Astronomy with scikit-learn

Release scipy 2012

Jacob VanderPlas

http://astroML.github.com/sklearn$_{t}utorial$

July 15, 2012

Contents

This tutorial offers a brief introduction to the fields of machine learning and statistical data analysis, and their application to several problems in the field of astronomy. These learning tasks are enabled by the tools available in the open-source package scikit-learn[a].

scikit-learn[b] is a Python module integrating classic machine learning algorithms in the tightly-knit world of scientific Python packages (numpy[c], scipy[d], matplotlib[e]). 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**.

Many of the examples and exercises in this tutorial require the ipython notebook[f], a tool which provides an intuitive web-based interactive environment for scientific python. Some of the material in the notebooks is duplicated in the following pages, but ipython notebook is required for some parts. For information on how to download the associated notebooks, see the *Tutorial Setup and Installation* (page **??**) page.

[a]http://www.scikit-learn.org
[b]http://www.scikit-learn.org
[c]http://numpy.scipy.org
[d]http://www.scipy.org
[e]http://matplotlib.sourceforge.net
[f]http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

Note: This document is meant to be used with **scikit-learn version 0.11+**. Find the latest version here.

CHAPTER 1

Tutorial Setup and Installation

Objectives

At the end of this section, you will

1. Have scikit-learn and all the prerequisites and dependencies for this tutorial installed on your machine.
2. Download the source files and data required for this tutorial

1.1 Python Prerequisites

This tutorial is based on scikit-learn, which has the following dependencies:

- numpy[1] : this is a python module which has powerful tools for the creation and manipulation of arrays. It is the foundation of most scientific computing packages in python
- scipy[2] : this is a python module which builds on numpy and provides fast implementations of many basic scientific algorithms.
- matplotlib[3] : this is a powerful package for generating plots, figures, and diagrams. Our main form of visual interaction with data and results depends on matplotlib.

We will also make extensive use of iPython[4], an interactive python interpreter. In particular, much of the interactive material requires ipython notebook[5] functionality, which was introduced in ipython version 0.12.

1.1.1 Installing scikit-learn and Dependencies

Please refer to the install page[6] for per-system instructions on installing scikit-learn. In addition to `numpy`, `scipy`, and `scikit-learn`, this tutorial will assume that you have `matplotlib` and `ipython` installed as well.

- Under **Debian or Ubuntu Linux** you should use:

[1]http://numpy.scipy.org
[2]http://www.scipy.org
[3]http://matplotlib.sourceforge.net/
[4]http://ipython.org
[5]http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html
[6]http://www.scikit-learn.org/stable/install.html#installing-an-official-release

```
% sudo apt-get install build-essential python-dev python-numpy \
  python-numpy-dev python-scipy libatlas-dev g++ python-matplotlib \
  ipython
```

- Under **MacOSX** you should probably use a scientific python distribution such as Scipy Superpack[7]
- Under **Windows** the Python(x,y)[8] is probably your best bet to get a working numpy / scipy environment up and running.
- Power-users may wish to install bleeding edge versions of these packages from the source. The source can be downloaded using `git` from the packages' respective GitHub[9] repositories.

Alternatively under Windows and MaxOSX you can use the EPD[10] (Enthought Python Distribution) which is a (non-open source) packaging of the scientific python stack.

Note: that to use ipython notebook[11], you must install `ipython` version 0.12 and several other dependencies. Refer to the ipython documentation for details.

1.2 Tutorial Files

The source code for the example files in the following pages is best accessed through cloning the scikit-learn repository using git[12]. Once `git` is installed the command to accomplish this is:

```
% git clone https://github.com/astroML/sklearn_tutorial
```

This creates a directory called `sklearn_tutorial` and copies all the source files of this tutorial. Most of the relevant files are in the `sklearn_tutorial/doc` sub-directory. In what follows, this directory will be named `$TUTORIAL_HOME`. It should contain the following folders:

- `data` - folder to put the datasets used during the tutorial
- `skeletons` - sample incomplete scripts for the exercices (these should be used only if `ipython notebook` is unavailable)
- `solutions` - solutions of the exercices (these should be used only if `ipython notebook` is unavailable)
- `notebooks` - ipython notebooks which provide an interactive interface to parts of this tutorial. These contain material which is not in the skeletons and solutions.

If you are not going to use ipython notebook to run the examples, you can copy the skeletons into a new folder named `workspace` where you will edit your own files for the exercices while keeping the original skeletons intact:

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

1.3 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). This will download a dataset to the current directory. This tutorial has three such datasets; they will be used in the examples and exercises later on.

[7] http://fonnesbeck.github.com/ScipySuperpack/
[8] http://www.pythonxy.com/
[9] http://www.github.com
[10] https://www.enthought.com/products/epd.php
[11] http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html
[12] http://git-scm.com/

To get all three datasets, run the following:

```
% cd $TUTORIAL_HOME/data/sdss_colors
% python fetch_data.py

% cd $TUTORIAL_HOME/data/sdss_photoz
% python fetch_data.py

% cd $TUTORIAL_HOME/data/sdss_spectra
% python fetch_data.py
```

CHAPTER 2

Machine Learning 101: General Concepts

Objectives

By the end of this section you will

1. Know how to extract features from real-world data in order to perform machine learning tasks.
2. Know the basic categories of *supervised learning*, including *classification* and *regression* problems.
3. Know the basic categories of *unsupervised learning*, including dimensionality reduction and clustering.
4. Understand the distinction between linearly separable and non-linearly separable data.

In addition, you will know several tools within scikit-learn which can be used to accomplish the above tasks.

In this section we will begin to explore the basic principles of machine learning. Machine Learning is about building **programs with tunable parameters** (typically an array of floating point values) that are adjusted automatically so as to improve their behavior by **adapting to previously seen data**.

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

A very simple example of a machine learning task can be seen in the following figure: it shows a collection of two-dimensional data, colored according to two different class labels. A classification algorithm is used to draw a dividing boundary between the two clusters of points:

As with all figures in this tutorial, the above image has a hyper-link to the python source code which is used to generate it.

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.

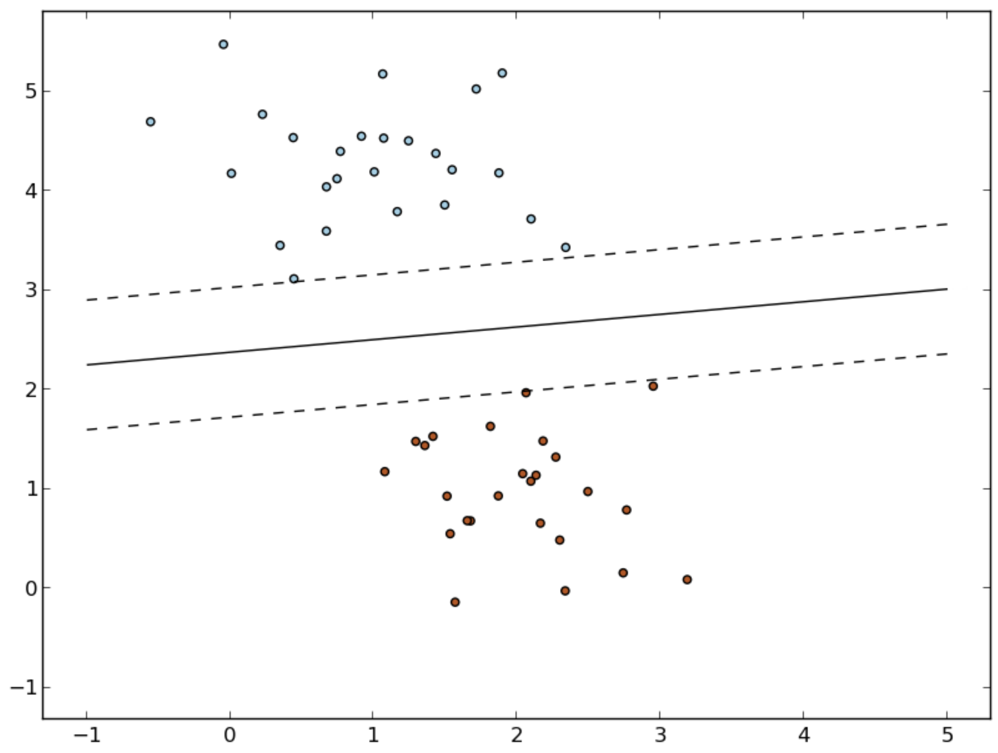


Figure 2.1: Example Linear Decision Boundary

`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 may use `scipy.sparse` matrices instead of `numpy` arrays so as to make the data fit in memory.

