UniLaSalle
Terre & Sciences

Machine Learning Methods
for Agricultural & Food Data Management

— Part 1 —

— Introduction to Machine Learning —

Paul HONEINE

LITIS Lab
paul.honeine@univ-rouen.fr

— Spring 2021 —
version 2.01

Prologue

Machine Learning Methods: Outline

Outline:

Course Material

Evaluation

Outline

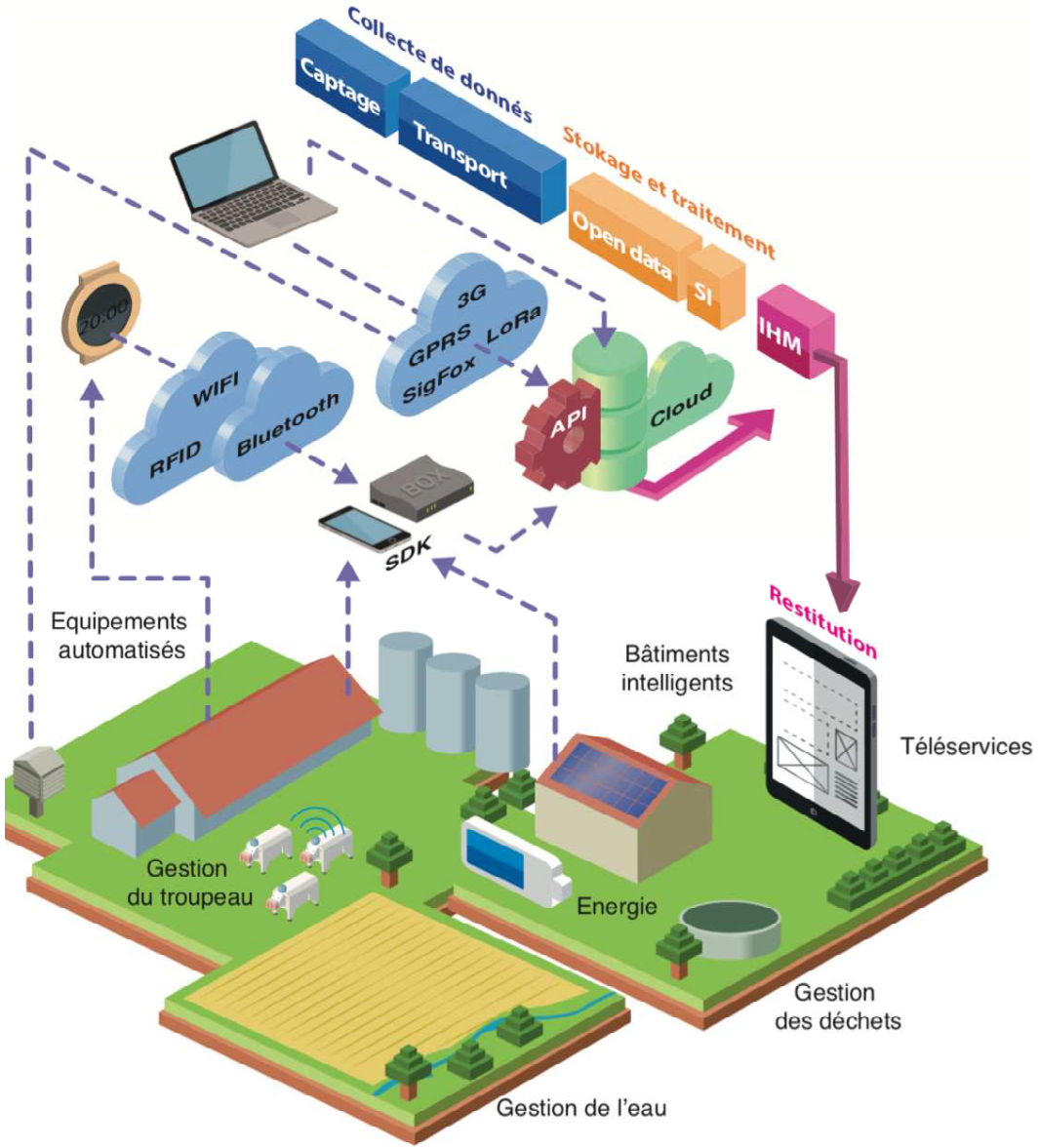
Prologue

Motivations

Farms are becoming a source of data (digital steering)

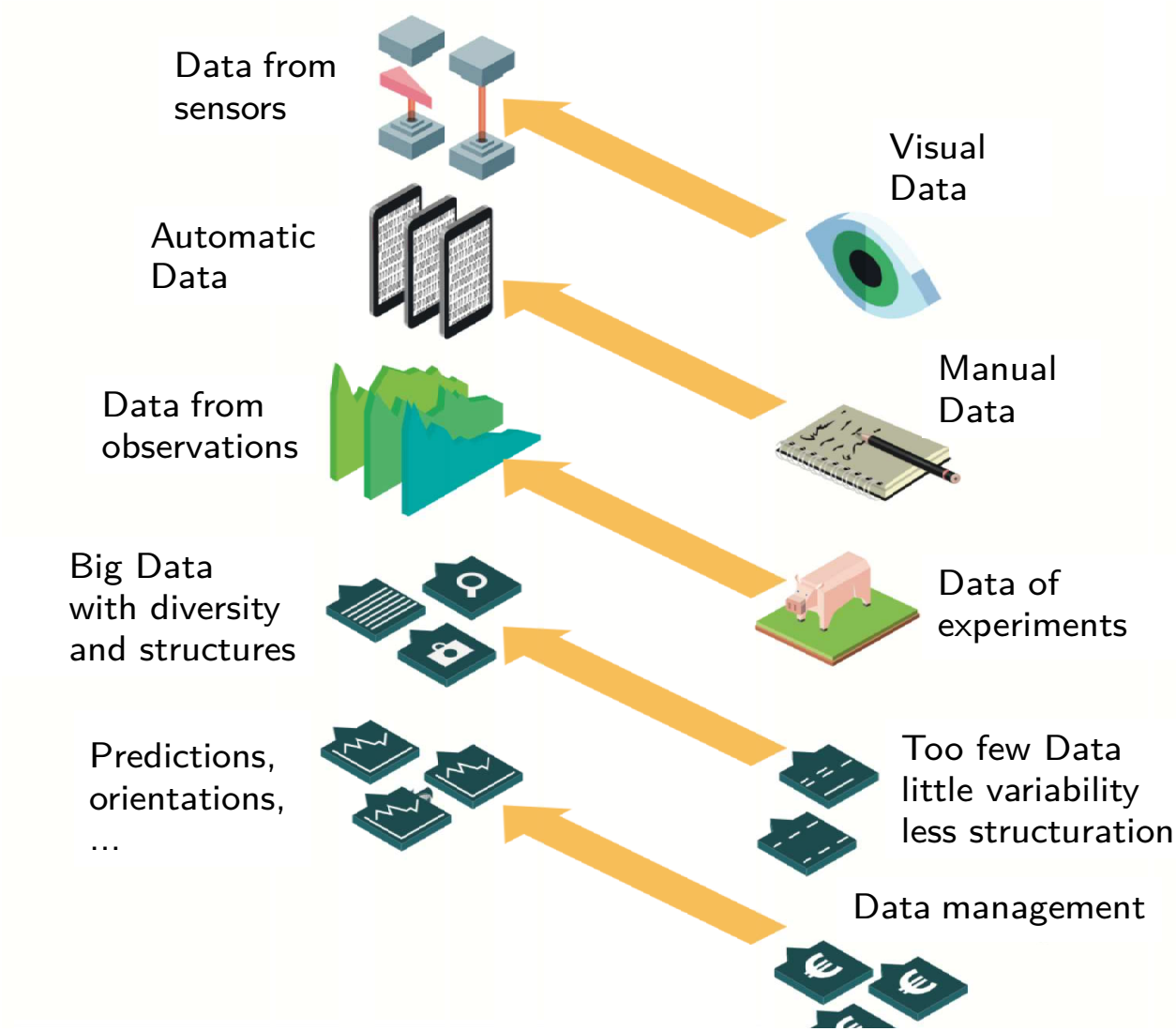
Motivations

Internet of Things (IoT) in agriculture



Motivations

Evolution of types and sources of data



This is where Machine Learning is necessary !

Methods for valuing data

From signals to data:

- With the proliferation of sensors, more and more data are collected automatically (*e.g.* temperature, rainfall in the field, ...). Most sensors require little or no human intervention. They can be embedded, for example in drones or more conventional equipment (*e.g.* tractors, harvesters, ...), or installed in livestock buildings.
- In most cases, sensor data requires processing to transform the signal into an interpretable value.
- Examples: precision agriculture with **wireless sensor networks** or with **remote sensing** using drone imaging or satellite imaging (in multi-spectral and hyperspectral cameras).
- Sensors produce raw measurements that need to be controlled and stored, then corrected to become biophysical variables (reflectance factor converted to vegetation index), and **finally applied decision models** combined with interpretation of agronomic data from some plots of lands (phenology, cultural management, ...) to lead to recommendations.

Methods for valuing data

From data to statistics:

- In order to promote Agricultural Big Data, data analysis skills become central and methodologies must evolve.
- With data acquisition directly in farms, the level of control is reduced, since this new data is captured directly by farmers or their advisors and often directly recorded by an automatic device. With this large-scale collection, the expert who collects or verifies the data is no longer present in this beginning of the chain. Checks are no longer possible on raw individual data. It is therefore necessary to set up algorithms to verify the data and their coherence and manage to manage the heterogeneities on the data and the missing data.
- In addition, the volumes of data may make the commonly used statistical methods (confidence intervals or comparison tests) irrelevant, but open up opportunities for the mobilization of new methods. There is a lot of expectation with regard to these new methods that would make it possible to propose specific predictions according to an important level of detail and thus to value the variability of the situations. But with these approaches, other problems must be considered (reproducibility of results, loss of readability, high risk of confusion between correlation and causality, sound effects, etc.).
- But in order to ensure a quality prediction, it is necessary to seek to benefit jointly from complementary sources of data, with highly contextualized agricultural data on the one hand and validated and expertized references from experimentation or observation networks on the other hand. For this, Bayesian methods could be used to integrate these different sources of knowledge.

