn.metrics import
precision_score as precision
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> precision(y_true, y_pred, average = None)
    array([ 0.66666667,  0.,  0.])
>>> precision(y_true,y_pred,average='micro')
    0.33333333333333331
```

Classification metrics

- Recall (same parameters of precision):

```
>>> from sklearn.metrics import recall_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> recall_score(y_true, y_pred, average =
None)
    array([ 1.,  0.,  0.])
>>> recall_score(y_true, y_pred, average =
'micro')
    0.33333333333333331
```

Classification metrics

- F1-score (same parameters of precision)

```
>>> from sklearn.metrics import f1_score as F1
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> F1(y_true, y_pred, average = None)
    array([ 0.8,  0.,  0.])
>>> F1(y_true,y_pred,average='micro')
    0.33333333333333331
```

Classification metrics

- The main classification metrics are summarized by the method `classification_score`

```
>>> from sklearn.metrics import
classification_report
>>> y_true = [0, 1, 2, 0, 2, 2]
>>> y_pred = [0, 2, 1, 1, 0, 1]
>>> names = ['class 0', 'class 1', 'class 2']
#names matching the labels (optional)
```

Classification metrics

- `>>> classification_report(y_true, y_pred, target_names = names)`

```
                precision    recall  f1-score
    support


    class 0       0.50      0.50      0.50
    2
    class 1       0.33      1.00      0.50
    1
    class 2       1.00      0.33      0.50
    3
avg / total       0.72      0.50      0.50
    6
```

(**support** is the number of of occurrences of each class in **y_true**)

Dataset Splitting

- Definition of a dataset to test the model in the training phase (*validation set*), in order to avoid overfitting.
- Module **cross_validation**.

Dataset Splitting

- Training set splitting

```
>>> from sklearn import cross_validation,
datasets
>>> iris = datasets.load_iris()
>> X_train, X_test, y_train, y_test =
cross_validation.train_test_split(iris.data,
iris.target, test_size=0.4) #40% of training
set is randomly taken as validation set
>>> X_train.shape, y_train.shape
    ((90, 4), (90,))
>>> X_test.shape, y_test.shape
    ((60, 4), (60,))
```

Cross validation

- Splitting the training set is often not the best choice:
  - the number of samples which can be used for learning the model is drastically reduced
  - the results can depend on a particular random choice for the pair of (train, validation) sets.
- **K-Fold cross validation**: the training set is split into k smaller sets (folds).
- For each of the k “folds”:
  - a model is trained using k-1 folds as training data;
  - the remaining fold is used as a test set to validate the model (e.g., by computing the accuracy).
  - the performance measure for each fold is then the averaged.

Cross validation

- K-Fold cross validation:

```
>>> from sklearn import cross_validation,
datasets, svm
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf = svm.SVC()
>>> cross_validation.cross_val_score(clf, X,
y, cv=5) #cv is the number of folds, I it is
integer
    array([ 0.96666667,  1.,  0.96666667,
0.96666667,  1.])
#the result array contains the accuracies
(default) of each fold
```

Cross validation

- The parameter **scoring** permits to compute different metrics (e.g., giving the values 'precision', 'recall', or 'f1')

```
>>> from sklearn import cross_validation,
datasets, svm
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf = svm.SVC()
>>> cross_validation.cross_val_score(clf, X,
y, cv=5, scoring = 'precision') #cv is the
number of folds, I it is integer
    array([ 0.96969697,  1.,  0.96969697,
0.96969697,  1.])
```

Cross validation: iterators

- The module **cross_validation** provides utilities to generate indices useful for splitting datasets for different cross validation methods
  - Kfold: creates arrays of indices for k-fold c.v.
  - LeaveOneOut: each test set fold contains only one sample.
- The LOO is equal to a *n*-fold c.v., *n* being the number of data samples.
  - LeavePOut: similar to LOO, but each test set contains *p* samples.

Cross validation: iterators

- Example

