# CONTENTS

1 Tutorial setup ................................................................. 3
   1.1 Install scikit-learn 0.7 ........................................ 3
   1.2 Download the datasets ........................................... 4

2 Machine Learning 101 ......................................................... 5
   2.1 Features and feature extraction ................................ 6
   2.2 Supervised Learning: `model.fit(X, y)` ...................... 8
   2.3 Unsupervised Learning: `model.fit(X)` ....................... 12
   2.4 Linearly separable data .......................................... 16
   2.5 Training set, test set and overfitting ......................... 16
   2.6 Key takeaway points ............................................... 20

3 Working with text data ...................................................... 21
   3.1 Downloading the data and loading it from Python ........ 21
   3.2 Extracting features from text files ........................... 22
   3.3 Training a linear classifier .................................... 24
   3.4 Building a pipeline ............................................... 24
   3.5 Evaluation of the performance on the test set ............. 25
   3.6 Parameter tuning using grid search ......................... 25

4 Exercises ........................................................................... 27
   4.1 Exercise 1: Sentiment Analysis on movie reviews .......... 27
   4.2 Exercise 2: Language identification ......................... 27
   4.3 Exercise 3: CLI text classification utility ................. 27
   4.4 Exercise 4: Face recognition .................................. 28

5 How to select the right algorithm for the task ........................ 29

6 Finding help ...................................................................... 31
   6.1 The project mailing list ........................................... 31
   6.2 Q&A communities with Machine Learning practitioners .. 31
scikits.learn is a Python module integrating classic machine learning algorithms in the tightly-knit world of scientific Python packages (numpy, scipy, matplotlib).
It aims to provide simple and efficient solutions to learning problems that are accessible to everybody and reusable in various contexts: **machine-learning as a versatile tool for science and engineering**.

**Note:** This document is meant to be used with scikit-learn version 0.7+. 
CHAPTER ONE

TUTORIAL SETUP

The following assumes you have extracted the source distribution of this tutorial somewhere on your local disk. Alternatively you can use git to clone this repo directly from github onto your local disk.

In the following we will name this folder $TUTORIAL_HOME. It should contain the following folders:

- **tutorial** - the source of the tutorial document written with sphinx
- **data** - folder to put the datasets used during the tutorial
- **skeletons** - sample incomplete scripts for the exercises
- **solutions** - solutions of the exercises

You can already copy the skeletons into a new folder named *workspace* where you will edit your own files for the exercises while keeping the original skeletons intact:

```bash
% cp -r skeletons workspace
```

### 1.1 Install scikit-learn 0.7

Please refer to the [scikit-learn install page](https://scikit-learn.org/stable/install.html) for per-system instructions.

You must have numpy, scipy and matplotlib installed first.

Here are the instructions to install the 0.7 release from source on a POSIX system (e.g. Linux and MacOSX). First download the release archive and extract it in the folder next to $TUTORIAL_HOME:

```bash
% wget http://pypi.python.org/packages/source/s/scikits.learn/scikits.learn-0.7.tar.gz
% tar zxvf scikits.learn-0.7.tar.gz
% cd scikits.learn-0.7
```

You can then build it locally and add it to your PYTHONPATH environment variable:

```bash
% python setup.py build_ext -i
% export PYTHONPATH='pwd'
```

If you want to install the library globally, do the following instead:

```bash
% python setup.py build
% sudo python setup.py install
```

Whatever the installation procedure you should check that the ‘0.7’ version is active in your python path:

```bash
% python -c "import scikits.learn; print scikits.learn.__version__"
0.7
```
You should also be able to launch the tests from anywhere in the system (if nose is installed) with the following:

```
% python -c "import scikits.learn as skl; skl.test()"
```

The output should end with OK as in:

```
----------------------------------------------------------------------
Ran 623 tests in 26.108s
OK (SKIP=2)
```

In the rest of the tutorial, the path to the extracted archive folder `scikits.learn-0.7` will be named `$SKL_HOME`.

### 1.2 Download the datasets

Machine Learning algorithms need data. Go to each `$TUTORIAL_HOME/data` sub-folder and run the `fetch_data.py` script from there (after having read them first).

For instance:

```
% cd $TUTORIAL_HOME/data/languages
% less fetch_data.py
% python fetch_data.py
```
CHAPTER TWO

MACHINE LEARNING 101

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

Machine Learning can be considered a subfield of Artificial Intelligence since those algorithms can be seen as building blocks to make computer learn to behave more intelligently by somehow generalizing rather than just storing and retrieving data items like a database system would do.

Figure 2.1: Decision boundary learned from data points from two categories: white and black

The following will introduce the main concepts used to qualify machine learning algorithms as implemented in scikit-learn:

- how to turn raw data into numerical arrays
- what is supervised learning
- what is unsupervised learning
- what is linearly separable data
- what is overfitting
2.1 Features and feature extraction

Most machine learning algorithms implemented in scikit-learn expect a numpy array as input $X$. The expected shape of $X$ is $(n_{samples}, n_{features})$.