Outline

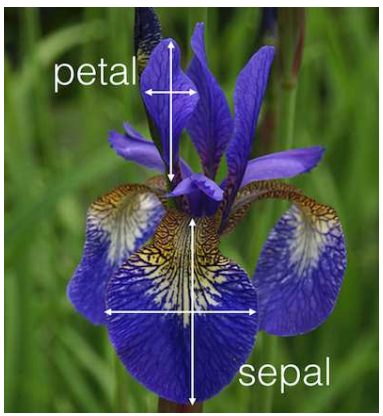
Introduction to Machine Learning

Introduction to Machine Learning

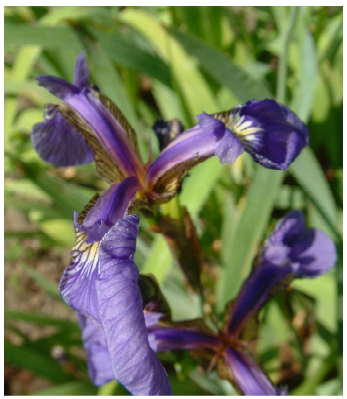
Example: (Fisher's) Iris flower dataset

- The dataset is a multivariate dataset introduced in [1], of 50 samples from each of 3 species of Iris (Iris setosa, Iris versicolor and Iris virginica). 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 4 features, Fisher developed a linear discriminant model to distinguish the species from each other.
- More details on the dataset [here]

Features



Iris setosa



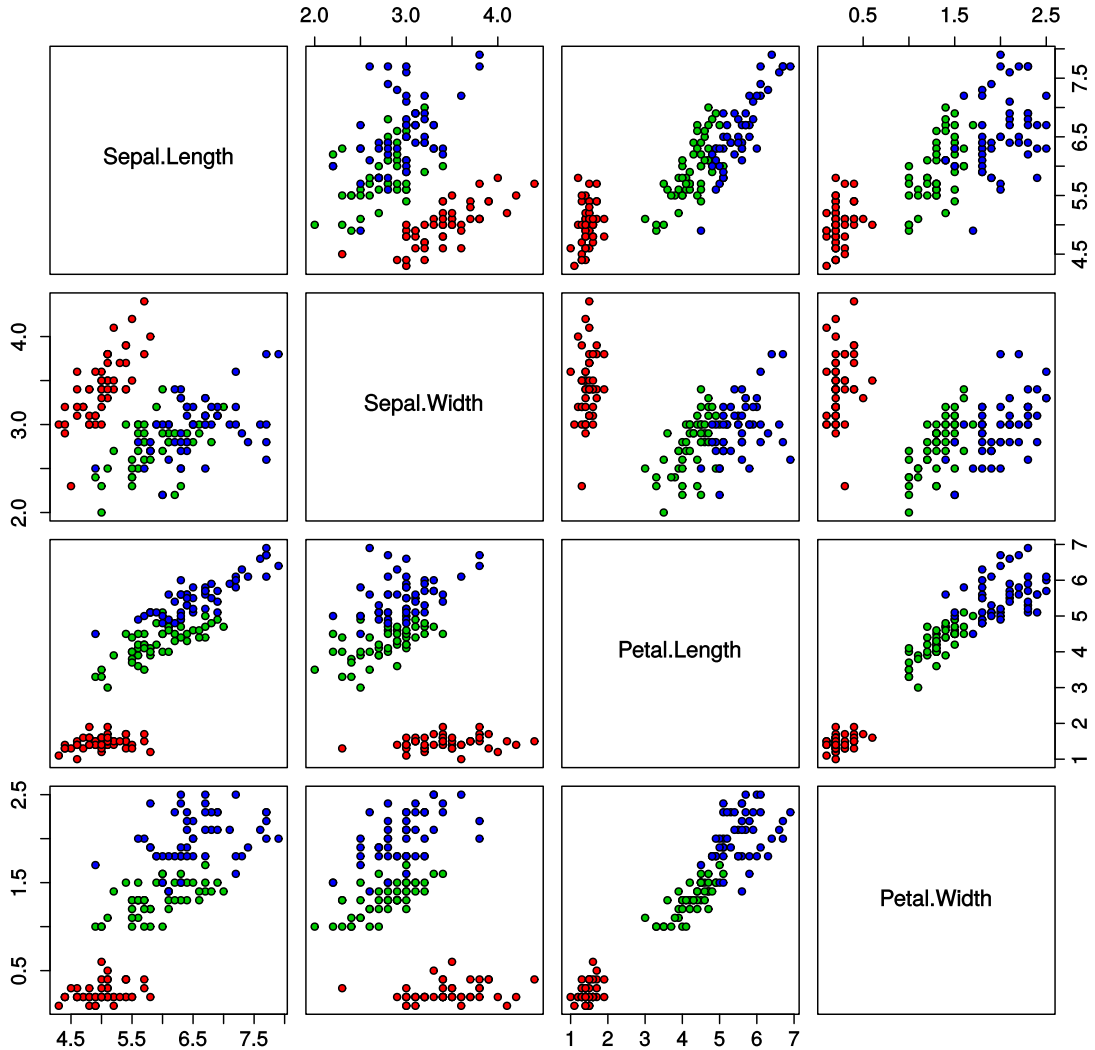
Iris versicolor

Iris virginica

[1] R. A. Fisher, "The use of multiple measurements in taxonomic problems". Annals of Eugenics. 7(2): 179–188. 1936. doi:10.1111/j.1469-1809.1936.tb02137.x

Introduction to Machine Learning

Example: (Fisher's) Iris flower dataset: the scatter plot (setosa, versicolor, virginica)



Towards Machine Learning...

Introduction to Machine Learning

Machine Learning:
ML: Collect data $\rightarrow$ choose features $\rightarrow$ choose model $\rightarrow$ train machine $\rightarrow$ evaluate

ML categories:

- Supervised learning
- Unsupervised learning

Models:

- Linear model
- Nonlinear models (Radial basis functions, nonlinear kernels, ...)

ML algorithms:

- **Supervised learning**
  - **Classification** (binary or multiclass): naive Bayes, linear discriminant analysis, Support Vector Machines, decision trees, k-nearest neighbor algorithm, neural networks (multilayer perceptron)
  - **Regression** (on $\mathbb{R}$ or multidimensional): Linear regression, logistic regression, neural networks (Multilayer perceptron), Support Vector Machines for regression
- **Unsupervised learning**
  - **Clustering** with k-means, hierarchical, ...
  - Anomaly detection (*e.g.* one-class detection)
  - Neural networks with Autoencoders, Hebbian learning, self-organizing map, ...
  - Blind signal separation with principal component analysis, independent component analysis, non-negative matrix factorization, ...

Outline

Machine Learning in Python

Scikit-learn: Machine Learning in Python

Scikit-learn: Machine Learning in Python

https://scikit-learn.org[2]

- 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

Installing, probably the easiest way (on any OS):

- Install Anaconda from https://www.anaconda.com/
- Use Jupyter (a web-based notebook environment) or Spider (an IDE with advanced editing and debugging)

Installing using pip or conda (*e.g.* on Linux or MacOS):

- Install `pip`: `sudo easy_install pip` or `sudo python -m ensurepip`
- Install it: `pip install -U scikit-learn` or `conda install scikit-learn`

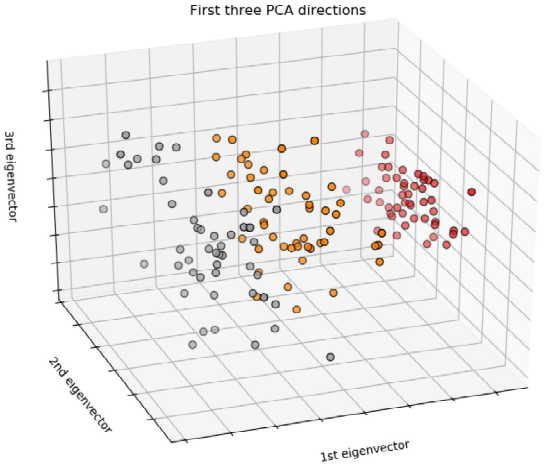
[2] This section is based on this website.

A primal example

Plot iris dataset

```
# Code source: Gael Varoquaux; Modified for documentation by Jaques Grobler; License: BSD 3 clause

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.decomposition import PCA

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features.
y = iris.target

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

plt.figure(2, figsize=(8, 6))
plt.clf()

# Plot the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1, edgecolor='k')
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())

# To getter a better understanding of interaction of the dimensions
# plot the first three PCA dimensions
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
X_reduced = PCA(n_components=3).fit_transform(iris.data)
ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2],
           c=y, cmap=plt.cm.Set1, edgecolor='k', s=40)
ax.set_title("First three PCA directions")
ax.set_xlabel("1st eigenvector")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("2nd eigenvector")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("3rd eigenvector")
ax.w_zaxis.set_ticklabels([])

plt.show()
```

```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> data.target[[10, 25, 50]]
array([0, 0, 1])
>>> list(data.target_names)
['setosa', 'versicolor', 'virginica']
```



An introduction to machine learning with scikit-learn

Machine learning: the problem setting

A ML problem considers a set of n samples of data and then tries to predict properties of unknown data. If each sample is a multi-dimensional entry (*i.e.,* multivariate data), it is said to have several attributes or **features**.

ML problems fall into the following categories:

- **Supervised learning**, in which the data comes with additional attributes that we want to predict.This problem can be either:
  - **Classification**: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories.
  - **Regression**: if the desired output consists of one or more continuous variables, then the task is called regression.
- **Unsupervised learning**, in which the training data consists of a set of input vectors without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called **clustering**, or to determine the distribution of data within the input space, known as **density estimation**, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of **visualization**.

Training set and testing set

ML is about learning some properties of a dataset, and then testing those properties against another dataset. A common practice in machine learning is to evaluate an algorithm by splitting a data set into two. We call one of those sets the training set, on which we learn some properties; we call the other set the testing set, on which we test the learned properties.