```
>>> from sklearn.cross_validation import KFold
>>> kf = KFold(4,n_folds=2) #4 samples,2 folds
>>> for train, test in kf:
...     print("%s %s" % (train, test))
    [2 3] [0 1]
    [0 1] [2 3]
```

Regression

- The target value is expected to be a linear combination of the input variables. In mathematical notion, if ŷ is the predicted value.

$$\hat{y}(w,x)=w_0+w_1x_1+\ldots+w_px_p$$

  Across the module, we designate the vector $w=(w_1,\ldots,w_p)$ as **coef_** and w_0 as **intercept_**.

- **Linear regression**: fits a linear model with coefficients $w=(w_1,\ldots,w_p)$ to minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation.

Regression

- Example of linear regression

```
>>> from sklearn import linear_model
>>> clf = linear_model.LinearRegression()
>>> clf.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
>>> clf.coef_
    [0.5  0.5]
>>> clf.intercept_
    2.22044604925e-16
```

Regression: evaluation metrics

- Evaluation metrics for regression models (in module **metrics**):
  - explained_variance_score(y_true, y_pred): explained variance regression score function
  - mean_absolute_error(y_true, y_pred): mean absolute error regression loss
  - mean_squared_error(y_true, y_pred[,...]) Mean squared error regression loss

...

Clustering

- In machine learning, clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters).