2.1.1 A simple example: the iris dataset

Note: The information in this section is available in an interactive notebook `01_datasets.ipynb`, which can be viewed using iPython notebook[1].

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

[1] http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

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 sklearn.datasets import load_iris
>>> iris = load_iris()
```

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

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

>>> n_samples
150

>>> n_features
4

>>> 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:

```
>>> 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, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

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

```
>>> 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:

```
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#blue (1.0 or 0.0)
6. color#red (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 unstructed 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 Bag of Words[2]

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** aranged in a code book. The code book may have been previously extracted from the same dataset using an unsupervised learning algorithm 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 features of the individual areas into a single array.

Sounds Same strategy as for images within 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, Unsupervised Learning, and `scikit-learn` syntax

Machine learning can be broken into two broad regimes: supervised learning and unsupervised learning. We'll introduce these concepts here, and discuss them in more detail below.

In **Supervised Learning**, we have a dataset consisting of both *features* and *labels*. The task is to construct an estimator which is able to predict the label of an object given the set of features. A relatively simple example is predicting the species of iris given a set of measurements of its flower. This is a relatively simple task. Some more complicated examples are:

[2]http://scikit-learn.org/dev/modules/feature_extraction.html#text-feature-extraction

- given a multicolor image of an object through a telescope, determine whether that object is a star, a quasar, or a galaxy.
- given a photograph of a person, identify the person in the photo.
- given a list of movies a person has watched and their personal rating of the movie, recommend a list of movies they would like (A famous example is the Netflix Prize[3]).

What these tasks have in common is that there is one or more unknown quantities associated with the object which needs to be determined from other observed quantities. Supervised learning is further broken down into two categories, *classification* and *regression*. In classification, the label is discrete, while in regression, the label is continuous. For example, in astronomy, the task of determining whether an object is a star, a galaxy, or a quasar is a classification problem: the label is from three distinct categories. On the other hand, we might wish to determine the age of an object based on such observations: this would be a regression problem: the label (age) is a continuous quantity.

Unsupervised Learning addresses a different sort of problem. Here the data has no labels, and we are interested in finding similarities between the objects in question. In a sense, you can think of unsupervised learning as a means of discovering labels from the data itself. Unsupervised learning comprises tasks such as dimensionality reduction, clustering, and density estimation. For example, in the iris data discussed above, we can used unsupervised methods to determine combinations of the measurements which best display the structure of the data. As we'll see below, such a projection of the data can be used to visualize the four-dimensional dataset in two dimensions. Some more involved unsupervised learning problems are:

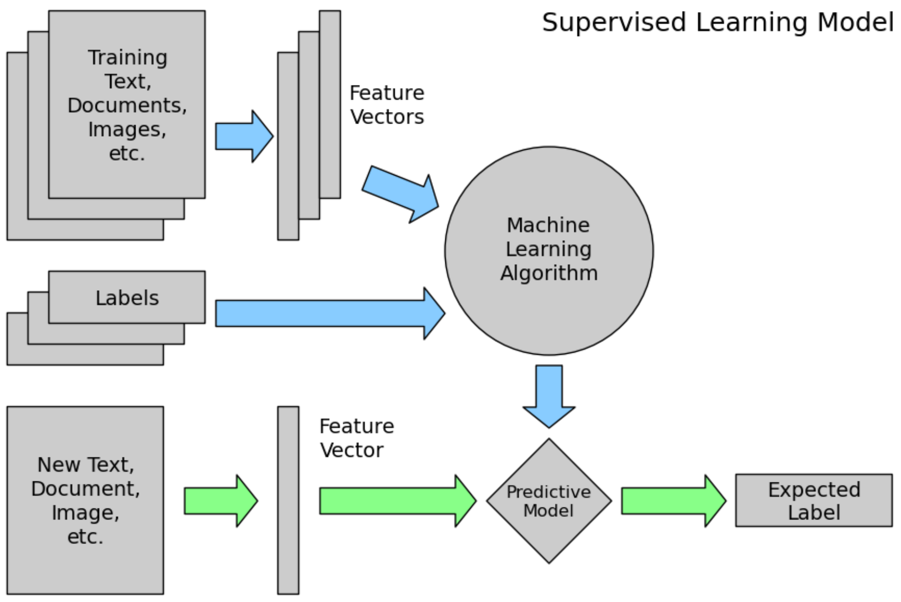
- given detailed observations of distant galaxies, determine which features or combinations of features are most important in distinguishing between galaxies.
- given a mixture of two sound sources (for example, a person talking over some music), separate the two (this is called the blind source separation[4] problem).
- given a video, isolate a moving object and categorize in relation to other moving objects which have been seen.

`scikit-learn` strives to have a uniform interface across all methods, and we'll see examples of these below. Given a `scikit-learn` estimator object named `model`, the following methods are available:

- **Available in all Estimators**
  - `model.fit()` : fit training data. For supervised learning applications, this accepts two arguments: the data `X` and the labels `y` (e.g. `model.fit(X, y)`). For unsupervised learning applications, this accepts only a single argument, the data `X` (e.g. `model.fit(X)`).
- **Available in supervised estimators**
  - `model.predict()` : given a trained model, predict the label of a new set of data. This method accepts one argument, the new data `X_new` (e.g. `model.predict(X_new)`), and returns the learned label for each object in the array.
  - `model.predict_proba()` : For classification problems, some estimators also provide this method, which returns the probability that a new observation has each categorical label. In this case, the label with the highest probability is returned by `model.predict()`.
- **Available in unsupervised estimators**
  - `model.transform()` : given an unsupervised model, transform new data into the new basis. This also accepts one argument `X_new`, and returns the new representation of the data based on the unsupervised model.
  - `model.fit_transform()` : some estimators implement this method, which performs a `fit` and a `transform` on the same input data.

[3]http://en.wikipedia.org/wiki/Netflix_prize
[4]http://en.wikipedia.org/wiki/Blind_signal_separation

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


Supervised Learning Model
Training Text, Documents, Images, etc.
Feature Vectors
Machine Learning Algorithm
Labels
New Text, Document, Image, etc.
Feature Vector
Predictive Model
Expected Label

As mentioned above, 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 during training. In `scikit-learn` this array is often noted `y` and has generally the shape `(n_samples,)`. After training, the fitted model 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**.

2.3.1 Classification

Classification is the task of predicting the value of a categorical variable given some input variables (a.k.a. the features or "predictors"). This section includes a first exploration of classification with scikit-learn. We'll explore a detailed example of classification with astronomical data in *Classification: Learning Labels of Astronomical Sources* (page **??**).

A first classifier example with `scikit-learn`

Note: The information in this section is available in an interactive notebook `02_iris_classification.ipynb`, which can be viewed using iPython notebook[5].

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:

[5] http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

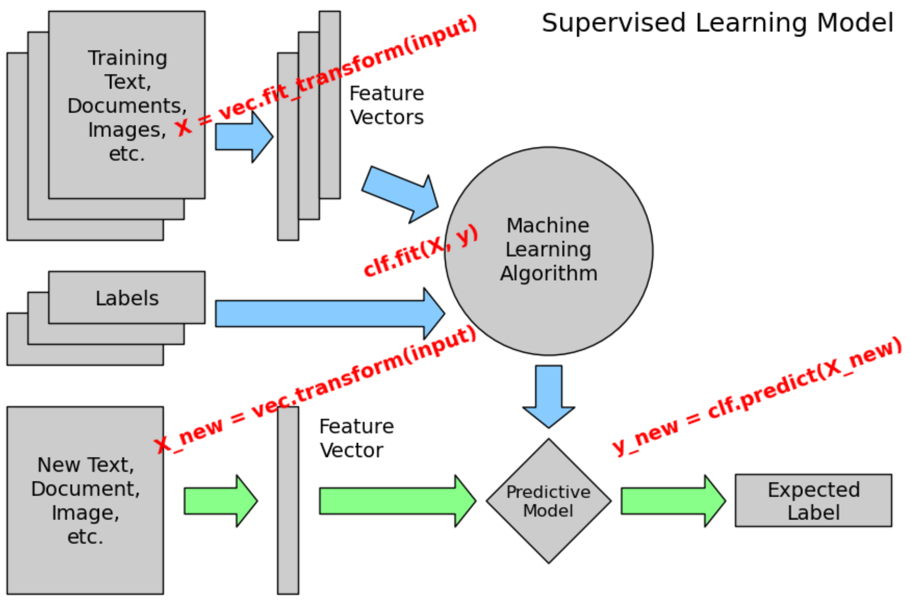

Supervised Learning Model
Training Text, Documents, Images, etc.
X = vec.fit_transform(input)
Feature Vectors
Machine Learning Algorithm
clf.fit(X, y)
Labels
X_new = vec.transform(input)
New Text, Document, Image, etc.
Feature Vector
Predictive Model
y_new = clf.predict(X_new)
Expected Label