Loading an example dataset

scikit-learn comes with a few standard datasets, including the iris and digits datasets.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> digits = datasets.load_digits()
```

A dataset is a dictionary-like object that holds all the data and some metadata about the data. This data is stored in the .data member, which is a `n_samples, n_features` array. In the case of supervised problem, one or more response variables are stored in the `.target` member. More details on the different datasets can be found in the dedicated section.
For instance, in the case of the digits dataset, `digits.data` gives access to the features that can be used to classify the digits samples:

```
>>> print(digits.data)
[[ 0.   0.   5. ...   0.   0.   0.]
 [ 0.   0.   0. ...  10.   0.   0.]
 [ 0.   0.   0. ...  16.   9.   0.]
 ...
 [ 0.   0.   1. ...   6.   0.   0.]
 [ 0.   0.   2. ...  12.   0.   0.]
 [ 0.   0.  10. ...  12.   1.   0.]]
```

and `digits.target` gives the ground truth for the digit dataset, that is the number corresponding to each digit image that we are trying to learn:

```
>>> digits.target
array([0, 1, 2, ..., 8, 9, 8])
```

Shape of the data arrays:
The data is always a 2D array, shape `(n_samples, n_features)`, although the original data may have had a different shape. In the case of the digits, each original sample is an image of shape `(8, 8)` and can be accessed using:

```
>>> digits.images[0]
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

Comment:
The loader function returns a dictionary-like object holding at least two items: an array of shape `n_samples` × `n_features` with key data and a numpy array of length `n_samples`, containing the target values, with key target. The datasets also contain a full description in their `DESCR` attribute and some contain `feature_names` and `target_names`.

Learning and predicting digits (1/2)

The task is to predict which digit a given image represents. We are given samples of each of the 10 competing classes ("0" to "9") on which we fit an estimator to be able to predict the class of a given sample.

In scikit-learn, an estimator for classification is a Python object that implements the methods `fit(X, y)` and `predict(T)`.

An example of an estimator is the class `sklearn.svm.SVC`, which implements support vector classification. The estimator's constructor takes as arguments the model's parameters.

We consider next the estimator as a black box:

```
>>> from sklearn import svm
>>> clf = svm.SVC(gamma=0.001, C=100.)
```

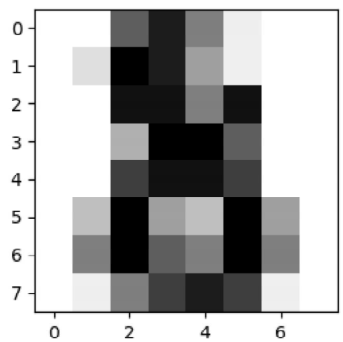
The `clf` (for classifier) estimator instance is first fitted to the model; that is, it must learn from the model. This is done by passing our training set to the `fit` method. For the training set, we'll use all the images from our dataset, except for the last image, which we'll reserve for our predicting. We select the training set with the `[:-1]` Python syntax, which produces a new array that contains all but the last item from `digits.data`:

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, cache_size=200, class_weight=None,
  coef0=0.0, decision_function_shape='ovr',
  degree=3, gamma=0.001, kernel='rbf', max_iter=-1,
  probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

Now you can predict new values. In this case, you'll predict using the last image from `digits.data`. By predicting, you'll determine the image from the training set that best matches the last image.

```
>>> clf.predict(digits.data[-1:])
array([8])
```

The corresponding image is:



As it can be seen, it is a challenging task: after all, the images are of poor resolution. Do you agree with the classifier ?

Comment on choosing the model parameters:
In this example, we have set the value of `gamma` manually. To find good values for these parameters, we can use tools such as grid search and cross validation.

Learning and predicting digits: some details

Model persistence

It is possible to save a model in scikit-learn by using Python's built-in persistence model, `pickle`:

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC(gamma='scale')
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr',
  degree=3, gamma='scale', kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)

>>> import pickle
>>> s = pickle.dumps(clf)
>>> clf2 = pickle.loads(s)
>>> clf2.predict(X[0:1])
array([0])
>>> y[0]
0
```

In the specific case of scikit-learn, it may be more interesting to use joblib's replacement for pickle (`joblib.dump` & `joblib.load`), which is more efficient on big data but it can only pickle to the disk and not to a string:

```
>>> from joblib import dump, load
>>> dump(clf, 'filename.joblib')
```

Later, you can reload the pickled model (possibly in another Python process) with:

```
>>> clf = load('filename.joblib')
```

Learning and predicting digits: some details

Conventions on Type Casting

Unless otherwise specified, input will be cast to `float64`.

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

>>> rng = np.random.RandomState(0)
>>> X = rng.rand(10, 2000)
>>> X = np.array(X, dtype='float32')
>>> X.dtype
dtype('float32')

>>> transformer = random_projection.GaussianRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.dtype
dtype('float64')
```

In this example, `X` is `float32`, which is cast to `float64` by `fit_transform(X)`.

```
>>> from sklearn import datasets
>>> from sklearn.svm import SVC
>>> iris = datasets.load_iris()
>>> clf = SVC(gamma='scale')
>>> clf.fit(iris.data, iris.target)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3,
  gamma='scale', kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

>>> list(clf.predict(iris.data[:3]))
[0, 0, 0]

>>> clf.fit(iris.data, iris.target_names[iris.target])
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3,
  gamma='scale', kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

>>> list(clf.predict(iris.data[:3]))
['setosa', 'setosa', 'setosa']
```

Here, the first `predict()` returns an integer array, since `iris.target` (an integer array) was used in `fit`. The second `predict()` returns a string array, since `iris.target_names` was for fitting.

Learning and predicting digits: some details

Refitting and updating parameters

Hyper-parameters of an estimator can be updated after it has been constructed via the `set_params()` method. Calling `fit()` more than once will overwrite what was learned by any previous `fit()`:

```
>>> import numpy as np
>>> from sklearn.svm import SVC

>>> rng = np.random.RandomState(0)
>>> X = rng.rand(100, 10)
>>> y = rng.binomial(1, 0.5, 100)
>>> X_test = rng.rand(5, 10)

>>> clf = SVC()
>>> clf.set_params(kernel='linear').fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3,
  gamma='auto_deprecated', kernel='linear', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
>>> clf.predict(X_test)
array([1, 0, 1, 1, 0])

>>> clf.set_params(kernel='rbf', gamma='scale').fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3,
  gamma='scale', kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
>>> clf.predict(X_test)
array([1, 0, 1, 1, 0])
```

Here, the default kernel `rbf` is first changed to `linear` via `SVC.set_params()` after the estimator has been constructed, and changed back to `rbf` to refit the estimator and to make a second prediction.

Learning and predicting digits: Multiclass vs. multilabel

When using `multiclass classifiers`, the learning and prediction task that is performed is dependent on the format of the target data fit upon:

```
>>> from sklearn.svm import SVC
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.preprocessing import LabelBinarizer

>>> X = [[1, 2], [2, 4], [4, 5], [3, 2], [3, 1]]
>>> y = [0, 0, 1, 1, 2]

>>> classif = OneVsRestClassifier(estimator=SVC(gamma='scale', random_state=0))
>>> classif.fit(X, y).predict(X)
array([0, 0, 1, 1, 2])
```

In above, the classifier is fit on a 1d array of multiclass labels and the `predict()` method therefore provides corresponding multiclass predictions. It is also possible to fit upon a 2d array of binary label indicators:

```
>>> y = LabelBinarizer().fit_transform(y)
>>> classif.fit(X, y).predict(X)
array([[1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

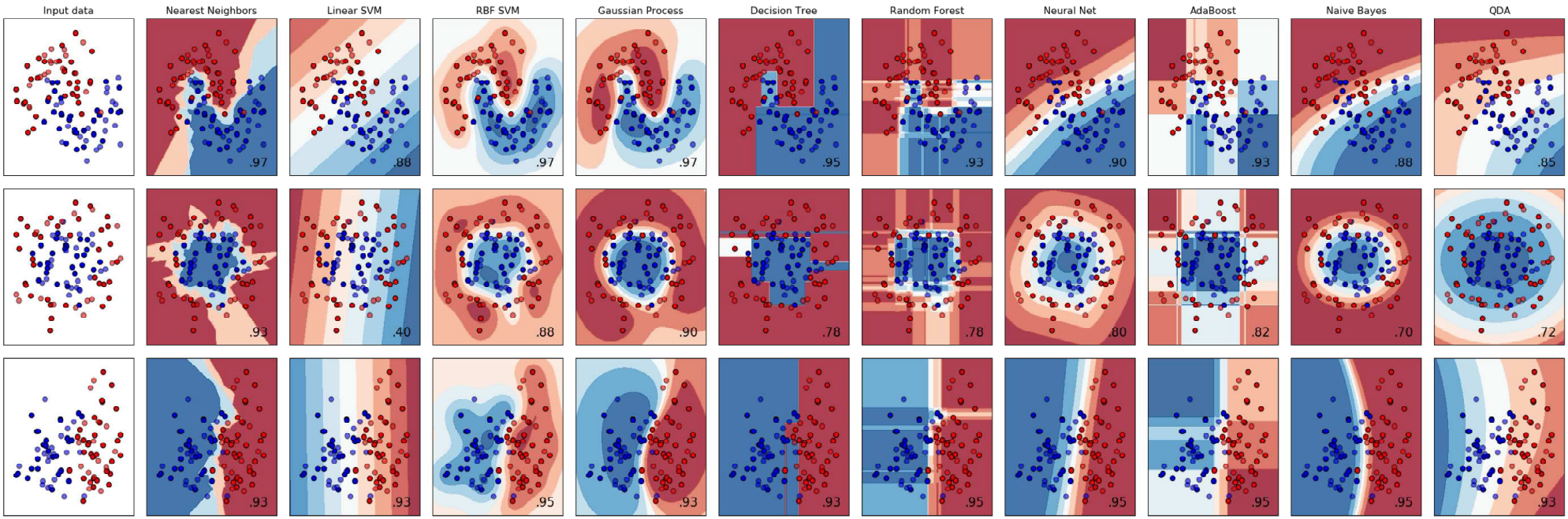
Here, the classifier is `fit()` on a 2d binary label representation of `y`, using the `LabelBinarizer`. In this case `predict()` returns a 2d array representing the corresponding multilabel predictions.

Note that the 4-th and 5-th instances returned all zeroes, indicating that they matched none of the three labels fit upon. With multilabel outputs, it is also possible for an instance to be assigned multiple labels:

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> y = [[0, 1], [0, 2], [1, 3], [0, 2, 3], [2, 4]]
>>> y = MultiLabelBinarizer().fit_transform(y)
>>> classif.fit(X, y).predict(X)
array([[1, 1, 0, 0, 0],
       [1, 0, 1, 0, 0],
       [0, 1, 0, 1, 0],
       [1, 0, 1, 0, 0],
       [1, 0, 1, 0, 0]])
```

In this case, the classifier is fit upon instances each assigned multiple labels. The `MultiLabelBinarizer` is used to binarize the 2d array of multilabels to `fit` upon. As a result, `predict()` returns a 2d array with multiple predicted labels for each instance.