- Clustering is an instance of unsupervised learning, i.e., learning where the data are unlabeled.

Clustering

- Clustering of unlabeled data can be performed with the module **sklearn.cluster**.
- Each clustering algorithm comes in two variants: a *class*, that implements the fit method to learn the clusters on train data, and a *function*, that, given train data, returns an array of integer labels corresponding to the different clusters. For the class, the labels over the training data can be found in the **labels_** attribute.

K-means clustering

- The **KMeans** algorithm clusters data by trying to separate samples in *n* groups of equal variance, in which each sample belongs to the cluster with the nearest mean.
- The means are commonly called the cluster “centroids”.
- Number of clusters (*k*) need to be specified.

K-means clustering

- The **KMeans** algorithm
  1) Centroids are initially selected from the samples.
  2) Each sample is assigned to the nearest centroid.
  3) New centroids are created with the mean of all samples assigned to each previous centroid
  4) Iteration of (2) and (3) until centroids do not move significantly

K-means clustering

- Example

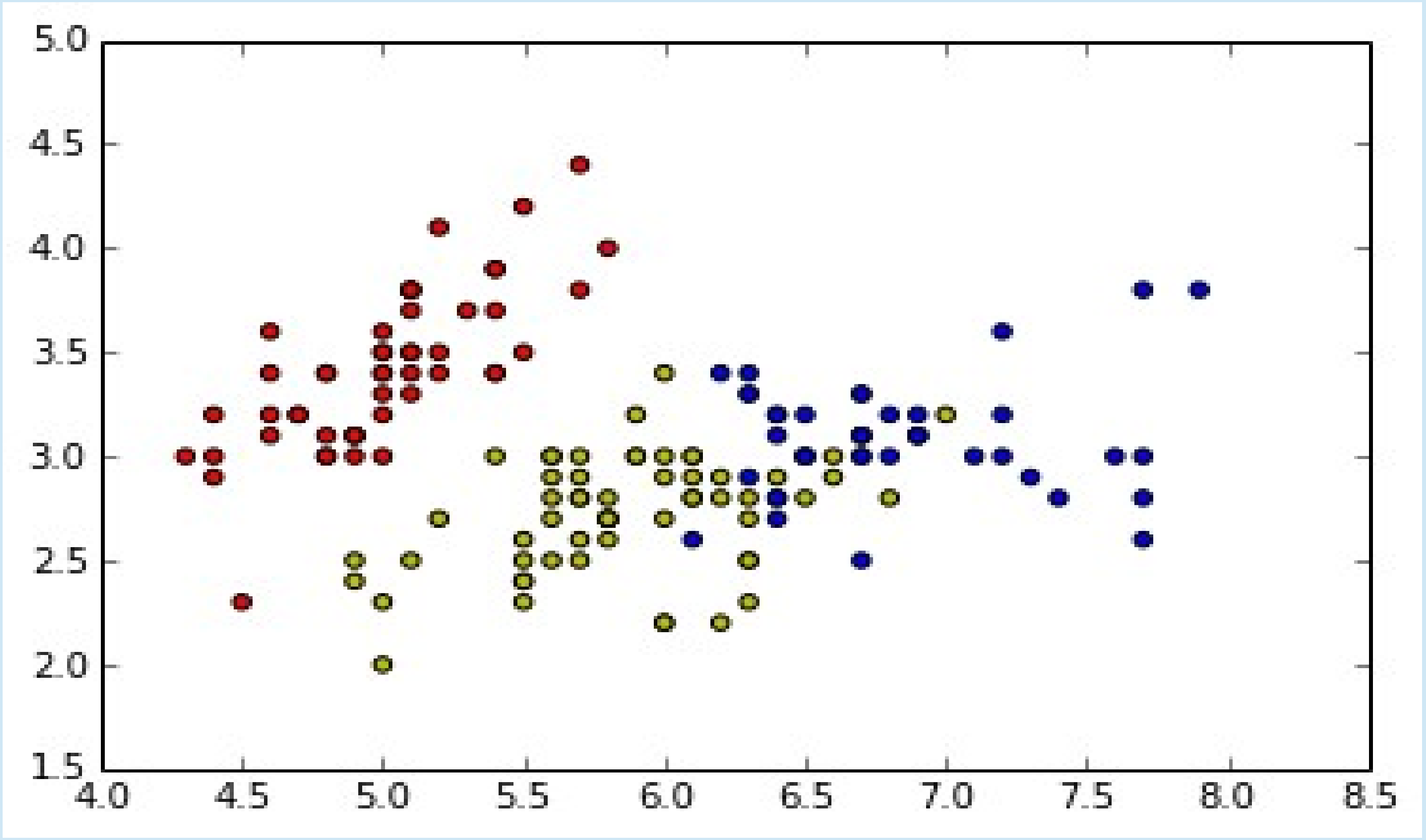
```
>>> import matplotlib.pyplot as plt
>>> from sklearn import datasets
>>> from sklearn.cluster import KMeans
>>> from matplotlib.colors import
ListedColormap as ColorMap