- **n_samples** The number of samples: each sample is an item to process (e.g. classify). A sample can be a document, a picture, a sound, a video, a row in database or CSV file, or whatever you can describe with a fixed set of quantitative traits.

- **n_features** The number of features or distinct traits that can be used to describe each item in a quantitative manner.

The number of features must be fixed in advance. However it can be very high dimensional (e.g. millions of features) with most of them being zeros for a given sample. In this case we use scipy.sparse matrices instead of numpy arrays so has to make the data fit in memory.

2.1.1 A simple example: the iris dataset

The machine learning community often uses a simple flowers database were each row in the database (or CSV file) is a set of measurements of an individual iris flower.

![Iris Virginia](source: Wikipedia)

Figure 2.2: Iris Virginia (source: Wikipedia)

Each sample in this dataset is described by 4 features and can belong to one of the target classes:

**Features in the Iris dataset**

0. sepal length in cm
1. sepal width in cm
2. petal length in cm
3. petal width in cm

**Target classes to predict**

0. Iris Setosa
1. Iris Versicolour
2. Iris Virginica

scikit-learn embeds a copy of the iris CSV file along with a helper function to load it into numpy arrays:
>>> from scikits.learn.datasets import load_iris
>>> iris = load_iris()

**Note:** To be able to copy and paste examples without taking care of the leading `>>>` and `...` prompt signs, enable the ipython doctest mode with: `%doctest_mode

The features of each sample flower is stored in the `data` attribute of the dataset:

```python
>>> n_samples, n_features = iris.data.shape

>>> n_samples
150

>>> n_features
4
```

```python
>>> iris.data[0]
array([ 5.1, 3.5, 1.4, 0.2])
```

The information about the class of each sample is stored in the `target` attribute of the dataset:

```python
>>> len(iris.target) == n_samples
True

>>> iris.target
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

The names of the classes is stored in the last attribute, namely `target_names`:

```python
>>> list(iris.target_names)
['setosa', 'versicolor', 'virginica']
```

### 2.1.2 Handling categorical features

Sometimes people describe samples with categorical descriptors that have no obvious numerical representation. For instance assume that each flower is further described by a color name among a fixed list of color names:

```python
color in ['purple', 'blue', 'red']
```

The simple way to turn this categorical feature into numerical features suitable for machine learning is to create new features for each distinct color name that can be valued to 1.0 if the category is matching or 0.0 if not.

The enriched iris feature set would hence be in this case:

- 0. sepal length in cm
- 1. sepal width in cm
- 2. petal length in cm
- 3. petal width in cm
- 4. color#purple (1.0 or 0.0)
5. color (1.0 or 0.0)
6. color (1.0 or 0.0)

2.1.3 Extracting features from unstructured data

The previous example deals with features that are readily available in a structured dataset with rows and columns of numerical or categorical values.

However, most of the produced data is not readily available in a structured representation such as SQL, CSV, XML, JSON or RDF.

Here is an overview of strategies to turn unstructured data items into arrays of numerical features.

**Text documents** Count the frequency of each word or pair of consecutive words in each document. This approach is called the **Bag of Words**.

Note: we include other file formats such as HTML and PDF in this category: an ad-hoc preprocessing step is required to extract the plain text in UTF-8 encoding for instance.

**Images**

- Rescale the picture to a fixed size and **take all the raw pixels values** (with or without luminosity normalization)
- Take some transformation of the signal (gradients in each pixel, wavelets transforms...)
- Compute the Euclidean, Manhattan or cosine **similarities of the sample to a set reference prototype images** arranged in a code book. The code book may have been previously extracted on the same dataset using an unsupervised learning algorithms on the raw pixel signal.
  
  Each feature value is the distance to one element of the code book.

- Perform **local feature extraction**: split the picture into small regions and perform feature extraction locally in each area.

  Then combine all the feature of the individual areas into a single array.

**Sounds** Same strategy as for images with in a 1D space instead of 2D

Practical implementations of such feature extraction strategies will be presented in the last sections of this tutorial.

2.2 Supervised Learning: `model.fit(X, y)`

A supervised learning algorithm makes the distinction between the raw observed data $X$ with shape $(n_{samples}, n_{features})$ and some label given to the model while training by some teacher. In scikit-learn this array is often noted $y$ and has generally the shape $(n_{samples},)$.

After training, the fitted model does no longer expect the $y$ as an input: it will try to predict the most likely labels $y_{new}$ for new a set of samples $X_{new}$.

Depending on the nature of the target $y$, supervised learning can be given different names:

- If $y$ has values in a fixed set of **categorical outcomes** (represented by integers) the task to predict $y$ is called **classification**.
- If $y$ has **floating point values** (e.g. to represent a price, a temperature, a size...), the task to predict $y$ is called **regression**.
### 2.2.1 Classification

**A first classifier example with scikit-learn**

In the iris dataset example, suppose we are assigned the task to guess the class of an individual flower given the measurements of petals and sepals. This is a classification task, hence we have:

```python
>>> X, y = iris.data, iris.target
```

Once the data has this format it is trivial to train a classifier, for instance a support vector machine with a linear kernel (or lack of thereof):

```python
>>> from scikits.learn.svm import LinearSVC
>>> clf = LinearSVC()
```

**Note:** Whenever you import a scikit-learn class or function of the first time, you are advised to read the docstring by using the `?` magic suffix of ipython, for instance `type: LinearSVC?`.

`clf` is a statistical model that has parameters that control the learning algorithm (those parameters are sometimes called the hyper-parameters). Those hyperparameters can be supplied by the user in the constructor of the model. We will explain later choose a good combination either using simple empirical rules or data driven selection:

```python
>>> clf
LinearSVC(loss='l2', C=1.0, intercept_scaling=1, fit_intercept=True,
          eps=0.0001, penalty='l2', multi_class=False, dual=True)
```

By default the real model parameters are not initialized. They will be automatically be tuned from the data by calling the `fit` method:

```python
>>> clf = clf.fit(X, y)
```

```python
>>> clf.coef_
array([[ 0.18423474, 0.45122764, -0.80794654, -0.45071379],
       [ 0.04864394, -0.88914385, 0.40540293, -0.93720122]],
```

---

**2.2. Supervised Learning:** `model.fit(X, y)`
Once the model is trained, it can be used to predict the most likely outcome on unseen data. For instance, let us define a list of a simple sample that looks like the first sample of the iris dataset:

```python
X_new = [[ 5.0, 3.6, 1.3, 0.25]]

clf.predict(X_new)
```

The outcome is 0 which is the id of the first iris class namely ‘setosa’.

The following figure places the location of the fit and predict calls on the previous flow diagram. The `vec` object is a vectorizer used for feature extractor that is not used in the case of the iris data which already comes as vectors of features:

![Figure 2.4: Supervised Learning with scikit-learn](image)

Some scikit-learn classifiers can further predict probabilities of the outcome. This is the case of logistic regression models:

```python
from scikits.learn.linear_model import LogisticRegression
clf2 = LogisticRegression().fit(X, y)
```

```python
clf2.predict_proba(X_new)
```

This means that the model estimates that the sample in `X_new` has:
• 90% likelihood to be belong to the ‘setosa’ class
• 9% likelihood to be belong to the ‘versicolor’ class
• 1% likelihood to be belong to the ‘virginica’ class

Of course the predict method that output the label id of the most likely outcome is also available:

```python
>>> clf2.predict(X_new)
array([0], dtype=int32)
```

Notable implementations of classifiers

- **scikit.learn.linear_model.LogisticRegression** Regularized Logistic Regression based on liblinear
- **scikit.learn.svm.LinearSVC** Support Vector Machines without kernels based on liblinear
- **scikit.learn.svm.SVC** Support Vector Machines with kernels based on libsvm
- **scikit.learn.linear_model.SGDClassifier** Regularized linear models (SVM or logistic regression) using a Stochastic Gradient Descent algorithm written in Cython
- **scikit.learn.neighbors.NeighborsClassifier** k-Nearest Neighbors classifier based on the ball tree datastructure for low dimensional data and brute force search for high dimensional data

Sample application of classifiers

The following table gives examples of applications of classifiers for some common engineering tasks:

<table>
<thead>
<tr>
<th>Task</th>
<th>Predicted outcomes</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-mail classification</td>
<td>Spam, normal, priority mail en, es, de, fr, ja, zh, ar, ru...</td>
</tr>
<tr>
<td>Language identification in text documents</td>
<td>Business, technology, sports...</td>
</tr>
<tr>
<td>News articles categorization</td>
<td>Negative, neutral, positive</td>
</tr>
<tr>
<td>Sentiment Analysis in customer feedback</td>
<td>Same / different persons</td>
</tr>
<tr>
<td>Face verification in pictures</td>
<td>Same / different persons</td>
</tr>
<tr>
<td>Speaker verification on voice recordings</td>
<td>Same / different persons</td>
</tr>
</tbody>
</table>

2.2.2 Regression

Regression is the task to predict the value of a continuously varying variable (e.g. a price, a temperature, a conversion rate...) given some input variables (a.k.a. the features, “predictors” or “regressors”). Some notable implementations of regression model in scikit-learn include:

- **scikit.learn.linear_model.Ridge** L2-regularized least squares linear model
- **scikit.learn.linear_model.ElasticNet** L1+L2-regularized least squares linear model trained using Coordinate Descent
- **scikit.learn.linear_model.LassoLARS** L1-regularized least squares linear model trained with Least Angle Regression
- **scikit.learn.linear_model.SGDRegressor** L1+L2-regularized least squares linear model trained using Stochastic Gradient Descent
- **scikit.learn.linear_model.ARDRegression** Bayesian Automated Relevance Determination regression

2.2. Supervised Learning: `model.fit(X, y)`
scikit.learn.svm.SVR Non-linear regression using Support Vector Machines (wrapper for libsvm)

2.3 Unsupervised Learning: `model.fit(X)`

An unsupervised learning algorithm only uses a single set of observations $X$ with shape $(n_{samples}, n_{features})$ and does not use any kind of labels.

An unsupervised learning model will try to fit its parameters so as to best summarize regularities found in the data.

The following introduces the main variants of unsupervised learning algorithms namely dimensionality reduction and clustering.

2.3.1 Dimensionality Reduction and visualization

Dimensionality reduction the task to derive a set of new artificial features that is smaller than the original feature set while retaining most of the variance of the original data.

Normalization and visualization with PCA

The most common technique for dimensionality reduction is called Principal Component Analysis.

PCA can be done using linear combinations of the original features using a truncated Singular Value Decomposition of the matrix $X$ so as to project the data onto a base of the top singular vectors.

If the number of retained components is 2 or 3, PCA can be used to visualize the dataset:

```python
>>> from scikit_learn.pca import PCA
>>> pca = PCA(n_components=2, whiten=True).fit(X)
```

Once fitted, the `pca` model exposes the singular vectors as in the `components_` attribute:
Let us project the iris dataset along those first 3 dimensions:

```python
>>> X_pca = pca.transform(X)
```

The dataset has been “normalized”, which means that the data is now centered on both components with unit variance:

```python
>>> X_pca.mean(axis=0)
array([-1.42478621e-15, 1.71936539e-15])
>>> X_pca.std(axis=0)
array([1., 1.])
```

Furthermore the samples components do no longer carry any linear correlation:

```python
>>> import numpy as np
>>> np.corrcoef(X_pca.T)
array([[ 1.          , 4.60742555e-16],
[ 4.60742555e-16, 1.          ]])
```

And visualize the dataset using `pylab`, for instance by defining the following utility function:

```python
>>> import pylab as pl
>>> from itertools import cycle
>>> def plot_2D(data, target, target_names):
...     colors = cycle('rgbcmykw')
...     target_ids = range(len(target_names))
...     pl.figure()
...     for i, c, label in zip(target_ids, colors, target_names):
...         pl.scatter(data[target == i, 0], data[target == i, 1],
...                     c=c, label=label)
...     pl.legend()
...
```

Calling `plot_2D(X_pca, iris.target, iris.target_names)` will display the following:

---

**Note:** The default implementation of PCA computes the SVD of the full data matrix which is not scalable when both `n_samples` and `n_features` are big (more than a few thousands).

If you are interested in a number of components that is much smaller than both `n_samples` and `n_features` consider using `scikits.learn.pca.RandomizedPCA` instead.

---

**Other applications of dimensionality reduction**

Dimensionality Reduction is not just useful for visualization of high dimensional datasets. It can also be used as a preprocessing step (often called data normalization) to help speed up supervised machine learning that are not computationally efficient with high `n_features` such as SVM classifiers with gaussian kernels for instance or that do not work well with linearly correlated features.
2.3.2 Clustering

Clustering is the task of gathering samples into groups of similar samples according to some predefined similarity or dissimilarity measure (such as the Euclidean distance).

For instance let us reuse the output of the 2D PCA of the iris dataset and try to find 3 groups of samples using the simplest clustering algorithm (KMeans):

```python
>>> from scikits.learn.cluster import KMeans
>>> from numpy.random import RandomState

>>> rng = RandomState(42)

>>> kmeans = KMeans(3, rng=rng).fit(X_pca)

>>> kmeans.cluster_centers_
array([[ 1.01505989, -0.70632886],
       [ 0.33475124, 0.89126382],
       [-1.287003 , -0.43512572]])

>>> kmeans.labels_[:10]
array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2])

>>> kmeans.labels_[-10:]
array([0, 0, 1, 0, 0, 0, 1, 0, 0, 1])
```

We can plot the assigned cluster labels instead of the target names with:
Notable implementations of clustering models

The following are two well known clustering algorithms. Like most unsupervised learning models in the scikit, they expect the data to be cluster to have shape \((n_{\text{samples}}, n_{\text{features}})\):

- \texttt{scikits.learn.cluster.KMeans} The simplest yet effective clustering algorithm. Need to be provided the number of clusters in advance and assume that the data is normalized as input (but use a PCA model as preprocessor).

- \texttt{scikits.learn.cluster.MeanShift} Can find better looking clusters than KMeans but is not scalable to high number of samples.

Other clustering algorithms do not work with a data matrix with shape \((n_{\text{samples}}, n_{\text{features}})\) but directly with a precomputed affinity matrix with shape \((n_{\text{samples}}, n_{\text{samples}})\):

- \texttt{scikits.learn.cluster.AffinityPropagation} Clustering algorithm based on message passing between data points.

- \texttt{scikits.learn.cluster.SpectralClustering} KMeans applied to a projection of the normalized graph Laplacian: finds normalized graph cuts if the affinity matrix is interpreted as an adjacency matrix of a graph.

Hierarchical clustering is being implemented in a branch that is likely to be merged into master before the release of scikit-learn 0.8.

Applications of clustering

Here are some common applications of clustering algorithms:

- Building customer profiles for market analysis
• Grouping related web news (e.g. Google News) and websearch results
• Grouping related stock quotes for investment portfolio management
• Can be used as a preprocessing step for recommender systems
• Can be used to build a code book of prototype samples for unsupervised feature extraction for supervised learning algorithms

2.4 Linearly separable data

Some supervised learning problems can be solved by very simple models (called generalized linear models) depending on the data. Others simply don’t.

To grasp the difference between the two cases run the interactive example from the examples folder of the scikit-learn source distribution:

```bash
% python $SKL_HOME/examples/applications/svm_gui.py
```

1. Put some data points belonging to one of the two target classes (‘white’ or ‘black’) using left click and right click.
2. Choose some parameters of a Support Vector Machine to be trained on this toy dataset (n_samples is the number of clicks, n_features is 2).
3. Click the Fit but to train the model and see the decision boundary. The accuracy of the model is displayed on stdout.

The following figures demonstrate one case where a linear model can perfectly separate the two classes while the other is not linearly separable (a model with a gaussian kernel is required in that case).

**Exercise**  Fit a model that is able to solve the XOR problem using the GUI: the XOR problem is composed of 4 samples:

- 2 white samples in the top-left and bottom-right corners
- 2 black samples in the bottom-left and top-right corners

**Question:** is the XOR problem linearly separable?

**Exercise**  Construct a problem with less than 10 points where the predictive accuracy of the best linear model is 50%.

2.5 Training set, test set and overfitting

The most common mistake beginners do when training statistical models is to evaluate the quality of the model on the same data using for fitting the model:

If you do this, you are doing it wrong!

2.5.1 The overfitting issue

The problem lies in the fact that some models can be subject to the overfitting issue: they can learn the training data by heart without generalizing. The symptoms are:

- the predictive accuracy on the data used for training can be excellent (sometimes 100%)
- however they do little better than random prediction when facing new data not part of the training set
Figure 2.8: Linear Support Vector Machine trained to perfectly separate 2 sets of data points labeled as white and black in a 2D space.
Figure 2.9: Support Vector Machine with gaussian kernel trained to separate 2 sets of data points labeled as white and black in a 2D space. This dataset would not have been separated by a simple linear model.
If you evaluate your model on your training data you won’t be able to tell whether your model is overfitting or not.

2.5.2 Solutions to overfitting

The solution to this issue is twofold:

1. Split your data into two sets to detect overfitting situations:
   - one for training and model selection: the **training set**
   - one for evaluation: the **test set**

2. Avoid overfitting by using simpler models (e.g. linear classifiers instead of gaussian kernel SVM) or by increasing the regularization parameter of the model if available (see the docstring of the model for details)

2.5.3 Measuring classification performance on a test set

Here is an example on you to split the data on the iris dataset.

First we need to shuffle the order of the samples and the target to ensure that all classes are well represented on both sides of the split:

```python
>>> indices = np.arange(n_samples)
>>> indices[:10]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```python
>>> RandomState(42).shuffle(indices)
>>> indices[:10]
array([ 73, 18, 118, 78, 76, 31, 64, 141, 68, 82])
```

```python
>>> X = iris.data[indices]
>>> y = iris.target[indices]
```

We can now split the data using a 2/3 - 1/3 ratio:

```python
>>> split = (n_samples * 2) / 3
>>> X_train, X_test = X[:split], X[split:]
>>> y_train, y_test = y[:split], y[split:]
```

```python
>>> X_train.shape
(100, 4)
```

```python
>>> X_test.shape
(50, 4)
```

```python
>>> y_train.shape
(100,)
```

```python
>>> y_test.shape
(50,)
```

We can now re-train a new linear classifier on the training set only:

```python
>>> clf = LinearSVC().fit(X_train, y_train)
```

To evaluate its quality we can compute the average number of correct classification on the test set:
>>> np.mean(clf.predict(X_test) == y_test)
1.0

This shows that the model has a predictive accuracy of 100% which means that the classification model was perfectly capable on generalizing what was learned from the training set to the test set: this is rarely so easy on real life datasets as we will see on the following chapter.

2.6 Key takeaway points

- **Build** $X$ (features vectors) with shape $(n_{samples}, n_{features})$
- **Supervised learning**: `clf.fit(X, y)` and then `clf.predict(X_new)`
  - Classification: $y$ is an array of integers
  - Regression: $y$ is an array of floats
- **Unsupervised learning**: `clf.fit(X)`
  - Dimensionality Reduction with `clf.transform(X_new)`
    * for visualization
    * for scalability
  - Clustering finds group id for each sample
- Some models work much better with data normalized with PCA
- Simple linear models can fail completely (non linearly separable data)
- Simple linear models often very useful in practice (esp. with large $n_{features}$)
- Before starting to training a model: split train / test data:
  - use training set for model selection and fitting
  - use test set for model evaluation
- Complex models can overfit (learn by heart) the training data and fail to generalize correctly on test data:
  - try simpler models first
  - tune the regularization parameter on a validation set
WORKING WITH TEXT DATA

The goal of this section is to explore some of the main scikit-learn tools on a single practical task: analysing a collection of text documents (newsgroups posts) on twenty different topics.

In this section we will see how to:

- load the file contents and the categories
- extract feature vectors suitable for machine learning
- train a linear model to perform categorization
- use a grid search strategy to find a good configuration of both the feature extraction components and the classifier

3.1 Downloading the data and loading it from Python

The dataset is called “Twenty Newsgroups”. Here is the official description, quoted from the website:

The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. To the best of my knowledge, it was originally collected by Ken Lang, probably for his Newsweeder: Learning to filter netnews paper, though he does not explicitly mention this collection. The 20 newsgroups collection has become a popular data set for experiments in text applications of machine learning techniques, such as text classification and text clustering.

To download the dataset, go to $TUTORIAL_HOME/twenty_newsgroups run the fetch_data.py script.

Once the data is downloaded, fire an ipython shell in the $TUTORIAL_HOME folder and define a variable to hold the list of categories to load. In order to get fast execution times for this first example we will work on a partial dataset with only 4 categories out of the 20 available in the dataset:

```python
>>> categories = ['alt.atheism', 'soc.religion.christian',
                 'comp.graphics', 'sci.med']
```

We can now load the list of files matching those categories as follows:

```python
>>> from scikits.learn.datasets import load_files
>>> twenty_train = load_files('data/twenty_newsgroups/20news-bydate-train',
                          categories=categories)
```

The returned dataset is a scikit-learn “bunch”: a simple holder object with fields that can be both accessed as python dict keys or object attributes for convenience, for instance the target_names holds the list of the requested category names:

```python
>>> twenty_train.target_names
['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']
```
The files themselves are not loaded in memory yet:

```python
>>> twenty_train.filenames.shape
(2257,)
>>> twenty_train.filenames[0]
'data/twenty_newsgroups/20news-bydate-train/comp.graphics/38244'
```

Let us print the first 2 lines of the first file:

```python
>>> print ''.join(open(twenty_train.filenames[0]).readlines()[:2]).strip()  
From: clipper@mccarthy.csd.uwo.ca (Khun Yee Fung)  
Subject: Re: looking for circle algorithm faster than Bresenham's
```

Supervised learning algorithms will require the category to predict for each document. In this case the category is the name of the newsgroup which also happens to be the name of folder holding the individual documents.

For speed and space efficiency reasons scikit-learn loads the target attribute as an array of integers that corresponds to the index of the category name in the `target_names` list. The category integer id of each sample is stored in the `target` attribute:

```python
>>> twenty_train.target[:10]
array([1, 0, 2, 2, 0, 1, 1, 3, 3, 2])
```

It is possible to get back the category names as follows:

```python
>>> for t in twenty_train.target[:10]:
...     print twenty_train.target_names[t]
...  
comp.graphics
alt.atheism
sci.med
sci.med
alt.atheism
comp.graphics
comp.graphics
soc.religion.christian
soc.religion.christian
sci.med
```

You can notice that the samples have been shuffled randomly (with a fixed RNG seed); this is useful if you select only the first samples to quickly train a model and get a first idea of the results before re-training on the complete dataset later.

### 3.2 Extracting features from text files

In order to perform machine learning on text documents, one first need to turn the text content into numerical feature vectors.

#### 3.2.1 Bags of words

The most intuitive way to do so is the bags of words representation:

1. assign a fixed integer id to each word occurring in any document of the training set (for instance by building a dictionary from words to integer indices).
2. for each document #i, count the number of occurrences of each word w and store it in $X[i, j]$ as the the value of feature #j where $j$ is the index of word w in the dictionary
The bags of words representation implies that \( n_{\text{features}} \) is the number of distinct words in the corpus: this number is typically larger that 100,000.

If \( n_{\text{samples}} == 10000 \), storing \( X \) as a numpy array of type float32 would require \( 10000 \times 100000 \times 4 \) bytes = **4GB in RAM** which is barely manageable on today’s computers.

Fortunately, **most values in \( X \) will be zeros** since for a given document less than a couple thousands of distinct words will be used. For this reason we say that bags of words are typically **high-dimensional sparse datasets**.

Hence it is highly recommended to use \texttt{scipy.sparse} matrices instead of numpy arrays to store the extracted features of a text corpus.

### 3.2.2 Tokenizing text with \texttt{scikit-learn}

\texttt{scikit-learn} offers a couple of basic yet useful utilities to work with text data. The first one is a preprocessor that removes accents and convert to lowercase on roman languages:

```python
>>> from scikit_learn.feature_extraction.text import RomanPreprocessor
>>> text = u"J’ai bien mangé."
>>> print RomanPreprocessor().preprocess(text)
j’ai bien mange.
```

The second one is a utility that extract that splits the text into words after having applied the preprocessor:

```python
>>> from scikit_learn.feature_extraction.text import WordNGramAnalyzer
>>> WordNGramAnalyzer().analyze(text)
[‘ai’, ‘bien’, ‘mange’]
```

Note that punctuation and single letter words have automatically been removed.

It is further possible to configure \texttt{WordNGramAnalyzer} to extract n-grams instead of single words:

```python
>>> WordNGramAnalyzer(min_n=1, max_n=2).analyze(text)
[u’ai’, u’bien’, u’mange’, u’ai bien’, u’bien mange’]
```

These tools are wrapped into a higher level component that is able to build a dictionary of features:

```python
>>> from scikit_learn.feature_extraction.text.sparse import CountVectorizer
>>> count_vect = CountVectorizer()
>>> docs_train = [open(f).read() for f in twenty_train.filenames]
>>> _ = count_vect.fit(docs_train)
```

Once fitted, the vectorizer has build a dictionary of feature indices:

```python
>>> count_vect.vocabulary.get(‘algorithm’)
1513
```

The index value of a word in the vocabulary is linked to its frequency in the whole training corpus.

Once the vocabulary is built, it is possible to rescan the training set so as to perform the actual feature extraction:

```python
>>> X_train_counts = count_vect.transform(docs_train)
>>> X_train_counts.shape
(2257, 33881)
```

### 3.2.3 From occurrences to frequencies

Occurrence count is a good start but there is an issue: longer documents will have higher average count values than shorter document, even though they might talk about the same topics.
To avoid these potential discrepancies it suffices to divide the number of occurrences of each word in a document by the total number of words in the document: these new features are called “TF” for Term Frequencies.

Another refinement on top of TF is to downscale weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus.

This downscaling is called TF-IDF for “Term Frequency times Inverse Document Frequency”.

Both TF and TF-IDF can be computed as follows:

```python
>>> from scikits.learn.feature_extraction.text.sparse import TfidfTransformer
>>> tf_transformer = TfidfTransformer(use_idf=False).fit(X_train_counts)
>>> X_train_tf = tf_transformer.transform(X_train_counts)
>>> X_train_tf.shape
(2257, 33881)
```

```python
>>> tfidf_transformer = TfidfTransformer().fit(X_train_counts)
>>> X_train_tfidf = tfidf_transformer.transform(X_train_counts)
>>> X_train_tfidf.shape
(2257, 33881)
```

### 3.3 Training a linear classifier

Now that we have our feature, we can train a linear classifier to try to predict the category of a post:

```python
>>> from scikits.learn.svm.sparse import LinearSVC
>>> clf = LinearSVC(C=1000).fit(X_train_tfidf, twenty_train.target)
```

To try to predict the outcome on a new document we need to extract the features using the same feature extracting chain:

```python
>>> docs_new = ['God is love', 'OpenGL on the GPU is fast']
>>> X_new_counts = count_vect.transform(docs_new)
>>> X_new_tfidf = tfidf_transformer.transform(X_new_counts)

>>> predicted = clf.predict(X_new_tfidf)

>>> for doc, category in zip(docs_new, predicted):
...     print '$r == $s' % (doc, twenty_train.target_names[category])
...
'God is love' => soc.religion.christian
'OpenGL on the GPU is fast' => comp.graphics
```

### 3.4 Building a pipeline

In order to make the vectorizer => transformer => classifier easier to work with, scikit-learn provides a Pipeline class that behaves like a compound estimator:

```python
>>> from scikits.learn.pipeline import Pipeline
>>> text_clf = Pipeline([
...    ('vect', CountVectorizer()),
...    ('tfidf', TfidfTransformer()),
...    ('clf', LinearSVC(C=1000)),
...])
```
We can now train the model with a single command:

```python
>>> _ = text_clf.fit(docs_train, twenty_train.target)
```

### 3.5 Evaluation of the performance on the test set

Evaluating the predictive accuracy of the model is equally easy:

```python
>>> import numpy as np

>>> twenty_test = load_files('data/twenty_newsgroups/20news-bydate-test',
... categories=categories)

>>> docs_test = [open(f).read() for f in twenty_test.filenames]

>>> predicted = text_clf.predict(docs_test)

>>> np.mean(predicted == twenty_test.target)
0.93075898801597867
```

*scikit-learn* further provides utilities for more detailed performance analysis of the results:

```python
>>> from scikits.learn import
classification_report

>>> print metrics.classification_report(twenty_test.target, predicted,
... class_names=twenty_test.target_names)