scikit-learn: Classifier Comparison



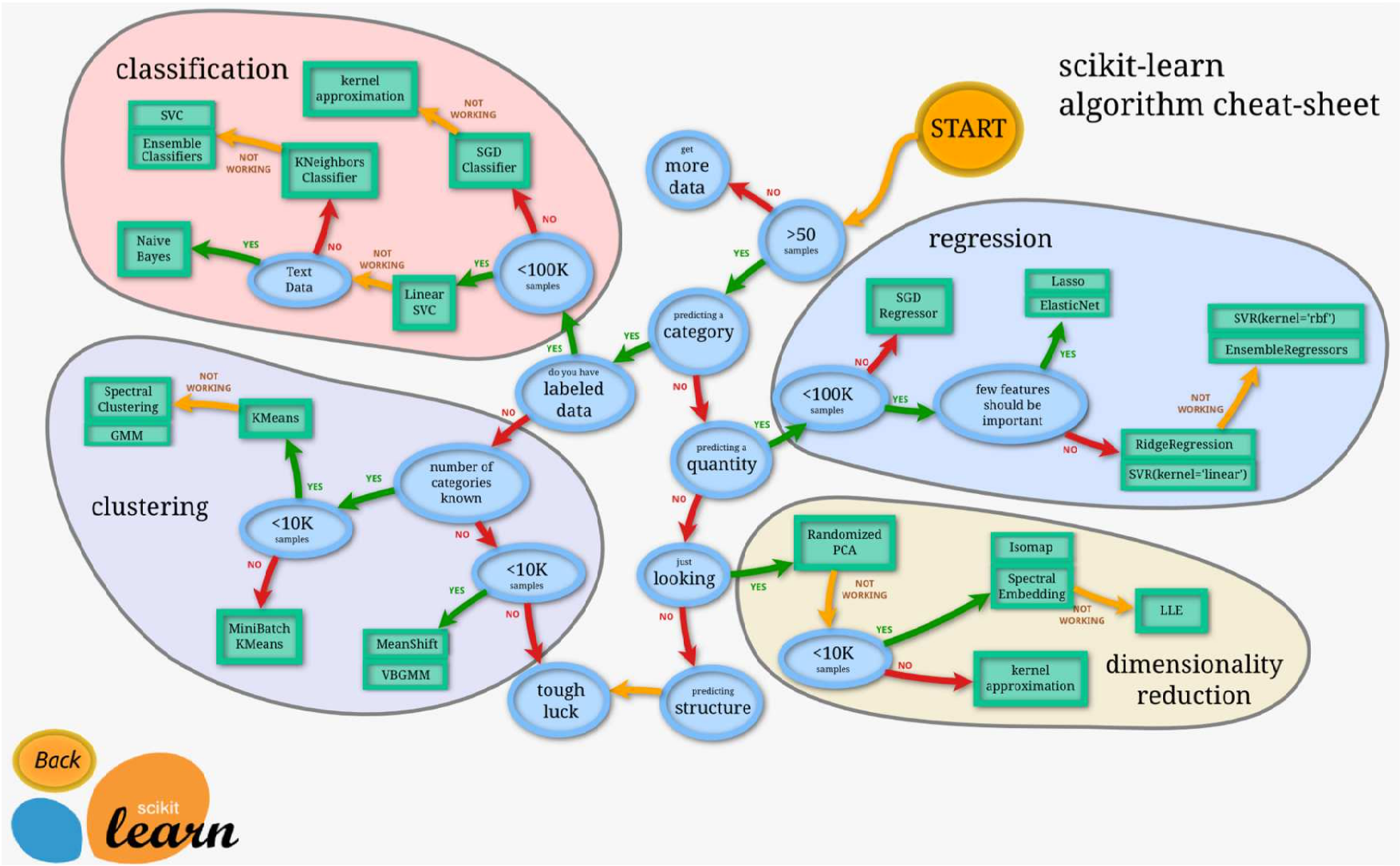
scikit-learn: Classifier Comparison

Plot classifier comparison

```
# Code source: Gael Varoquaux & Andreas Muller
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

h = .02  # step size in the mesh

names = ["Nearest Neighbors", "Linear SVM", "RBF SVM", "Gaussian Process",
         "Decision Tree", "Random Forest", "Neural Net", "AdaBoost",
         "Naive Bayes", "QDA"]

classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(1.0 * RBF(1.0)),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    MLPClassifier(alpha=1),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis()]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1), linearly_separable]

figure = plt.figure(figsize=(27, 9))
i = 1
# iterate over datasets
for ds_cnt, ds in enumerate(datasets):
    # preprocess dataset, split into training and test part
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=.4, random_state=42)

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
```

```
    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    if ds_cnt == 0:
        ax.set_title("Input data")
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               edgecolors='k')
    # Plot the testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
               alpha=0.6,  edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color
        # to each point in the mesh [x_min, x_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

        # Plot the training points
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
                   cmap=cm_bright, edgecolors='k')
        # Plot the testing points
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test,
                   cmap=cm_bright, edgecolors='k', alpha=0.6)

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        if ds_cnt == 0:
            ax.set_title(name)
        ax.text(xx.max()-.3, yy.min()+.3, ('%.2f' % score).lstrip('0'),
                size=15, horizontalalignment='right')
        i += 1

plt.tight_layout()
plt.show()
```