Figure 2.2: Overview of supervised Learning with scikit-learn

```
>>> 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:

```
>>> from sklearn.svm import LinearSVC
>>> clf = LinearSVC()
```

Note: Whenever you import a scikit-learn class or function for 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 hyperparameters). Those hyperparameters can be supplied by the user in the constructor of the model. We will explain later how to choose a good combination using either simple empirical rules or data driven selection:

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

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

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

>>> clf.coef_
array([[ 0.18...,  0.45..., -0.80..., -0.45...],
       [ 0.05..., -0.89...,  0.40..., -0.93...],
       [-0.85..., -0.98...,  1.38...,  1.86...]])

>>> clf.intercept_
array([ 0.10...,  1.67..., -1.70...])
```

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 simple sample that looks like the first sample of the iris dataset:

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

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

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 extraction that is not used in the case of the iris data (it already comes as vectors of features):

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

```
>>> from sklearn.linear_model import LogisticRegression
>>> clf2 = LogisticRegression().fit(X, y)
>>> clf2
LogisticRegression(C=1.0, dual=False, fit_intercept=True, intercept_scaling=1,
          penalty='l2', tol=0.0001)
>>> clf2.predict_proba(X_new)
array([[  9.07512928e-01,   9.24770379e-02,   1.00343962e-05]])
```

This means that the model estimates that the sample in `X_new` has:

- 90% likelyhood to belong to the 'setosa' class
- 9% likelyhood to belong to the 'versicolor' class
- 1% likelyhood to belong to the 'virginica' class

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

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

Notable implementations of classifiers

`sklearn.linear_model.LogisticRegression`[6]

Regularized Logistic Regression based on `liblinear`

`sklearn.svm.LinearSVC`[7]

Support Vector Machines without kernels based on `liblinear`

`sklearn.svm.SVC`[8]

Support Vector Machines with kernels based on `libsvm`

`sklearn.linear_model.SGDClassifier`[9]

Regularized linear models (SVM or logistic regression) using a Stochastic Gradient Descent algorithm written in `Cython`

`sklearn.neighbors.NeighborsClassifier`

[6]http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression
[7]http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC
[8]http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
[9]http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html#sklearn.linear_model.SGDClassifier

k-Nearest Neighbors classifier based on the ball tree datastructure for low dimensional data and brute force search for high dimensional data

`sklearn.naive_bayes.GaussianNB`[10]

Gaussian Naive Bayes model. This is an unsophisticated model which can be trained very quickly. It is often used to obtain baseline results before moving to a more sophisticated classifier.

Sample application of classifiers

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

Task	Predicted outcomes
E-mail classification	Spam, normal, priority mail
Language identification in text documents	en, es, de, fr, ja, zh, ar, ru...
News articles categorization	Business, technology, sports...
Sentiment analysis in customer feedback	Negative, neutral, positive
Face verification in pictures	Same / different person
Speaker verification in voice recordings	Same / different person
Astronomical Sources	Object type or class

2.3.2 Regression

Regression is the task of predicting 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"). We'll explore a detailed example of regression with astronomical data in *Regression: Photometric Redshifts of Galaxies* (page **??**).

Some notable implementations of regression models in `scikit-learn` include:

`sklearn.linear_model.Ridge`[11]

L2-regularized least squares linear model

`sklearn.linear_model.ElasticNet`[12]

L1+L2-regularized least squares linear model trained using Coordinate Descent

`sklearn.linear_model.LassoLARS`

L1-regularized least squares linear model trained with Least Angle Regression

`sklearn.linear_model.SGDRegressor`[13]

L1+L2-regularized least squares linear model trained using Stochastic Gradient Descent

`sklearn.linear_model.ARDRegression`[14]

Bayesian Automated Relevance Determination regression

`sklearn.svm.SVR`[15]

Non-linear regression using Support Vector Machines (wrapper for `libsvm`)

`sklearn.ensemble.RandomForestRegressor`[16]

[10]http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB
[11]http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html#sklearn.linear_model.Ridge
[12]http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html#sklearn.linear_model.ElasticNet
[13]http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html#sklearn.linear_model.SGDRegressor
[14]http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ARDRegression.html#sklearn.linear_model.ARDRegression
[15]http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR
[16]http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html#sklearn.ensemble.RandomForestRegressor

An ensemble method which constructs multiple decision trees from subsets of the data.

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

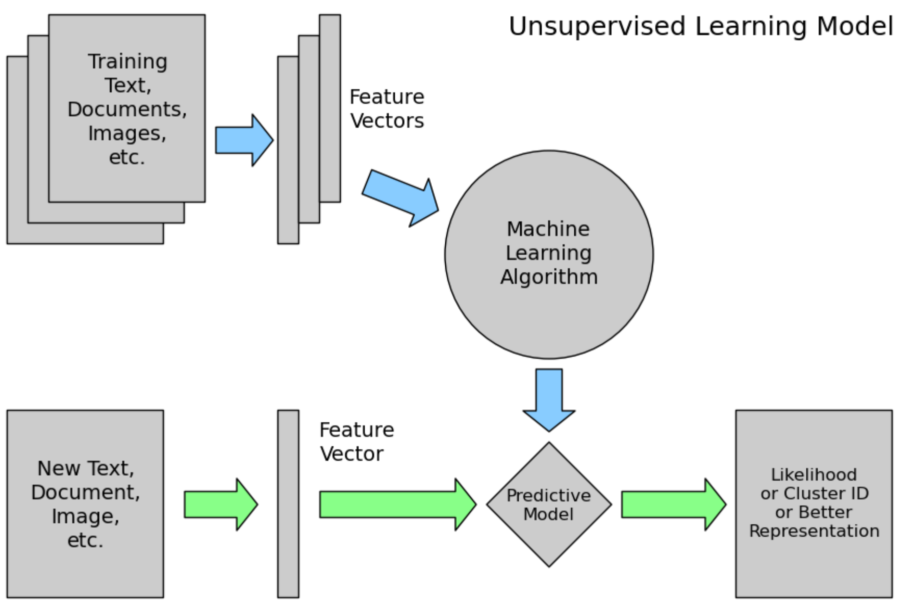

Unsupervised Learning Model
Training Text, Documents, Images, etc.
Feature Vectors
Machine Learning Algorithm
New Text, Document, Image, etc.
Feature Vector
Predictive Model
Likelihood or Cluster ID or Better Representation

Figure 2.3: Unsupervised Learning overview

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.

2.4.1 Dimensionality Reduction and visualization

Dimensionality reduction is the task of deriving 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

Note: The information in this section is available in an interactive notebook `03_iris_dimensionality.ipynb`, which can be viewed using iPython notebook[17].

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[18] of the matrix `X` so as to project the data onto a base of the top singular vectors.

[17]http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html
[18]http://en.wikipedia.org/wiki/Singular_value_decomposition#Truncated_SVD

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

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

Once fitted, the `pca` model exposes the singular vectors in the `components_` attribute:

```
>>> pca.components_
array([[ 0.17..., -0.04...,  0.41...,  0.17...],
       [-1.33..., -1.48...,  0.35...,  0.15...]])

>>> pca.explained_variance_ratio_
array([ 0.92...,  0.05...])

>>> pca.explained_variance_ratio_.sum()
0.97...
```

Let us project the iris dataset along those first 2 dimensions:

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

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

```
>>> import numpy as np
>>> np.round(X_pca.mean(axis=0), decimals=5)
array([-0.,  0.])

>>> np.round(X_pca.std(axis=0), decimals=5)
array([ 1.,  1.])
```

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

```
>>> import numpy as np
>>> np.round(np.corrcoef(X_pca.T), decimals=5)
array([[ 1., -0.],
       [-0.,  1.]])
```

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

```
>>> 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()
...     pl.show()
...
```

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

Note that this projection was determined *without* any information about the labels (represented by the colors): this is the sense in which the learning is unsupervised. Nevertheless, we see that the projection gives us insight into the distribution of the different flowers in parameter space: notably, *iris setosa* is much more distinct than the other two species.

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 that a few thousands).