>>> iris = datasets.load_iris()
>>> X = iris.data[:, :2]
>>> cmap_light = ColorMap(['r','b', 'y'])

>>> k_means = KMeans(n_clusters = 3)
>>> k_means.fit(X)
>>> y_pred = k_means.predict(X)
>>> plt.scatter(X[:, 0], X[:, 1], c =
y_pred, cmap = cmap_light);
```

K-means clustering

- Example



Spectral clustering

- **SpectralClustering** does a low-dimension embedding of the affinity matrix between samples, followed by a KMeans in the low dimensional space.
- The number of clusters needs to be specified.
- When calling **fit**, an affinity matrix is constructed. Alternatively, using '**precomputed'** for the **affinity** parameter, a user-provided affinity matrix can be used.

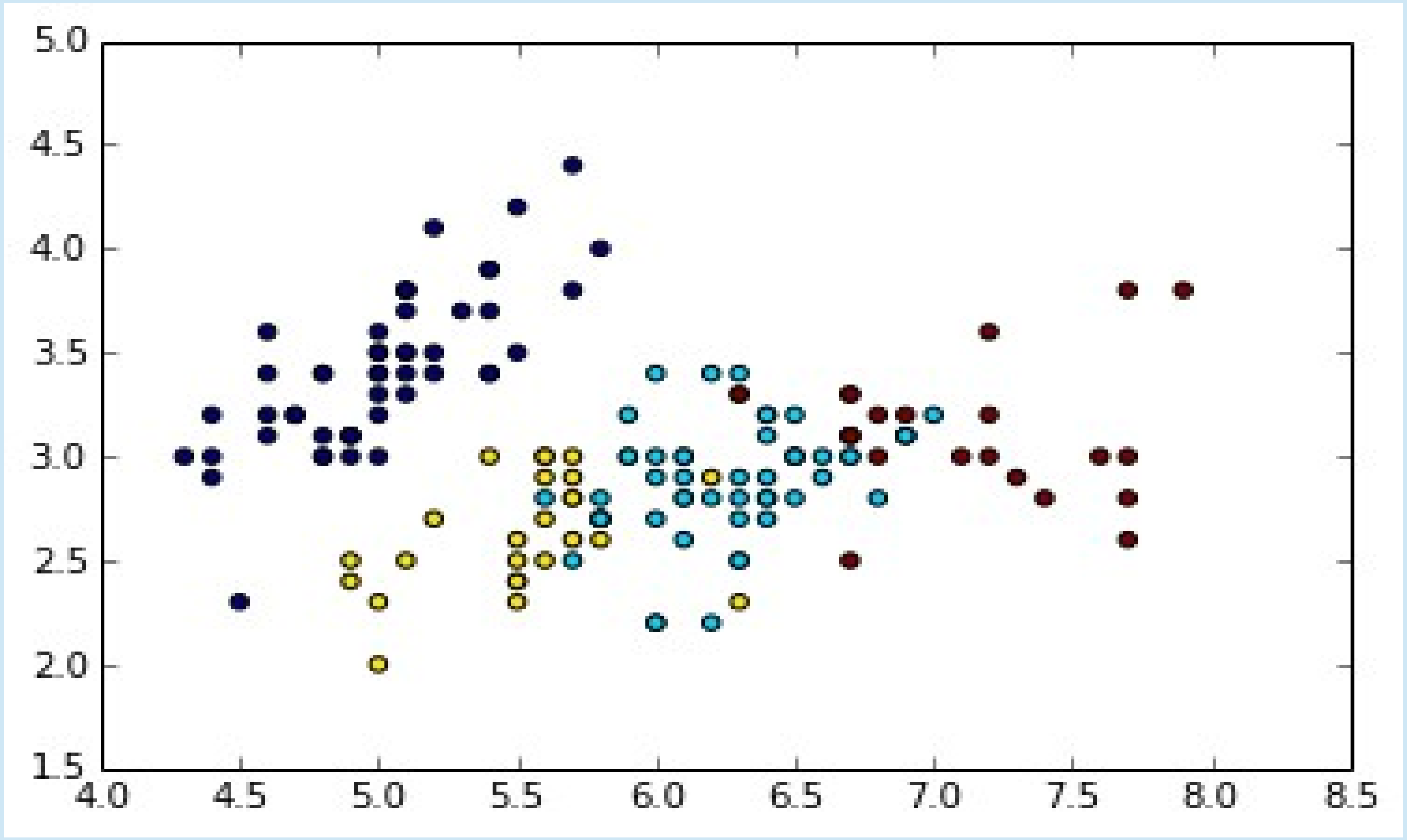
Spectral clustering

- Example