scikit-learn: A Roadmap

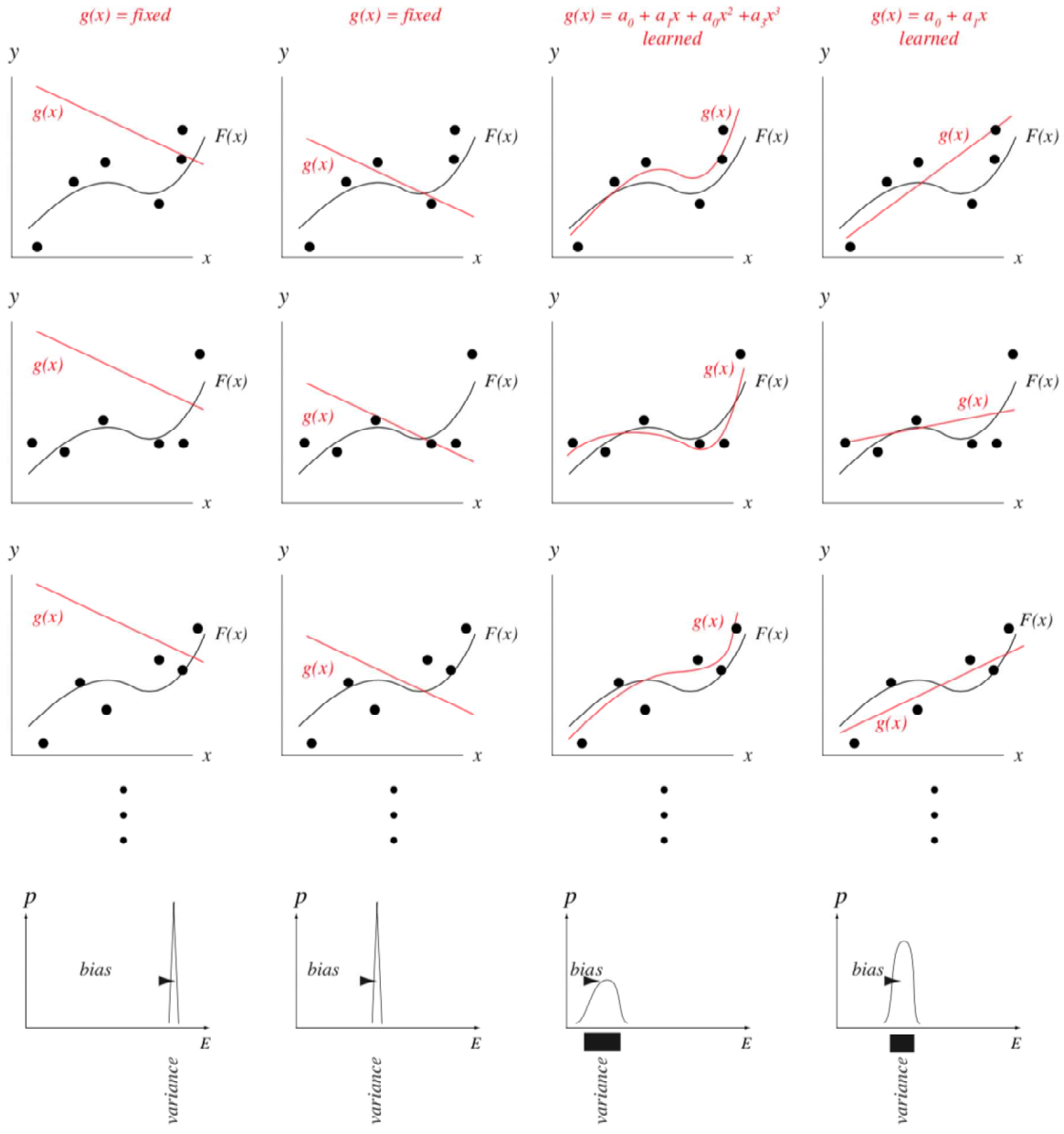


https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

Outline

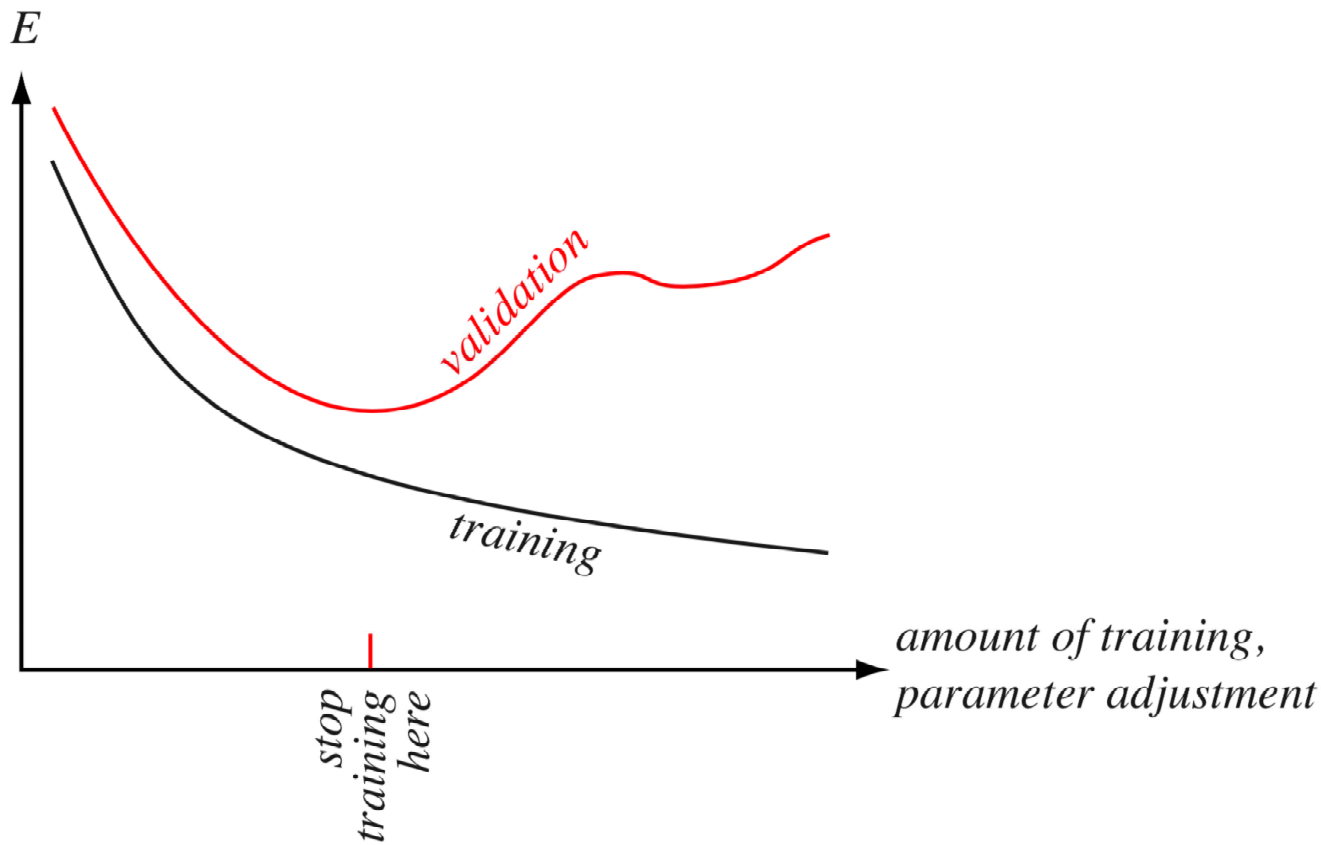
Which Model-Machine-Parameters to Choose ?

Example of Bias-Variance Tradeoff: Regression



Cross-validation

Cross-validation: k-fold and leave-one-out



PS: Cross-validation is a heuristic

Cross-validation

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called overfitting.

To avoid it, it is common practice when performing supervised machine learning to hold out part of the available data as a test set `X_test, y_test`.

In scikit-learn, a random split into training and test sets can be quickly computed with the `train_test_split` helper function.

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> from sklearn import datasets
>>> from sklearn import svm

>>> iris = datasets.load_iris()
>>> iris.data.shape, iris.target.shape
((150, 4), (150,))

# quickly sample a training set while holding out 40%
# of the data for testing (evaluating) the classifier:
>>> X_train, X_test, y_train, y_test = train_test_split(
        iris.data, iris.target, test_size=0.4, random_state=0)
>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))

>>> clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.96...
```

Cross-validation

When evaluating **different settings ("hyperparameters")** for estimators, such as the C setting that must be manually set for an SVM, there is still a **risk of overfitting** on the test set because the parameters can be tweaked until the estimator performs optimally. This way, **knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance**.

To solve this problem, yet another part of the dataset can be held out as a so-called **"validation set"**: training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called **cross-validation**: A test set should still be held out for final evaluation, but the validation set is no longer needed. In the basic approach, called k-fold cross-validation, the training set is split into k subsets (aka "folds"). The following procedure is followed for each of the k folds:

- A model is trained using $k - 1$ of the folds as training data;
- the resulting model is validated on the remaining fold of the data (*i.e.*, it is used as a test set to compute a performance measure such as accuracy).

The **performance measure** reported by k-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but **does not waste too much data** (as is the case when fixing an arbitrary validation set), which is a major advantage.

Cross-validation

The simplest way to use cross-validation is to call the `cross_val_score` helper function on the estimator and the dataset.

The following example demonstrates how to estimate the accuracy of a linear kernel support vector machine on the iris dataset by splitting the data, fitting a model and computing the score 5 consecutive times (with different splits each time):

```
>>> from sklearn.model_selection import cross_val_score
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = cross_val_score(clf, iris.data, iris.target, cv=5)
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

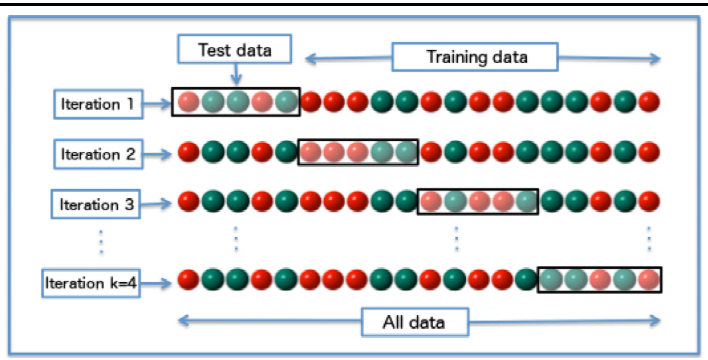
The mean score and the 95% confidence interval of the score estimate are given by:

```
>>> print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
Accuracy: 0.98 (+/- 0.03)
```

Cross-validation and Leave-one-out

k-fold cross-validation: `KFold` divides all the samples in k groups of samples, called folds, of equal sizes (if possible). The prediction function is learned using $k-1$ folds, and the fold left out is used for test. Therefore, the k models are each learned on $\approx (k-1)n/k$ samples.

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold

>>> X = ["a", "b", "c", "d"]
>>> kf = KFold(n_splits=2)
>>> for train, test in kf.split(X):
      print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```



Leave-one-out: `LeaveOneOut` is a simple cross-validation (obtained from $k = n$).

```
>>> from sklearn.model_selection import LeaveOneOut

>>> X = [1, 2, 3, 4]
>>> loo = LeaveOneOut()
>>> for train, test in loo.split(X):
        print("%s %s" % (train, test))
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

- It is computationally expensive, with n models each trained on $n-1$ samples.
- In terms of accuracy, leave-one-out often results in high variance as an estimator for the test error.
- Intuitively, since $n-1$ of the n samples are used to build each model, models constructed from folds are virtually identical to each other and to the model built from the entire training set

General rule: most authors and empirical evidence, suggest that 5- or 10- fold cross validation should be preferred to leave-one-out.

Outline

Data

Available Data

Data from Machine Learning community:

- OpenML: https://www.openml.org/
- Kaggle: https://www.kaggle.com/datasets
- UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/index.php

Data from Agriculture and Food Data Management communities:

- Alim'confiance (dataset, export or API tab): https://dgal.opendatasoft.com
- Surfaces, rendements et productivités des productions végétales (dataset, export or API tab): http://www.api-agro.fr
- Global Open Data for Agriculture and Nutrition : https://www.godan.info/
- https://world.openfoodfacts.org/data (Big Data, watch out !)

Scikit-learn Loading External Data

scikit-learn works on any numeric data stored as numpy arrays or scipy sparse matrices. Other types that are convertible to numeric arrays such as pandas DataFrame are also acceptable.

- pandas.io provides tools to read data from common formats including CSV, Excel, JSON and SQL. DataFrames may also be constructed from lists of tuples or dicts. Pandas handles heterogeneous data smoothly and provides tools for manipulation and conversion into a numeric array suitable for scikit-learn.
- scipy.io specializes in binary formats often used in scientific computing context such as .mat and .arff
- numpy/routines.io for standard loading of columnar data into numpy arrays
- scikit-learn's `datasets.load_svmlight_file` for svmlight or libSVM sparse format
- scikit-learn's `datasets.load_files` for directories of text files where the name of each directory is the name of each category and each file inside of each directory corresponds to one sample from that category

For some miscellaneous data such as images, videos, and audio:

- skimage.io or Imageio for loading images and videos into numpy arrays
- scipy.io.wavfile.read for reading WAV files into a numpy array

Categorical (or nominal) features stored as strings (common in pandas DataFrames) will need converting to numerical features using `sklearn.preprocessing.OneHotEncoder` or `sklearn.preprocessing.OrdinalEncoder` or similar. See next section or Preprocessing data.

Scikit-learn Downloading Datasets from OpenML.org

The sklearn.datasets package is able to download datasets from the OpenML.org repository using the function `sklearn.datasets.fetch_openml`

Example: Download a dataset of gene expressions in mice brains, containing 1080 examples in 8 classes:

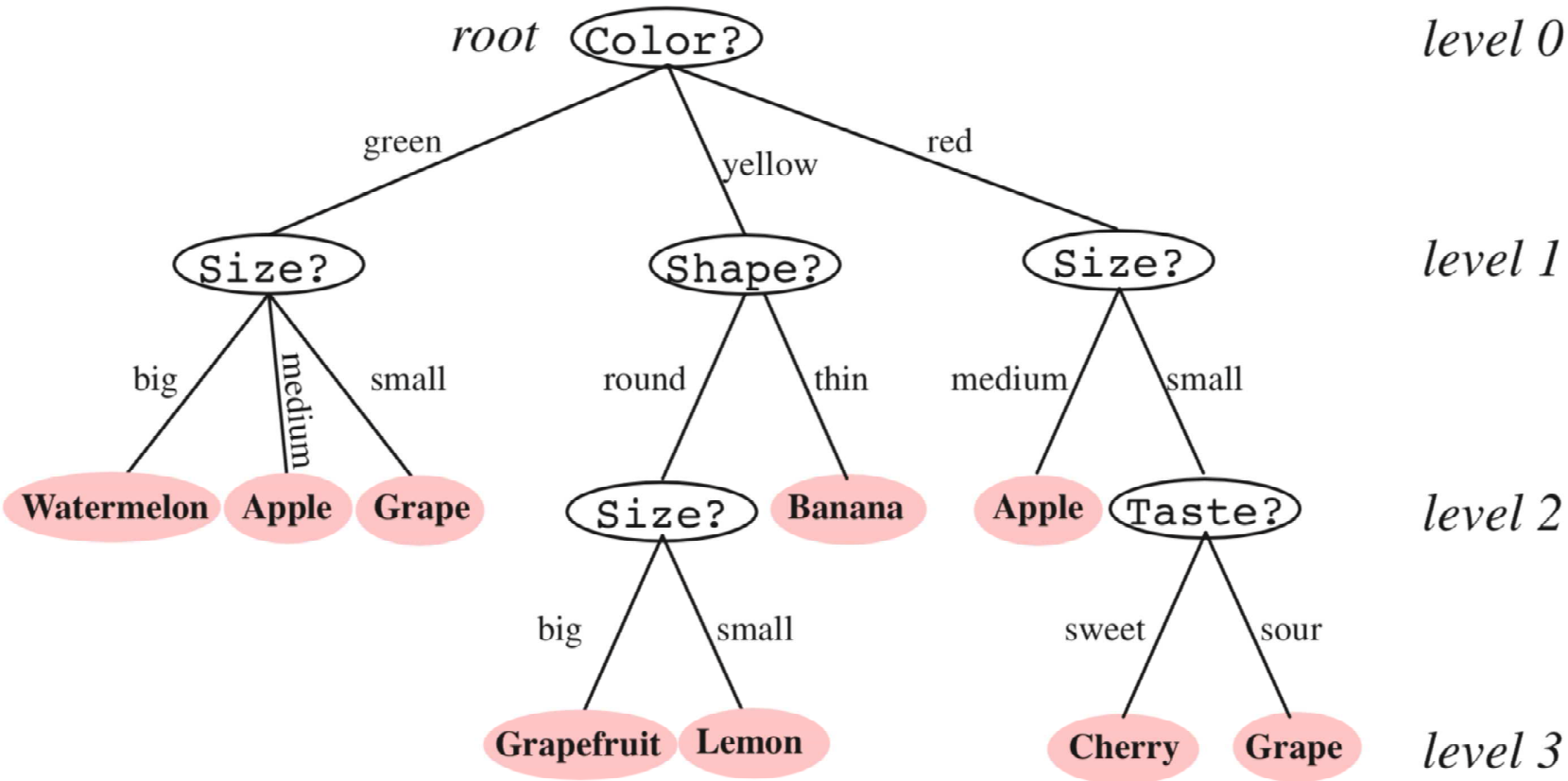
```
>>> from sklearn.datasets import fetch_openml
>>> mice = fetch_openml(name='miceprotein', version=4)
>>> mice.data.shape
(1080, 77)
>>> mice.target.shape
(1080,)
>>> np.unique(mice.target)
array(['c-CS-m', 'c-CS-s', 'c-SC-m', 'c-SC-s', 't-CS-m', 't-CS-s', 't-SC-m', 't-SC-s'], dtype=object)
>>> print(mice.DESCR)
**Author**: Clara Higuera, Katheleen J. Gardiner, Krzysztof J. Cios
**Source**: [UCI](https://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression) - 2015
**Please cite**: Higuera C, Gardiner KJ, Cios KJ (2015) Self-Organizing
Feature Maps Identify Proteins Critical to Learning in a Mouse Model of Down
Syndrome. PLoS ONE 10(6): e0129126...