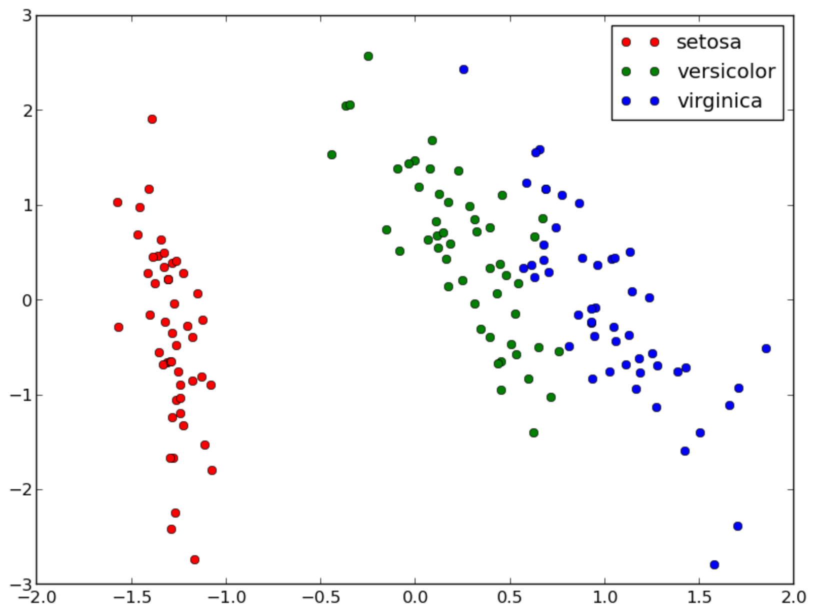


Figure 2.4: 2D PCA projection of the iris dataset

If you are interested in a number of components that is much smaller than both `n_samples` and `n_features`, consider using `sklearn.decomposition.RandomizedPCA`[19] 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 methods 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.

Note: `scikit-learn` also features an implementation of Independant Component Analysis (ICA) and several manifold learning methods (See *Exercise 3: Dimensionality Reduction of Spectra* (page **??**))

2.4.2 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 example, 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):

```
>>> from sklearn.cluster import KMeans
>>> from numpy.random import RandomState
>>> rng = RandomState(42)

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

[19] http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.RandomizedPCA.html#sklearn.decomposition.RandomizedPCA

```
>>> np.round(kmeans.cluster_centers_, decimals=2)
array([[ 1.02, -0.71],
       [ 0.33,  0.89],
       [-1.29, -0.44]])

>>> 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:

```
plot_2D(X_pca, kmeans.labels_, ["c0", "c1", "c2"])
```

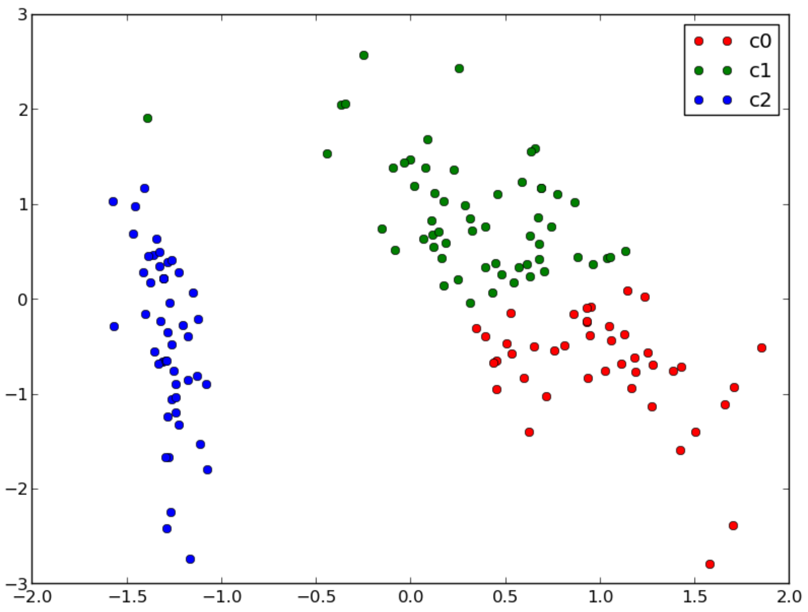


Figure 2.5: KMeans cluster assignements on 2D PCA iris data

Exercise

Repeat the clustering algorithm from above, but fit the clusters to the full dataset `X` rather than the projection `X_pca`. Do the labels computed this way better match the true labels?

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 clustered to have the shape `(n_samples, n_features)`:

`sklearn.cluster.KMeans`[20]

The simplest, yet effective clustering algorithm. Needs to be provided with the number of clusters in advance, and assumes that the data is normalized as input (but use a PCA model as preprocessor).

[20]http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans

`sklearn.cluster.MeanShift`[21]

Can find better looking clusters than KMeans but is not scalable to high number of samples.

`sklearn.cluster.DBSCAN`[22] Can detect irregularly shaped clusters based on density, i.e. sparse regions in the input space are likely to become inter-cluster boundaries. Can also detect outliers (samples that are not part of a cluster).

`sklearn.manifold.LocallyLinearEmbedding`[23]

Locally Linear Embedding is a nonlinear neighbors-based manifold learning technique. The scikit-learn implementation makes available several variants to the basic algorithm.

`sklearn.manifold.Isomap`[24]

Isomap is another neighbors-based manifold learning method that can find nonlinear projections of data.

Other clustering algorithms do not work with a data array of shape `(n_samples, n_features)` but directly with a precomputed affinity matrix of shape `(n_samples, n_samples)`:

`sklearn.cluster.AffinityPropagation`[25]

Clustering algorithm based on message passing between data points.

`sklearn.cluster.SpectralClustering`[26]

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.

`sklearn.cluster.Ward`[27]

`Ward` implements hierarchical clustering based on the Ward algorithm, a variance-minimizing approach. At each step, it minimizes the sum of squared differences within all clusters (inertia criterion).

`DBSCAN` can work with either an array of samples or an affinity matrix.

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

[21] http://scikit-learn.org/stable/modules/generated/sklearn.cluster.MeanShift.html#sklearn.cluster.MeanShift
[23] http://scikit-learn.org/stable/modules/generated/sklearn.manifold.LocallyLinearEmbedding.html#sklearn.manifold.LocallyLinearEmbedding
[24] http://scikit-learn.org/stable/modules/generated/sklearn.manifold.Isomap.html#sklearn.manifold.Isomap
[25] http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AffinityPropagation.html#sklearn.cluster.AffinityPropagation
[26] http://scikit-learn.org/stable/modules/generated/sklearn.cluster.SpectralClustering.html#sklearn.cluster.SpectralClustering
[27] http://scikit-learn.org/stable/modules/generated/sklearn.cluster.Ward.html#sklearn.cluster.Ward