```
>>> import matplotlib.pyplot as plt
>>> from sklearn import datasets
>>> from sklearn.cluster import SpectralClustering
>>> iris = datasets.load_iris()
>>> X = iris.data[:, :2]
>>> sc = SpectralClustering(n_clusters = 4)
>>> y_pred = sc.fit_predict(X)
>>> plt.scatter(X[:, 0], X[:, 1], c = y_pred)
```

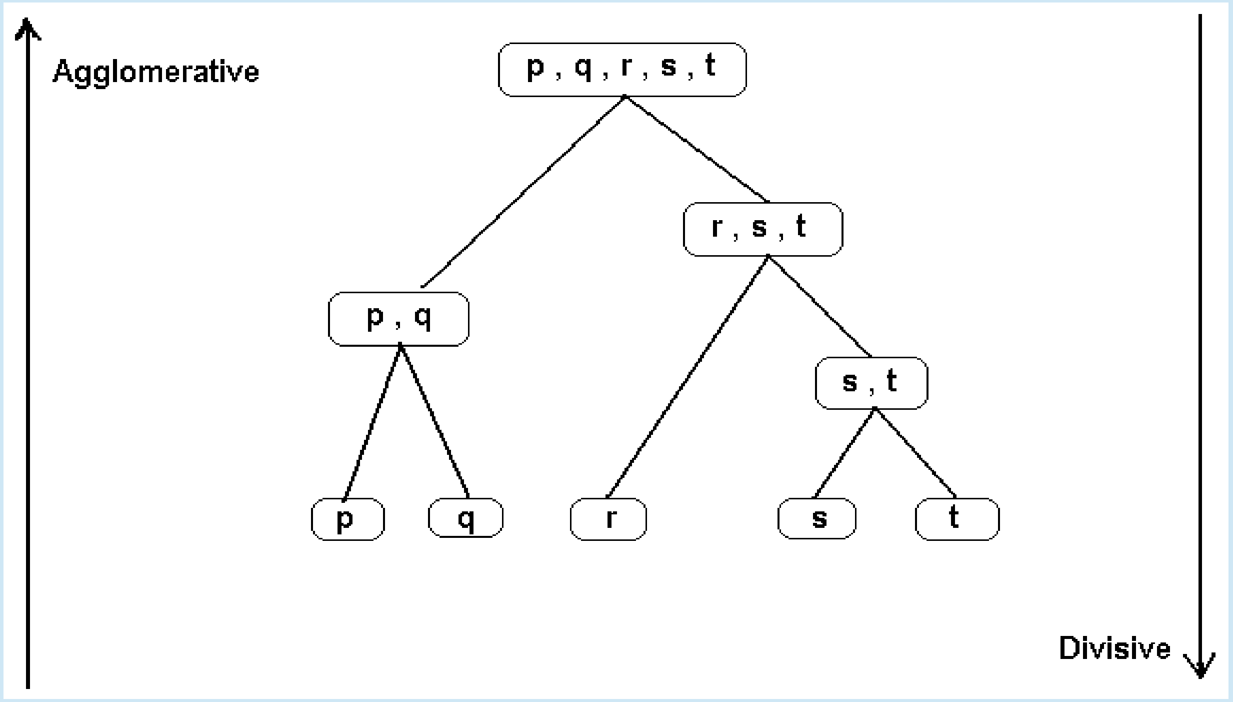
K-means clustering

- Example



Hierarchical clustering

- Clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a binary tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample.



Hierarchical clustering

- Approaches
  - Agglomerative: "bottom up" approach; each sample starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.
  - Divisive: "top down" approach; all samples start in one cluster, and splits are performed recursively as one moves down the hierarchy.

Hierarchical clustering