>>> mice.details
{'id': '40966', 'name': 'MiceProtein', 'version': '4', 'format': 'ARFF',
'upload_date': '2017-11-08T16:00:15', 'licence': 'Public',
'url': 'https://www.openml.org/data/v1/download/17928620/MiceProtein.arff',
'file_id': '17928620', 'default_target_attribute': 'class',
'row_id_attribute': 'MouseID',
'ignore_attribute': ['Genotype', 'Treatment', 'Behavior'],
'tag': ['OpenML-CC18', 'study_135', 'study_98', 'study_99'],
'visibility': 'public', 'status': 'active',
'md5_checksum': '3c479a6885bfa0438971388283a1ce32'}
>>> mice.url
'https://www.openml.org/d/40966'
```

```
>>> mice = fetch_openml(data_id=40966)
>>> mice.details
{'id': '4550', 'name': 'MiceProtein', 'version': '1', 'format': 'ARFF',
'creator': ...,
'upload_date': '2016-02-17T14:32:49', 'licence': 'Public', 'url':
'https://www.openml.org/data/v1/download/1804243/MiceProtein.ARFF', 'file_id':
'1804243', 'default_target_attribute': 'class', 'citation': 'Higuera C,
Gardiner KJ, Cios KJ (2015) Self-Organizing Feature Maps Identify Proteins
Critical to Learning in a Mouse Model of Down Syndrome. PLoS ONE 10(6):
e0129126. [Web Link] journal.pone.0129126', 'tag': ['OpenML100', 'study_14',
'study_34'], 'visibility': 'public', 'status': 'active', 'md5_checksum':
'3c479a6885bfa0438971388283a1ce32'}
```

Outline

Categorical Features

Decision Trees as a Non-metric Method



Encoding Categorical Features in Python: `OrdinalEncoder`

Often features are not given as continuous values but categorical. For example a person could have features `["male", "female"]`, `["from Europe", "from US", "from Asia"]`, `["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]`. Such features can be efficiently coded as integers, for instance `["male", "from US", "uses Internet Explorer"]` could be expressed as `[0, 1, 3]` while `["female", "from Asia", "uses Chrome"]` would be `[1, 2, 1]`.

The `OrdinalEncoder` converts categorical features to such integer codes, by transforming each categorical feature to one new feature of integers (`0` to `n_categories - 1`):

```
>>> enc = preprocessing.OrdinalEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OrdinalEncoder(categories='auto', dtype=<... 'numpy.float64'>)
>>> enc.transform([['female', 'from US', 'uses Safari']])
array([[0., 1., 1.]])
```

Such integer representation **should not be used directly** with all scikit-learn estimators, as these expect continuous input, and would interpret the categories as being ordered, which is often not desired (*i.e.*, the set of browsers was ordered arbitrarily).

The only use of `OrdinalEncoder` is when categorical features can be ordered, such as:

- `'worst'` $\prec$ `'worse'` $\prec$ `'bad'` $\prec$ `'good'` $\prec$ `'better'` $\prec$ `'best'`
- `'F'` $\prec$ `'E'` $\prec$ `'D'` $\prec$ `'C'` $\prec$ `'B'` $\prec$ `'A'` (ECTS grade)

Encoding Categorical Features in Python: `OneHotEncoder` (1/2)

One-hot encoding: the good way to encode categorical features

Categorical features can (and should) be converted to numerical features with a one-hot encoding (aka one-of-K or dummy encoding). To this end, `OneHotEncoder` transforms each categorical feature with `n_categories` possible values into `n_categories` binary features, with one of them 1, and all others 0.

```
from numpy import array
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
# define example
data = ['cold', 'cold', 'warm', 'cold', 'hot',
        'hot', 'warm', 'cold', 'warm', 'hot']
values = array(data)
print("Data: ", values)
# integer encode
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(values)
print("Label Encoder:" ,integer_encoded)
# onehot encode
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded=integer_encoded.reshape(len(integer_encoded),1)
onehot_encoded=onehot_encoder.fit_transform(integer_encoded)
print("OneHot Encoder:", onehot_encoded)
```

Output

```
Data:  ['cold' 'cold' 'warm' 'cold'
'hot' 'hot' 'warm' 'cold' 'warm' 'hot']
Label Encoder:  [0 0 2 0 1 1 2 0 2 1]
OneHot Encoder:  [[1.  0.  0.]
  [1.  0.  0.]
  [0.  0.  1.]
  [1.  0.  0.]
  [0.  1.  0.]
  [0.  1.  0.]
  [0.  0.  1.]
  [1.  0.  0.]
  [0.  0.  1.]
  [0.  1.  0.]]
```

Continuing the example in previous slide:

```
>>> enc = preprocessing.OneHotEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None, categories=None, dtype=<... 'numpy.float64'>,
              handle_unknown='error', n_values=None, sparse=True)
>>> enc.transform([['female', 'from US', 'uses Safari'], ['male','from Europe','uses Safari']]).toarray()
array([[1., 0., 0., 1., 0., 1.],
       [0., 1., 1., 0., 0., 1.]])
```

By default, the values each feature can take is inferred automatically from the dataset and can be found in the `categories_` attribute:

```
>>> enc.categories_
[array(['female', 'male'], dtype=object),
 array(['from Europe', 'from US'], dtype=object),
 array(['uses Firefox', 'uses Safari'], dtype=object)]
```

Encoding Categorical Features in Python: `OneHotEncoder` (2/2)

It is possible to specify this explicitly using the parameter `categories`. There are two genders, four possible continents and four web browsers in our dataset:

```
>>> genders = ['female', 'male']
>>> locations = ['from Africa', 'from Asia', 'from Europe', 'from US']
>>> browsers = ['uses Chrome', 'uses Firefox', 'uses IE', 'uses Safari']
>>> enc = preprocessing.OneHotEncoder(categories=[genders, locations, browsers])
>>> # Note that for there are missing categorical values for the 2nd and 3rd
>>> # feature
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None, categories=[...],
       dtype=<... 'numpy.float64'>, handle_unknown='error',
       n_values=None, sparse=True)
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 1., 0., 0., 1., 0., 0., 0.]])
```

If there is a possibility that the training data might have missing categorical features, it can often be better to specify `handle_unknown='ignore'` instead of setting the `categories` manually as above. When `handle_unknown='ignore'` is specified and unknown categories are encountered during transform, no error will be raised but the resulting one-hot encoded columns for this feature will be all zeros (`handle_unknown='ignore'` is only supported for one-hot encoding):

```
>>> enc = preprocessing.OneHotEncoder(handle_unknown='ignore')
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None, categories=None,
       dtype=<... 'numpy.float64'>, handle_unknown='ignore',
       n_values=None, sparse=True)
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 0., 0., 0.]])
```

sklearn.preprocessing.OneHotEncoder

`OneHotEncoder` encodes categorical integer features as a one-hot numeric array (*i.e.*, "one-of-K" or "dummy" encoding scheme). Its input should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. It creates a binary column for each category and returns a sparse matrix or dense array.

This encoding is needed for feeding categorical data to many scikit-learn estimators, notably linear models and SVMs with the standard kernels.

Example: For a 2-feature dataset, the encoder finds the unique values per feature and transform the data to a binary one-hot encoding.

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> enc = OneHotEncoder(handle_unknown='ignore')
>>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
>>> enc.fit(X)
...
OneHotEncoder(categorical_features=None, categories=None,
       dtype=<... 'numpy.float64'>, handle_unknown='ignore',
       n_values=None, sparse=True)
```

```
>>> enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> enc.transform([['Female', 1], ['Male', 4]]).toarray()
array([[1., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0.]])
>>> enc.inverse_transform([[0, 1, 1, 0, 0], [0, 0, 0, 1, 0]])
array([['Male', 1],
       [None, 2]], dtype=object)
>>> enc.get_feature_names()
array(['x0_Female', 'x0_Male', 'x1_1', 'x1_2', 'x1_3'], dtype=object)
```

Methods:

`fit(X[, y])`	Fit OneHotEncoder to X
`fit_transform(X[, y])`	Fit OneHotEncoder to X, then transform X
`get_feature_names([input_features])`	Return feature names for output features
`get_params([deep])`	Get parameters for this estimator
`inverse_transform(X)`	Convert the back data to the original representation
`set_params(**params)`	Set the parameters of this estimator
`transform(X)`	Transform X using one-hot encoding

Encoding Categorical Features in Python: An Example

Column Transformer with Mixed Types

This example illustrates how to apply different preprocessing and feature extraction pipelines to different subsets of features, using `sklearn.compose.ColumnTransformer`. This is particularly handy for the case of datasets that contain heterogeneous data types, since we may want to scale the numeric features and one-hot encode the categorical ones.