2.5 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. (if you don't have the scikit-learn source code locally installed, you can find the script here):

```
% python $SKL_HOME/examples/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 accurracy 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).

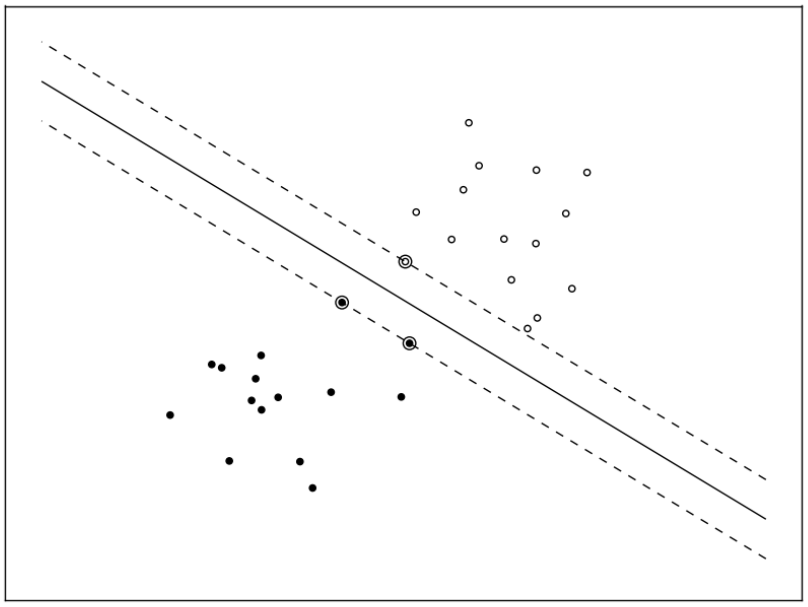


Figure 2.6: Linear Support Vector Machine trained to perfectly separate 2 sets of data points labeled as white and black in a 2D space.

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?

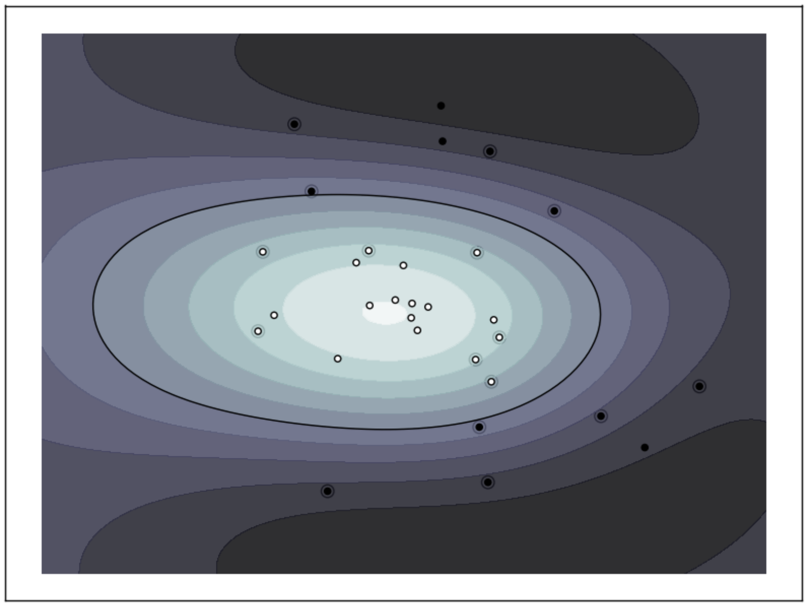


Figure 2.7: Support Vector Machine with gaussian kernel trained to separate 2 sets of data points labeled as white and black in a 2D space. This dataset would not have been seperated by a simple linear model.

Exercise

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

Note: the higher the dimension of the feature space, the more likely the data is linearly separable: for instance this is often the case for text classification tasks.

2.6 Hyperparameters, training set, test set and overfitting

The above SVM example displays an example of *hyperparameters*, which are model parameters set before the training process. For example, when using an RBF model, we choose the kernel coefficient `gamma` before fitting the data. We must be able to then evaluate the goodness-of-fit of our model given this choice of hyperparameter.

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

> If you do this, **you are doing it wrong!**

2.6.1 The overfitting issue

Evaluating the quality of the model on the data used to fit the model can lead to *overfitting*. Consider the following dataset, and three fits to the data (we'll explore this example in more detail in the *next section* (page **??**)).

Evaluating the $d = 6$ model using the training data might lead you to believe the model is very good, when in fact it does not do a good job of representing the data. 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:

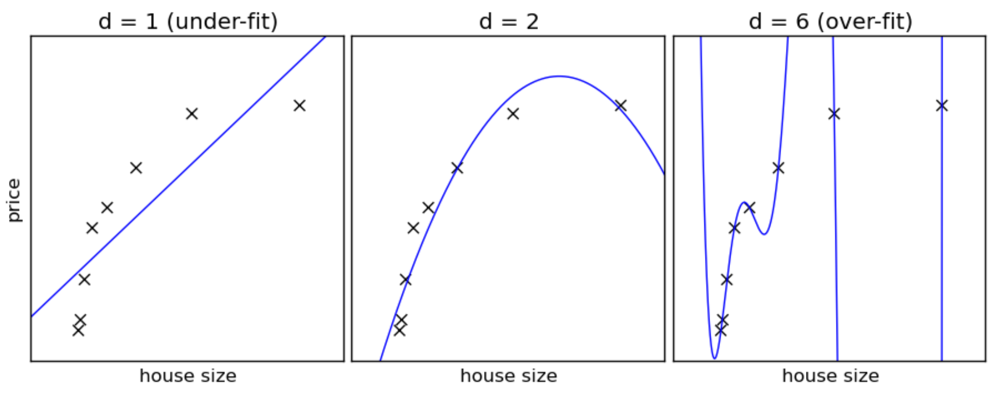


Figure 2.8: Examples of over-fitting and under-fitting a two-dimensional dataset.

- the predictive accurracy on the data used for training can be excellent (sometimes 100%)
- however, the models do little better than random prediction when facing new data that was not part of the training set

If you evaluate your model on your training data you won't be able to tell whether your model is overfitting or not.

2.6.2 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)

An even better option when experimenting with classifiers is to divide the data into three sets: training, testing and holdout. You can then optimize your features, settings and algorithms for the testing set until they seem good enough, and finally test on the holdout set (perhaps after adding the test set to the training set).

When the amount of labeled data available is small, it may not be feasible to construct training and test sets. In that case, you can choose to use **k-fold cross validation**: divide the dataset into $k = 10$ parts of (roughly) equal size, then for each of these ten parts, train the classifier on the other nine and test on the held-out part.

2.6.3 Measuring classification performance on a test set

Note: The information in this section is available in an interactive notebook `05_iris_crossval.ipynb`, which can be viewed using iPython notebook[28].

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:

[28]http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

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

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

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

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

```
>>> split = (n_samples * 2) / 3

>>> X_train, X_test = X[:split], X[split:]
>>> y_train, y_test = y[:split], y[split:]

>>> X_train.shape
(100, 4)

>>> X_test.shape
(50, 4)

>>> y_train.shape
(100,)

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

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

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

To evaluate its quality we can compute the average number of correct classifications on the test set:

```
>>> np.mean(clf.predict(X_test) == y_test)
1.0
```

This shows that the model has a predictive accurracy of 100% which means that the classification model was perfectly capable of 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 in the following chapter.

In the *next section* (page **??**), we will explore in more detail the bias-variance tradeoff and the practical use of machine learning techniques.

2.7 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 train a model: split train / test data:
    - use training set for model selection and fitting
    - use test set for model evaluation
    - use cross-validation when your dataset is small
- 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

CHAPTER 3

Machine Learning 102: Practical Advice

Objectives

By the end of this section you will

1. Be able to describe the *hyperparameters* of a model, and how *cross-validation* can be used to estimate them.
2. Be able to describe the concepts of *bias* & *variance*, and *overfitting* & *underfitting*
3. Be able to use *learning curves* to recognize when a model has high bias or high variance, and choose the corect course of action.

Note: The information in this section is available in an interactive notebook `06_learning_curves.ipynb`, which can be viewed using iPython notebook[1].

In practice, much of the task of machine learning involves selecting algorithms, parameters, and sets of data to optimize the results of the method. All of these things can affect the quality of the results, but it's not always clear which is best. For example, if your results have an error that's larger than you hoped, you might imagine that increasing the training set size will always lead to better results. But this is not the case! Below, we'll explore the reasons for this.

Note: much of the material in this section was adapted from Andrew Ng's excellent set of machine learning video lectures. See http://www.ml-class.org.

In this section we'll work with an extremely simple learning model: polynomial regression. This simply fits a polynomial of degree d to the data: if $d = 1$, then it is simple linear regression. Polynomial regression can be done with the functions `polyfit` and `polyval`, available in `numpy`. For example:

```
>>> import numpy as np
>>> np.random.seed(42)
>>> x = np.random.random(20)
>>> y = np.sin(2 * x)
>>> p = np.polyfit(x, y, 1)  # fit a 1st-degree polynomial (i.e. a line) to the data
>>> print p
[ 0.97896174  0.20367395]
>>> x_new = np.random.random(3)
>>> y_new = np.polyval(p, x_new)  # evaluate the polynomial at x_new
```

[1] http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

```
>>> print abs(np.sin(x_new) - y_new)
[ 0.22826933  0.20119119  0.20166572]
```

Using a 1st-degree polynomial fit (that is, fitting a straight line to `x` and `y`), we predicted the value of `y` for a new input. This prediction has an absolute error of about 0.2 for the few test points which we tried. We can visualize the fit with the following function:

```
>>> import pylab as pl
>>> def plot_fit(x, y, p):
...     xfit = np.linspace(0, 1, 1000)
...     yfit = np.polyval(p, xfit)
...     pl.scatter(x, y, c='k')
...     pl.plot(xfit, yfit)
...     pl.xlabel('x')
...     pl.ylabel('y')
```

Calling `plot_fit` with the `x`, `y`, and `p` values from above produces the following figure:

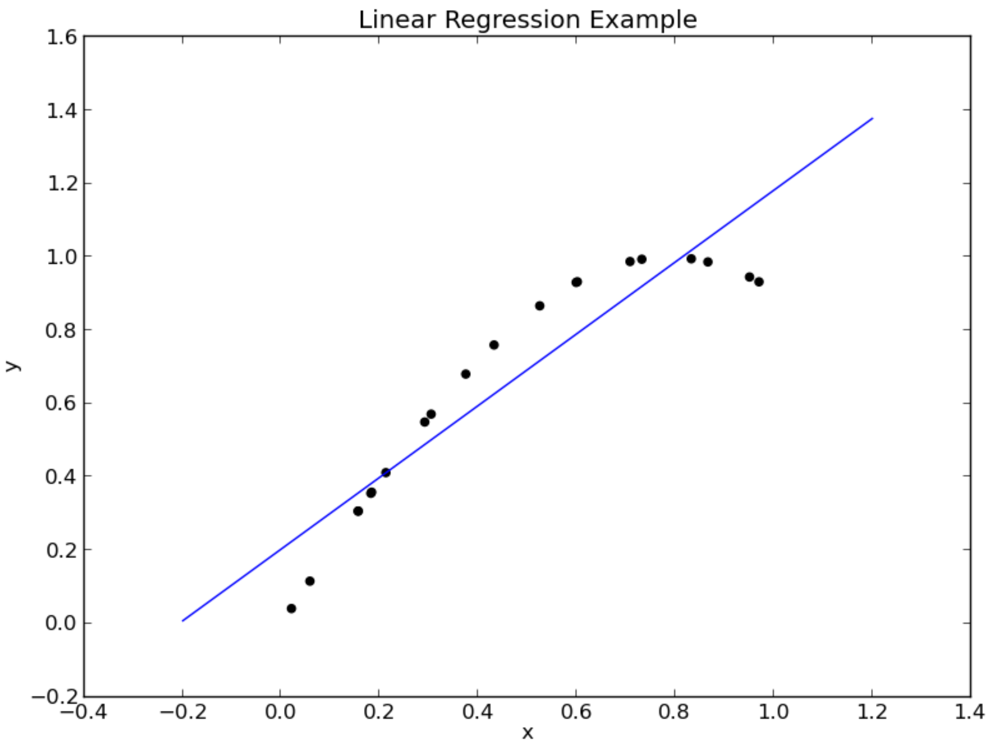


Figure 3.1: Best-fit linear regression to sinusoidal data.

When the error of predicted results is larger than desired, there are a few courses of action that can be taken:

1. Increase the number of training points *N*. This might give us a training set with more coverage, and lead to greater accuracy.
2. Increase the degree *d* of the polynomial. This might allow us to more closely fit the training data, and lead to a better result
3. Add more features. If we were to, for example, perform a linear regression using x, $\sqrt{x}$, x^{-1}, or other functions, we might hit on a functional form which can better be mapped to the value of *y*.

The best course to take will vary from situation to situation, and from problem to problem. In this situation, number

2 and 3 may be useful, but number 1 will certainly not help: our model does not intrinsically fit the data very well. In machine learning terms, we say that it has high *bias* and that the data is *under-fit*. The ability to quickly figure out how to tune and improve your model is what separates good machine learning practitioners from the bad ones. In this section we'll discuss some tools that can help determine which course is most likely to lead to good results.

3.1 Bias, Variance, Over-fitting, and Under-fitting

We'll work with a simple example. Imagine that you would like to build an algorithm which will predict the price of a house given its size. Naively, we'd expect that the cost of a house grows as the size increases, but there are many other factors which can contribute. Imagine we approach this problem with the polynomial regression discussed above. We can tune the degree d to try to get the best fit.

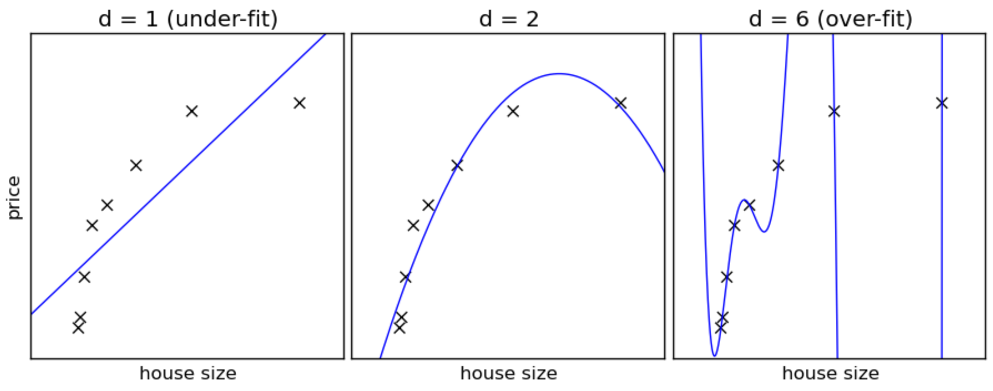


Figure 3.2: Polynomials of various degrees. $d = 1$ under-fits the data, while $d = 6$ over-fits the data.

In the above figure, we see fits for three different values of d. For $d = 1$, the data is *under-fit*. This means that the model is too simplistic: no straight line will ever be a good fit to this data. In this case, we say that the model suffers from high *bias*. The model itself is biased, and this will be reflected in the fact that the data is poorly fit. At the other extreme, for $d = 6$ the data is *over-fit*. This means that the model has too many free parameters (6 in this case) which can be adjusted to perfectly fit the training data. If we add a new point to this plot, though, chances are it will be very far from the curve representing the degree-6 fit. In this case, we say that the model suffers from high *variance*. The reason for this label is that if any of the input points are varied slightly, it could result in an extremely different model.

In the middle, for $d = 2$, we have found a good mid-point. It fits the data fairly well, and does not suffer from the bias and variance problems seen in the figures on either side. What we would like is a way to quantitatively identify bias and variance, and optimize the *metaparameters* (in this case, the polynomial degree d) in order to determine the best algorithm. This can be done through a process called cross-validation.

3.2 Cross-Validation and Testing

In order to quantify the effects of bias and variance and construct the best possible estimator, we will split our training data into three parts: a *training set*, a *cross-validation set*, and a *test set*. As a general rule, the training set should be about 60% of the samples, and the cross-validation and test sets should be about 20% each.

The general idea is as follows. The model parameters (in our case, the coefficients of the polynomials) are learned using the training set as above. The error is evaluated on the cross-validation set, and the meta-parameters (in our case, the degree of the polynomial) are adjusted so that this cross-validation error is minimized. Finally, the labels are predicted for the test set. These labels are used to evaluate how well the algorithm can be expected to perform on unlabeled data.

Note: Why do we need both a cross-validation set and a test set? Many machine learning practitioners use the same set of data as both a cross-validation set and a test set. This is not the best approach, for the same reasons we outlined above. Just as the parameters can be over-fit to the training data, the meta-parameters can be over-fit to the cross-validation data. For this reason, the minimal cross-validation error tends to under-estimate the error expected on a new set of data.

The cross-validation error of our polynomial classifier can be visualized by plotting the error as a function of the polynomial degree *d*. This plot is shown in the following figure:

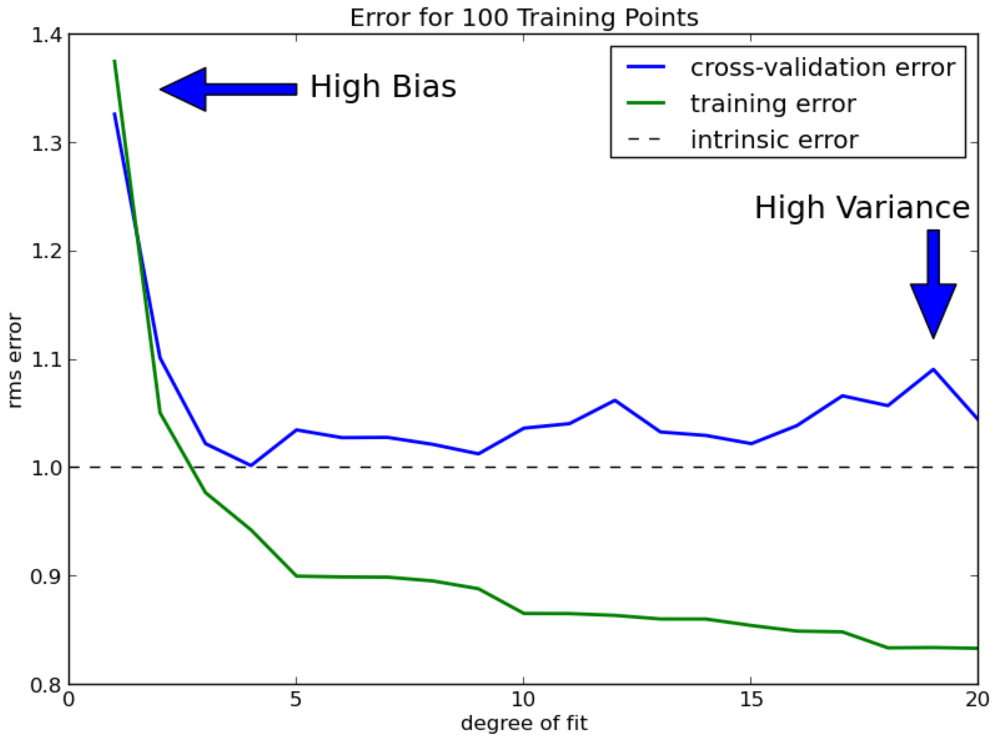


Figure 3.3: The training error and cross-validation error as a function of the polynomial degree *d*.