<table>
<thead>
<tr>
<th></th>
<th>precision</th>
<th>recall</th>
<th>f1-score</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt.atheism</td>
<td>0.93</td>
<td>0.85</td>
<td>0.89</td>
<td>319</td>
</tr>
<tr>
<td>comp.graphics</td>
<td>0.97</td>
<td>0.95</td>
<td>0.96</td>
<td>389</td>
</tr>
<tr>
<td>sci.med</td>
<td>0.94</td>
<td>0.95</td>
<td>0.95</td>
<td>396</td>
</tr>
<tr>
<td>soc.religion.christian</td>
<td>0.88</td>
<td>0.95</td>
<td>0.92</td>
<td>398</td>
</tr>
</tbody>
</table>

avg / total 0.93 0.93 0.93 1502

>>> metrics.confusion_matrix(twenty_test.target, predicted)
array([[271, 3, 9, 36],
       [ 4, 371, 9, 5],
       [ 4, 6, 377, 9],
       [11, 4, 4, 379]])
```

### 3.6 Parameter tuning using grid search

Instead of tweaking the parameters of the various components of the chain, it is possible to run an exhaustive search of the best parameters on a grid of possible values:

```python
>>> from scikits.learn.grid_search import GridSearchCV

>>> parameters = {
... 'vect__analyzer__max_n': (1, 2), # words or bigrams
... 'tfidf__use_idf': (True, False),
... 'clf__C': (100, 1000),
... }

>>> gs_clf = GridSearchCV(text_clf, parameters, n_jobs=-1)
```

The grid search instance behaves like a normal *scikit-learn* model. Let us perform the search on a smaller subset of the dataset to speed up the computation:
>>> gs_clf.fit(docs_train[:400], twenty_train.target[:400])

The best model found during fit is available as a special attribute:

```python
>>> best_parameters = gs_clf.best_estimator._get_params()
```

```python
>>> for param_name in sorted(parameters.keys()):
...     print "%s: %r" % (param_name, best_parameters[param_name])
... 
clf__C: 100
tfidf__use_idf: True
vect__analyzer__max_n: 2
```
EXERCISES

To do the exercises, copy the content of the ‘skeletons’ folder as a new folder named ‘workspace’.
Then fire an ipython shell and run the work-in-progress script with:

[1] %run workspace/exercise_XX_script.py arg1 arg2 arg3

If an exception is triggered, use the %debug to fire-up a post mortem ipdb session.
Refine the implementation and iterate until the exercise is solved.

4.1 Exercise 1: Sentiment Analysis on movie reviews

- Write a text classification pipeline to classify movie reviews as either positive or negative.
- Find a good set of parameters using grid search.
- Evaluate the performance on a held out test set.

ipython command line:

%run workspace/exercise_01_sentiment.py data/movie_reviews/txt_sentoken/

4.2 Exercise 2: Language identification

- Write a text classification pipeline using a custom preprocessor and CharNGramAnalyzer using data from Wikipedia articles as training set.
- Evaluate the performance on some held out test set.

ipython command line:

%run workspace/exercise_02_language_train_model.py data/languages/paragraphs/

4.3 Exercise 3: CLI text classification utility

Using the results of the previous exercises and the cPickle module of the standard library, write a command line utility that detect the language of some text provided on stdin and estimate the polarity (positive or negative) if the text is written in English.

Bonus point if the utility is able to give a confidence level for its predictions.
4.4 Exercise 4: Face recognition

Build a classifier that recognize person on faces pictures from the Labeled Faces in the Wild dataset.

ipython command line:

%run workspace/exercise_04_face_recognition.py data/data/labeled_faces_wild/lfw_preprocessed/
To conclude this session here are some practical hints for selecting the right algorithm when facing a practical problem.

- If the data is high dimensional and sparse (text data), most of the time linear classifiers with a bit of regularization will work well.

- If the data is dense, low to medium dimensional: try to further reduce the dimensionality with PCA for instance and try both linear and non linear models (e.g. SVC with RBF kernel).

- **SVC** with gaussian RBF kernel and **KMeans** clustering can benefit a lot from data normalization with (PCA or RandomizedPCA with whiten=True). Try various values for n_components with grid search to be sure no to truncate the data too hard.

- There is no free lunch: the best algorithm is data dependant. If you try many different models, reserve a held out evaluation set that is not used during the model selection process.
# FINDING HELP

## 6.1 The project mailing list

If you encounter a bug with scikit-learn or something that needs clarification in the docstring or the online documentation, please feel free to ask on the Mailing List.

## 6.2 Q&A communities with Machine Learning practitioners

**Metaoptimize/QA** A forum for Machine Learning, Natural Language Processing and other Data Analytics discussions (similar to what Stackoverflow is for developers): [http://metaoptimize.com/qa](http://metaoptimize.com/qa)

A good starting point is the discussion on [good freely available textbooks on machine learning](http://metaoptimize.com/qa)

**Quora.com** Quora as a topic for Machine Learning related questions that also features some interesting discussions: [http://quora.com/Machine-Learning](http://quora.com/Machine-Learning)

Have a look at the best questions section, eg: What are some good resources for learning about machine learning.