In this example, the numeric data is standard-scaled after mean-imputation, while the categorical data is one-hot encoded after imputing missing values with a new category (`'missing'`). Finally, the preprocessing pipeline is integrated in a full prediction pipeline using `sklearn.pipeline.Pipeline`, together with a classification model.

Plot Column Transformer Mixed Types

```
# Author: Pedro Morales <part.morales@gmail.com>; License: BSD 3 clause

import pandas as pd
import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV

np.random.seed(0)

# Read data from Titanic dataset.
titanic_url = ('https://raw.githubusercontent.com/amueller/scipy-2017-sklearn/091d371/notebooks/datasets/titanic3.csv')
data = pd.read_csv(titanic_url)

# We will train our classifier with the following features:
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
                                      ('scaler', StandardScaler())])
categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
                                          ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(transformers=[('num', numeric_transformer, numeric_features),
                                               ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline. Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression(solver='lbfgs'))])

X = data.drop('survived', axis=1)
y = data['survived']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

Output

```
model score: 0.790
```

Outline

Preprocessing Data

Preprocessing Data: Standardization

Standardization, or mean removal and variance scaling to 1

Standardization (centering and reducing) of the training dataset:

$$\boldsymbol{x}_{i,\text{red}} = \frac{\boldsymbol{x}_i - \mu}{\sigma}$$

with $\mu = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i$ and $\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\boldsymbol{x}_i - \mu)^2}$.

Transforming a test data $\boldsymbol{x}$:

$$\boldsymbol{x}_{\text{red}} = \frac{\boldsymbol{x} - \mu}{\sigma}$$

where μ and σ estimated on the training dataset.

Every sklearn's transform's `fit()` just calculates the parameters (*e.g.* μ and σ in case of `StandardScaler`) and saves them as an internal objects state. Afterwards, you can call its `transform()` method to apply the transformation to a particular set of examples.

`fit_transform()` joins these two steps and is used for the initial fitting of parameters on the training set, but it also returns the transformed/standardized counterpart. Internally, it just calls first `fit()` and then `transform()` on the same data.

Preprocessing Data: Standardization in Python

Many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around zero and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

The function `scale` provides a quick and easy way to perform this operation on a single array-like dataset:

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

>>> X_scaled
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

Scaled data has zero mean and unit variance:

```
>>> X_scaled.mean(axis=0)
array([0., 0., 0.])

>>> X_scaled.std(axis=0)
array([1., 1., 1.])
```

The `preprocessing` module further provides a utility class `StandardScaler` that implements the `Transformer` API to compute the mean and standard deviation on a training set so as to be able to later reapply the same transformation on the testing set. This class is hence suitable for use in the early steps of a `sklearn.pipeline.Pipeline`:

```
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> scaler
StandardScaler(copy=True, with_mean=True, with_std=True)

>>> scaler.mean_
array([1. ..., 0. ..., 0.33...])

>>> scaler.scale_
array([0.81..., 0.81..., 1.24...])

>>> scaler.transform(X_train)
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

The scaler instance can then be used on new data to transform it the same way it did on the training set:

```
>>> X_test = [[-1., 1., 0.]]
>>> scaler.transform(X_test)
array([[-2.44...,  1.22..., -0.26...]])
```

It is possible to disable either centering or scaling by either passing `with_mean=False` or `with_std=False` to the constructor of `StandardScaler`.

Preprocessing Data: Scaling Features to a Range

An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using `MinMaxScaler` or `MaxAbsScaler`, respectively.

The motivation to use this scaling include robustness to very small standard deviations of features.

```
>>> X_train = np.array([[ 1., -1.,  2.],
                        [ 2.,  0.,  0.],
                        [ 0.,  1., -1.]])
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[0.5       , 0.        , 1.        ],
       [1.        , 0.5       , 0.33333333],
       [0.        , 1.        , 0.        ]])
```

The same instance of the transformer can then be applied to some new test data unseen during the fit call: the same scaling and shifting operations will be applied to be consistent with those performed on the train data:

```
>>> X_test = np.array([[-3., -1.,  4.]])
>>> X_test_minmax = min_max_scaler.transform(X_test)
>>> X_test_minmax
array([[-1.5       ,  0.        ,  1.66666667]])
```

It is possible to introspect the scaler attributes to find about the exact nature of the transformation learned on the training data:

```
>>> min_max_scaler.scale_
array([0.5       , 0.5       , 0.33...])
>>> min_max_scaler.min_
array([0.        , 0.5       , 0.33...])
```

If `MinMaxScaler` is given an explicit `feature_range=(min, max)`, the full formula is:

```
X_std = (X-X.min(axis=0))/(X.max(axis=0)-X.min(axis=0))

X_scaled = X_std * (max - min) + min
```

`MaxAbsScaler` works in a very similar fashion, but scales in a way that the training data lies within the range `[-1, 1]` by dividing through the largest maximum value in each feature. It is meant for data that is already centered at zero or sparse data.

Here is how to use the toy data from the previous example with this scaler:

```
>>> X_train = np.array([[ 1., -1.,  2.],
                        [ 2.,  0.,  0.],
                        [ 0.,  1., -1.]])

>>> max_abs_scaler = preprocessing.MaxAbsScaler()
>>> X_train_maxabs =max_abs_scaler.fit_transform(X_train)
>>> X_train_maxabs      # doctest +NORMALIZE_WHITESPACE^
array([[ 0.5, -1. ,  1. ],
       [ 1. ,  0. ,  0. ],
       [ 0. ,  1. , -0.5]])
>>> X_test = np.array([[ -3., -1.,  4.]])
>>> X_test_maxabs = max_abs_scaler.transform(X_test)
>>> X_test_maxabs
array([[-1.5, -1. ,  2. ]])
>>> max_abs_scaler.scale_
array([2.,  1.,  2.])
```

As with `scale`, the module further provides convenience functions `minmax_scale` and `maxabs_scale` if you don't want to create an object.

Final remark

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> # Don't cheat - fit only on training data
>>> scaler.fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> # apply same transformation to test data
>>> X_test = scaler.transform(X_test)
```

An alternative and recommended approach is to use `StandardScaler` in a `Pipeline`

Outline

Feature Selection

Feature Selection: Removing Low-variance Features

The classes in the `sklearn.feature_selection` module can be used for feature selection/dimensionality reduction on sample sets, either to improve estimators' accuracy scores or to boost their performance on very high-dimensional datasets. Thus, feature selection can be seen as a preprocessing step to an estimator.

Removing features with low variance:

`VarianceThreshold` is a simple baseline approach to feature selection. It removes all features whose variance doesn't meet some threshold. By default, it removes all zero-variance features, *i.e.*, features that have the same value in all samples.

Example: suppose that we have a dataset with boolean features, and we want to remove all features that are either one or zero (on or off) in more than 80% of the samples. Boolean features are Bernoulli random variables, and the variance of such variables is given by $\text{Var}(x) = p(1-p)$, so we can select using the threshold `.8 * (1 - .8)`:

```
>>> from sklearn.feature_selection import VarianceThreshold
>>> X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
>>> sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
>>> sel.fit_transform(X)
array([[0, 1],
       [1, 0],
       [0, 0],
       [1, 1],
       [1, 0],
       [1, 1]])
```

As expected, `VarianceThreshold` has removed the first column, which has a probability of containing a zero.

Feature Selection: Univariate Feature Selection

Univariate feature selection works by selecting the best features based on univariate statistical tests. Scikit-learn exposes feature selection routines as objects that implement:

- `SelectKBest` removes all but the k-highest scoring features
- `SelectPercentile` removes all but a specified highest scoring percentage
- using common univariate statistical tests for each feature: false positive rate `SelectFpr`, false discovery rate `SelectFdr`, or family wise error `SelectFwe`.
- `GenericUnivariateSelect` allows to perform univariate feature selection with a configurable strategy. This allows to select the best univariate selection strategy with hyper-parameter search estimator.

Example: Perform a χ^2-test to the samples to retrieve only the two best features:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import chi2
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> X_new = SelectKBest(chi2, k=2).fit_transform(X, y)
>>> X_new.shape
(150, 2)
```

These objects take as input a scoring function that returns univariate scores and p-values (or only scores for `SelectKBest` and `SelectPercentile`):

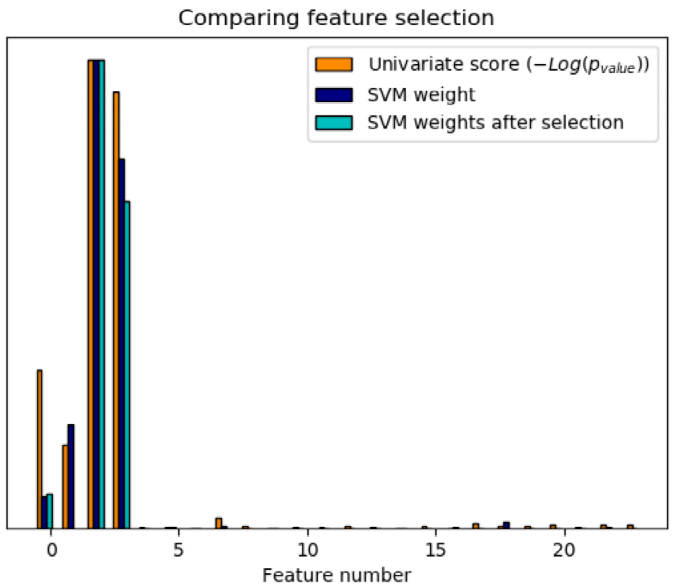
- For regression: `f_regression`, `mutual_info_regression`
- For classification: `chi2`, `f_classif`, `mutual_info_classif`

The methods based on F-test estimate the degree of linear dependency between two random variables. On the other hand, mutual information methods can capture any kind of statistical dependency, but being nonparametric, they require more samples for accurate estimation. See https://scikit-learn.org/.../plot_f_test_vs_mi.html.