This figure compactly shows the reason that cross-validation is important. On the left side of the plot, we have very low-degree polynomial, which under-fits the data. This leads to a very high error for both the training set and the cross-validation set. On the far right side of the plot, we have a very high degree polynomial, which over-fits the data. This can be seen in the fact that the training error is very low, while the cross-validation error is very high. Plotted for comparison is the intrinsic error (this is the scatter artificially added to the data: click on the above image to see the source code). For this toy dataset, error = 1.0 is the best we can hope to attain. Choosing $d = 6$ in this case gets us very close to the optimal error.

The astute reader will realize that something is amiss here: in the above plot, $d = 6$ gives the best results. But in the previous plot, we found that $d = 6$ vastly over-fits the data. What's going on here? The difference is the number of training points used. In the previous example, there were only eight training points. In this example, we have 100. As a general rule of thumb, the more training points used, the more complicated model can be used. But how can you determine for a given model whether more training points will be helpful? A useful diagnostic for this are *learning curves*.

3.3 Learning Curves

A learning curve is a plot of the training and cross-validation error as a function of the number of training points. Note that when we train on a small subset of the training data, the training error is computed using this subset, not the full training set. These plots can give a quantitative view into how beneficial it will be to add training samples.

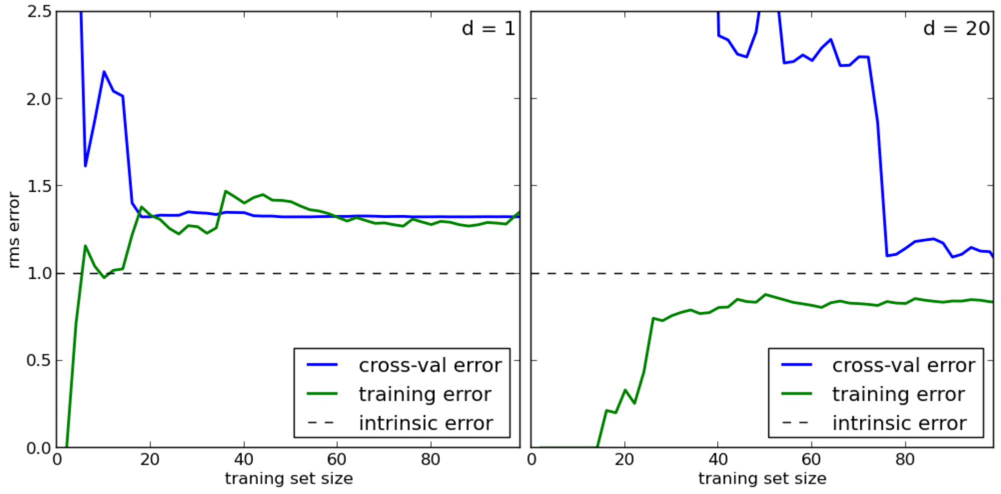


Figure 3.4: Learning Curves for a case of high bias (left, *d* = 2) and high variance (right, *d* = 20)

On the left plot, we have the learning curve for *d* = 1. From the above discussion, we know that *d* = 1 is a high-bias estimator which under-fits the data. This is indicated by the fact that both the training and cross-validation errors are very high. If this is the case, adding more training data will not help matters: both lines have converged to a relatively high error.

In the right plot, we have the learning curve for *d* = 20. From the above discussion, we know that *d* = 20 is a high-variance estimator which over-fits the data. This is indicated by the fact that the training error is much less than the cross-validation error. As we add more samples to this training set, the training error will continue to climb, while the cross-validation error will continue to decrease, until they meet in the middle. In this case, our intrinsic error is 1.0 (again, this is artificially set in the code: click on the image to browse the source code), and we can see that adding more data will allow the estimator to very closely match the best possible cross-validation error.

Note: With a degree-20 polynomial, we'd expect the training error to be identically zero for training set size $N <=$ 20. Why is this? It is because when the degrees of freedom are greater than the number of constraints, the problem should be perfectly solvable: a curve can be found which passes through every point (for example, imagine fitting a line to a single point. You'd be very surprised if you got anything but a perfect fit!) In the right-hand plot we see that this (correct) intuition fails in practice. The reason is due to floating-point precision: to perfectly fit these data points with a polynomial requires a fit that oscillates to extreme values in the space between the points (compare to the degree-6 polynomial above). The nature of our dataset means that this oscillation is outside machine precision, so that the resulting fit has a small residual.

3.4 Summary

We've seen above that an under-performing algorithm can be due to two possible situations: high bias (under-fitting) and high variance (over-fitting). In order to evaluate our algorithm, we set aside a portion of our training data for cross-validation. Using the technique of learning curves, we can train on progressively larger subsets of the data, evaluating the training error and cross-validation error to determine whether our algorithm has high variance or high bias. But what do we do with this information?

3.4.1 High Bias

If our algorithm shows high bias, the following actions might help:

1. **Add more features.** In our example of predicting home prices, it may be helpful to make use of information such as the neighborhood the house is in, the year the house was built, the size of the lot, etc. Adding these features to the training and test sets can improve a high-bias estimator
2. **Use a more sophisticated model.** Adding complexity to the model can help improve on bias. For a polynomial fit, this can be accomplished by increasing the degree d. Each learning technique has its own methods of adding complexity.
3. **Use fewer samples.** Though this will not improve the classification, a high-bias algorithm can attain nearly the same error with a smaller training sample. For algorithms which are computationally expensive, reducing the training sample size can lead to very large improvements in speed.
4. **Decrease regularization.** Regularization is a technique used to impose simplicity in some machine learning models, by adding a penalty term that depends on the characteristics of the parameters. If a model has high bias, decreasing the effect of regularization can lead to better results.

3.4.2 High Variance

If our algorithm shows high variance, the following actions might help:

1. **Use fewer features.** Using a feature selection technique may be useful, and decrease the over-fitting of the estimator.
2. **Use more training samples.** Adding training samples can reduce the effect of over-fitting, and lead to improvements in a high variance estimator.
3. **Increase Regularization.** Regularization is designed to prevent over-fitting. In a high-variance model, increasing regularization can lead to better results.

These choices become very important in real-world situations. For example, due to limited telescope time, astronomers must seek a balance between observing a large number of objects, and observing a large number of features for each object. Determining which is more important for a particular learning task can inform the observing strategy that the astronomer employs. In a later exercise, we will explore the use of learning curves for the photometric redshift problem.

CHAPTER 4

Classification: Learning Labels of Astronomical Sources

Modern astronomy is concerned with the study and characterization of distant objects such as stars[1], galaxies[2], or quasars[3]. Objects can often be very quickly characterized through detailed measurements of their optical spectrum[4]. A spectrum is a measure of the photon flux (that is, the number of photons per second) as a function of photon frequency or wavelength.

The above spectrum is that of the star Vega, the brightest star in the northern constellation Lyra. Its surface is at about 9600 degrees Kelvin, and its spectrum is roughly that of a 9600K black-body[5], with absorption due to molecules in its atmosphere. Because of the quantum mechanical properties of atoms, different atoms can absorb light at only specific, discrete wavelengths. Because of this fact, characteristic patterns in the spectrum of a distant star can be used to infer its chemical composition!