- **AgglomerativeClustering** follows a bottom-up approach: it recursively merges the pair of clusters that minimally increases a given linkage distance.
- The **linkage** parameter determines the metric used for the merge strategy (default value is 'ward'):
  - ward: minimizes the sum of squared differences within all clusters.
  - complete: minimizes the maximum distance between samples of pairs of clusters.
  - average: minimizes the average of the distances between all samples of pairs of clusters.

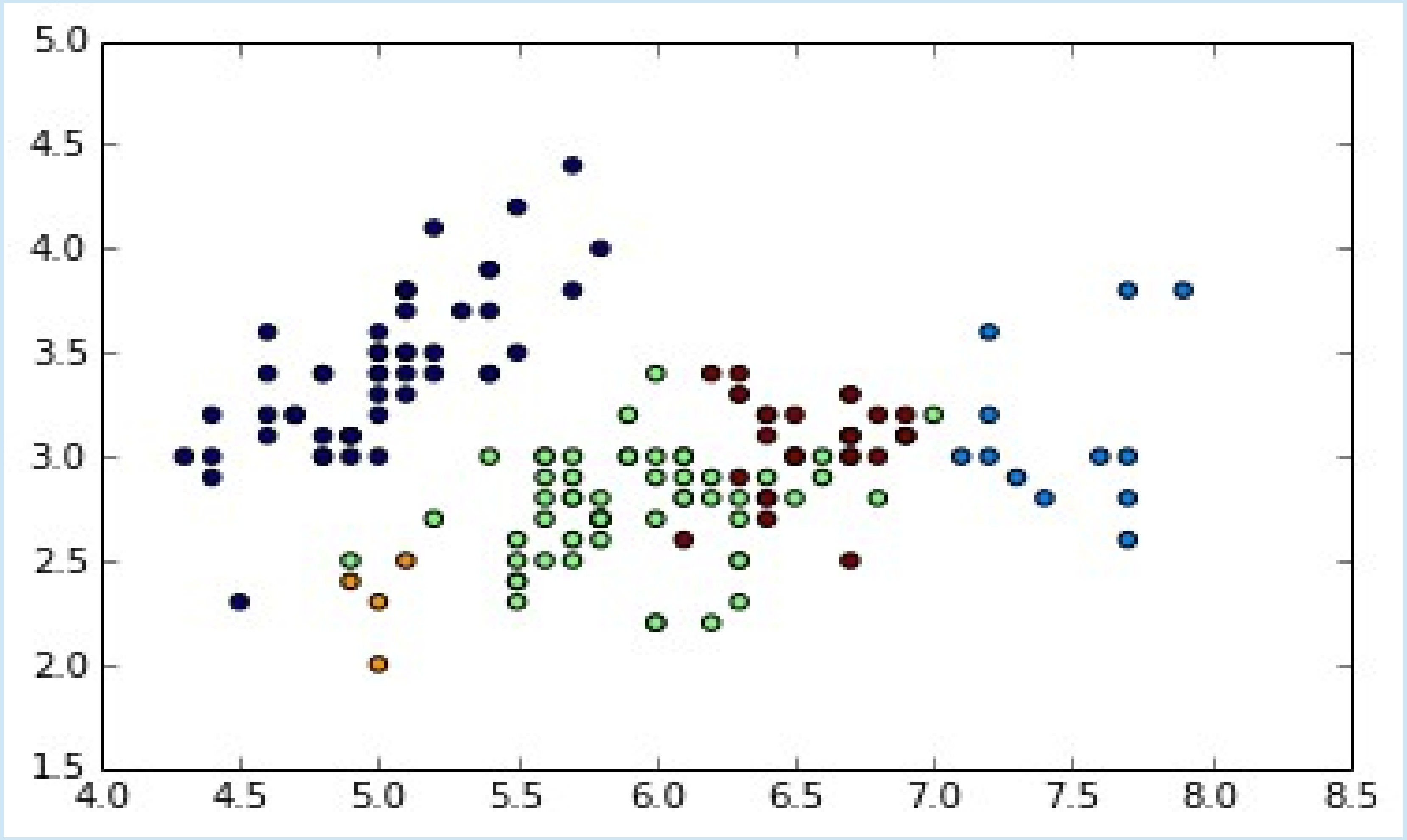
Hierarchical clustering

- Example

```
>>> import matplotlib.pyplot as plt
>>> from sklearn import datasets
>>> from sklearn.cluster import
AgglomerativeClustering as Aggregator
>>> iris = datasets.load_iris()
>>> X = iris.data[:, :2]
>>> ac = Aggregator(n_clusters = 5)
>>> y_pred = ac.fit_predict(X)
>>> plt.scatter(X[:, 0], X[:, 1], c = y_pred,
linkage='average');
```

Hierarchical clustering

- Example



The end