Feature Selection: Univariate Feature Selection

Plot feature selection

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

from sklearn import datasets, svm
from sklearn.feature_selection import SelectPercentile, f_classif

# #############################################################
# Import some data to play with

# The iris dataset
iris = datasets.load_iris()

# Some noisy data not correlated
E = np.random.uniform(0, 0.1, size=(len(iris.data), 20))

# Add the noisy data to the informative features
X = np.hstack((iris.data, E))
y = iris.target

plt.figure(1)
plt.clf()

X_indices = np.arange(X.shape[-1])

# #############################################################
# Univariate feature selection with F-test for feature scoring
# We use the default selection function: the 10% most significant features
selector = SelectPercentile(f_classif, percentile=10)
selector.fit(X, y)
scores = -np.log10(selector.pvalues_)
scores /= scores.max()
plt.bar(X_indices - .45, scores, width=.2,
        label=r'Univariate score ($-Log(p_{value})$)', color='darkorange', edgecolor='black')

# #############################################################
# Compare to the weights of an SVM
clf = svm.SVC(kernel='linear')
clf.fit(X, y)

svm_weights = (clf.coef_ ** 2).sum(axis=0)
svm_weights /= svm_weights.max()

plt.bar(X_indices - .25, svm_weights, width=.2, label='SVM weight', color='navy', edgecolor='black')

clf_selected = svm.SVC(kernel='linear')
clf_selected.fit(selector.transform(X), y)

svm_weights_selected = (clf_selected.coef_ ** 2).sum(axis=0)
svm_weights_selected /= svm_weights_selected.max()

plt.bar(X_indices[selector.get_support()] - .05, svm_weights_selected,
        width=.2, label='SVM weights after selection', color='c', edgecolor='black')

plt.title("Comparing feature selection")
plt.xlabel('Feature number')
plt.yticks(())
plt.axis('tight')
plt.legend(loc='upper right')
plt.show()
```



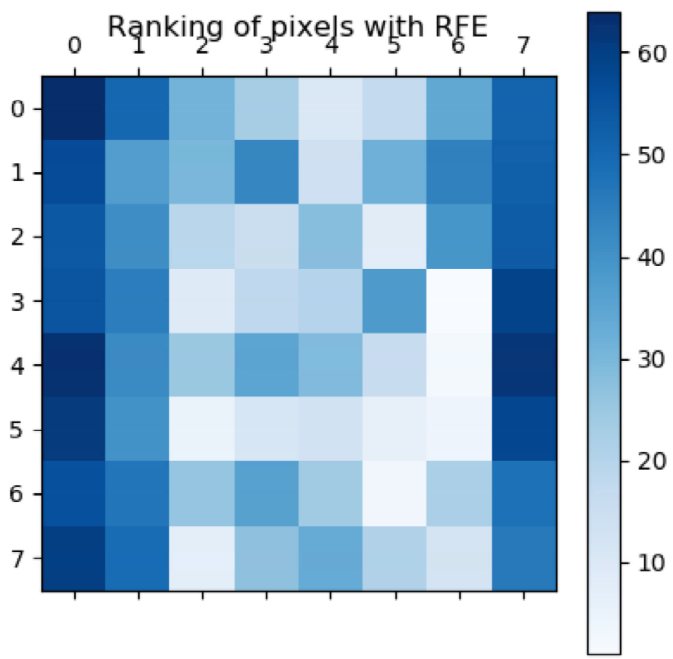
Feature Selection: Recursive Feature Elimination

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), recursive feature elimination (`RFE`) selects features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through a `coef_` attribute or through a `feature_importances_` attribute. Then, the least important features are pruned from current set of features.That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

`RFECV` performs RFE in a cross-validation loop to find the optimal number of features.

Plot RFE digits

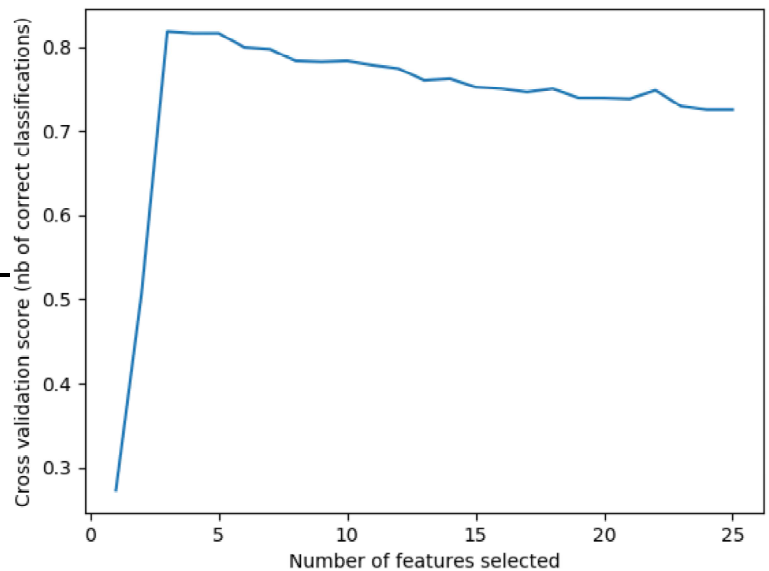
```
from sklearn.svm import SVC
from sklearn.datasets import load_digits
from sklearn.feature_selection import RFE
import matplotlib.pyplot as plt

# Load the digits dataset
digits = load_digits()
X = digits.images.reshape((len(digits.images), -1))
y = digits.target

# Create the RFE object and rank each pixel
svc = SVC(kernel="linear", C=1)
rfe = RFE(estimator=svc, n_features_to_select=1, step=1)
rfe.fit(X, y)
ranking = rfe.ranking_.reshape(digits.images[0].shape)

# Plot pixel ranking
plt.matshow(ranking, cmap=plt.cm.Blues)
plt.colorbar()
plt.title("Ranking of pixels with RFE")
plt.show()
```



Feature Selection: Recursive Feature Elimination with Cross-validation

Plot RFE with cross validation

```
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.datasets import make_classification

# Build a classification task using 3 informative features
X, y = make_classification(n_samples=1000, n_features=25, n_informative=3, n_redundant=2,
                           n_repeated=0, n_classes=8, n_clusters_per_class=1, random_state=0)

# Create the RFE object and compute a cross-validated score.
svc = SVC(kernel="linear")
# The "accuracy" scoring is proportional to the number of correct classifications
rfecv = RFECV(estimator=svc, step=1, cv=StratifiedKFold(2), scoring='accuracy')
rfecv.fit(X, y)

print("Optimal number of features : %d" % rfecv.n_features_)

# Plot number of features VS. cross-validation scores
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Cross validation score (nb of correct classifications)")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```



Feature Selection as Part of a Pipeline

Feature selection is usually used as a pre-processing step before doing the actual learning. The recommended way to do this in scikit-learn is to use a `sklearn.pipeline.Pipeline`.

Example:

```
clf = Pipeline([
  ('feature_selection', SelectFromModel(LinearSVC(penalty="l1"))),
  ('classification', RandomForestClassifier())
])
clf.fit(X, y)
```

In this example, a `sklearn.svm.LinearSVC` is coupled with `sklearn.feature_selection.SelectFromModel` to evaluate feature importances and select the most relevant one. Then, a `sklearn.ensemble.RandomForestClassifier` is trained on the transformed output, *i.e.*, using only relevant features. You can perform similar operations with the other feature selection methods and also classifiers that provide a way to evaluate feature importances of course. See the `sklearn.pipeline.Pipeline` examples for more details.

Outline

Final Remark

Pipeline

Pipeline of transforms with a final estimator.

Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implement fit and transform methods. The final estimator only needs to implement fit. The transformers in the pipeline can be cached using memory argument.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a '__', as in the example below. A step's estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting to None.

A simplified pipeline construction: `sklearn.pipeline.make_pipeline`

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.preprocessing import StandardScaler
>>> make_pipeline(StandardScaler(), GaussianNB(priors=None))
...
Pipeline(memory=None,
         steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('gaussiannb', GaussianNB(priors=None, var_smoothing=1e-09))])
```

sklearn.pipeline.Pipeline

```
>>> from sklearn import svm
>>> from sklearn.datasets import samples_generator
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import f_regression
>>> from sklearn.pipeline import Pipeline
>>> # generate some data to play with
>>> X, y = samples_generator.make_classification(
...     n_informative=5, n_redundant=0, random_state=42)
>>> # ANOVA SVM-C
>>> anova_filter = SelectKBest(f_regression, k=5)
>>> clf = svm.SVC(kernel='linear')
>>> anova_svm = Pipeline([('anova', anova_filter), ('svc', clf)])
>>> # You can set the parameters using the names issued
>>> # For instance, fit using a k of 10 in the SelectKBest
>>> # and a parameter 'C' of the svm
>>> anova_svm.set_params(anova__k=10, svc__C=.1).fit(X, y)
...
Pipeline(memory=None,
         steps=[('anova', SelectKBest(...)),
                ('svc', SVC(...))])
>>> prediction = anova_svm.predict(X)
>>> anova_svm.score(X, y)
0.83
>>> # getting the selected features chosen by anova_filter
>>> anova_svm.named_steps['anova'].get_support()
...
array([False, False,  True,  True, False, False, True,  True, False,
       True,  False,  True,  True, False, True,  False, True, True,
       False, False])
>>> # Another way to get selected features chosen by anova_filter
>>> anova_svm.named_steps.anova.get_support()
...
array([False, False,  True,  True, False, False, True,  True, False,
       True,  False,  True,  True, False, True,  False, True, True,
       False, False])
```

Outline: Next

Part 1: Introduction to Machine Learning

Part 2: ("Primal") Machine Learning Algorithms

Part 3: Statistical Learning Theory and Support Vector Machines

Part 4: Multiclass and Regression