In the spectrum above, the deepest of these absorption spikes are due to the energy levels of Hydrogen. From examination of high-resolution spectra like this one, one can learn a lot about the physical processes at work in a distant astronomical source. Unfortunately, spectra like these are very time-consuming and expensive to obtain, especially for very faint objects. For this reason, astronomers have long observed objects through broad-band filters. For the u-band filter shown above, the flux is given by

$$f_u = \int_0^\infty f_u(\lambda) S_\nu(\lambda) \frac{d\lambda}{\lambda}$$

where $f_u(\lambda)$ is the filter transmission, and $S_\nu(\lambda)$ is the flux density of the spectrum at wavelength λ. For historical reasons, astronomers report the flux using the magnitude system, where the magnitude is defined by

$$u = -2.5 \log_{10} \left[\frac{f_u}{3631 Jy} \right]$$

The denominator is a normalization constant, measured in Janskys. To reduce the uncertainty associated with absolute calibration from telescope to telescope or from night to night, astronomers generally work in terms of the *color*,

[1] http://en.wikipedia.org/wiki/Star
[2] http://en.wikipedia.org/wiki/Galaxy
[3] http://en.wikipedia.org/wiki/Quasar
[4] http://en.wikipedia.org/wiki/Spectrum
[5] http://en.wikipedia.org/wiki/Black_body

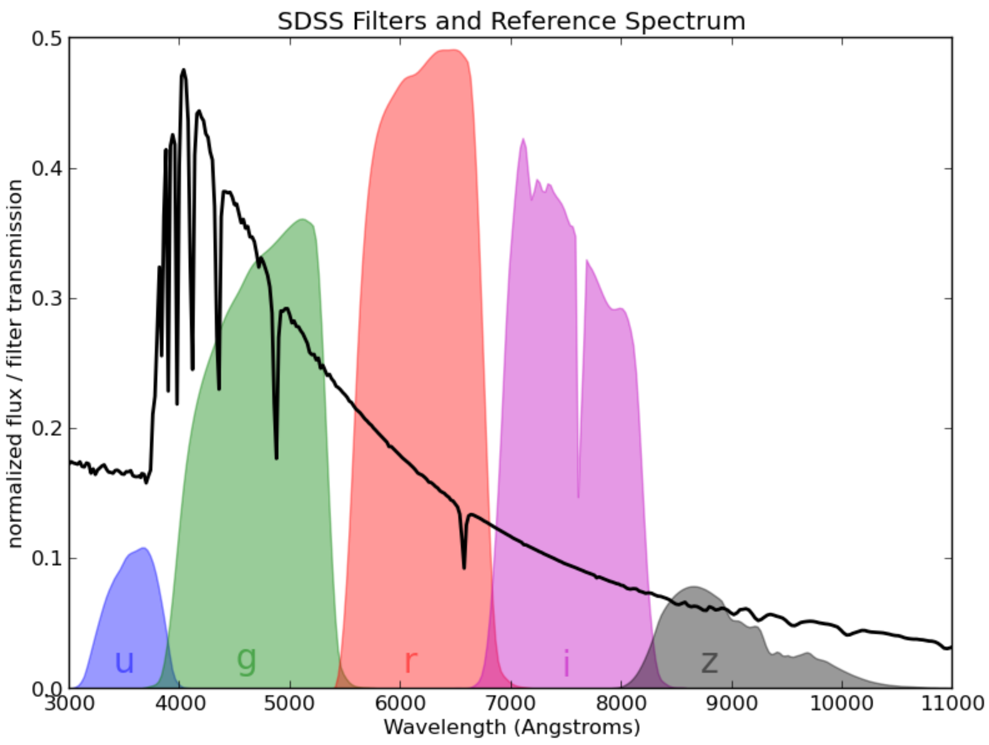


Figure 4.1: The spectrum of the star Vega (α-Lyr) with the five filters from the Sloan Digital Sky Survey (SDSS), which are denoted by the letters *u* (ultraviolet), *g* (green), *r* (red), *i* (infrared), and *z* (infrared).

defined as the difference of magnitudes between two different filter bands. Subtracting two magnitudes reduces this uncertainty. For example, an observation of the star Vega above will consist of a vector four numbers: `[u-g, g-r, r-i, i-z]`.

With the difficulty of obtaining informative spectra, and the relative ease of obtaining color information, machine-learning tasks in Astronomy are often based on a small spectroscopic training set, which is applied to a larger set of photometric observations with unknown classification. We'll examine a few of these situations here.

4.1 Motivation: Why is this Important?

The study of *quasars*, an amalgamation of the words "quasi-stellar radio source", has led to many advances in our understanding of fundamental physics. Quasars, also commonly referred to as QSOs (Quasi-Stellar Objects) or AGNs (Active Galactic Nuclei) are galaxies which contain supermassive black holes at their core. These black holes can weigh-in at over 10 billion times the mass of our sun, and can be luminous enough to out-shine their entire galaxy. Here we show three different objects, chosen from among the millions of sources catalogued by the Sloan Digital Sky Survey[6]:

The featured object is at the center of each image. On the left is a star, in the center is a galaxy, and on the right is a distant quasar. From these images alone, it would be impossible to distinguish between the star and the quasar: both are unresolved point-sources of similar apparrent brightness. If a spectrum were available, distinguishing between them could be accomplished rather straightforwardly, but spectra are not always available. Using multi-color photometric information, rather than just a single image, however, this task becomes feasible. The goal here is to design a machine-learning algorithm which can accurately distinguish stars from quasars based on multi-color photometric measurements.

[6]http://www.sdss.org

4.2 Star-Quasar Classification: Naive Bayes

Note: The information in this section is available in an interactive notebook `07_classification_example.ipynb`, which can be viewed using iPython notebook[7].

In the folder `$TUTORIAL_HOME/data/sdss_colors`, there is a script `fetch_data.py` which will download the colors of over 700,000 stars and quasars from the Sloan Digital Sky Survey. 500,000 of them are training data, spectroscopically identified as stars or quasars. The remaining 200,000 have been classified based on their photometric colors.

Here we will use a Naive Bayes estimator to classify the objects. First, we will construct our training data and test data arrays:

```
>>> import numpy as np
>>> train_data = np.load('data/sdss_colors/sdssdr6_colors_class_train.npy')
>>> test_data = np.load('data/sdss_colors/sdssdr6_colors_class.200000.npy')
```

Now we must put these into arrays of shape `(n_samples, n_features)` in order to pass them to routines in scikit-learn. Training samples with zero-redshift are stars, while samples with positive redshift are quasars:

```
>>> X_train = np.vstack([train_data['u-g'],
...                      train_data['g-r'],
...                      train_data['r-i'],
...                      train_data['i-z']]).T
>>> y_train = (train_data['redshift'] > 0).astype(int)
>>> X_test = np.vstack([test_data['u-g'],
...                     test_data['g-r'],
...                     test_data['r-i'],
...                     test_data['i-z']]).T
>>> y_test = (test_data['label'] == 0).astype(int)
```

Notice that we've set this up so that quasars have `y = 1`, and stars have `y = 0`. Now we'll set up a Naive Bayes classifier. This will fit a four-dimensional uncorrelated gaussian to each distribution, and from these gaussians quickly predict the label for a test point:

```
>>> from sklearn import naive_bayes
>>> gnb = naive_bayes.GaussianNB()
>>> gnb.fit(X_train, y_train)
>>> y_pred = gnb.predict(X_test)
```

Let's check our accuracy. This is the fraction of labels that are correct:

[7] http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

```
>>> accuracy = float(np.sum(y_test == y_pred)) / len(y_test)
>>> print accuracy
0.617245
```

We have 61% accuracy. Not very good. But we must be careful here: the accuracy does not always tell the whole story. In our data, there are many more stars than quasars

```
>>> print np.sum(y_test == 0)
186721
>>> print np.sum(y_test == 1)
13279
```

Stars outnumber Quasars by a factor of 14 to 1. In cases like this, it is much more useful to evaluate the fit based on *precision* and *recall*. Because there are many fewer quasars than stars, we'll call a quasar a *positive* label and a star a *negative* label. The precision asks what fraction of positively labeled points are correctly labeled:

$$\text{precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

The recall asks what fraction of positive samples are correctly identified:

$$\text{recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

We can calculate this for our results as follows:

```
>>> TP = np.sum((y_pred == 1) & (y_test == 1))
>>> FP = np.sum((y_pred == 1) & (y_test == 0))
>>> FN = np.sum((y_pred == 0) & (y_test == 1))
>>> print TP / float(TP + FP)  # precision
0.142337086782
>>> print TP / float(TP + FN)  # recall
0.948113562768
```

For convenience, these can be computed using the tools in the `metrics` sub-package of scikit-learn:

```
>>> from sklearn import metrics
>>> metrics.precision_score(y_test, y_pred)
0.14233708678153123
>>> metrics.recall_score(y_test, y_pred)
0.94811356276828074
```

Another useful metric is the F1 score, which gives a single score based on the precision and recall for the class:

$$\text{F1} = 2\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

In a perfect classification, the precision, recall, and F1 score are all equal to 1.

```
>>> metrics.f1_score(y_test, y_pred)
0.24751550658108151
```

For convenience, `sklearn.metrics` provides a function that computes all of these scores, and returns a nicely formatted string. For example:

```
>>> print metrics.classification_report(y_test, y_pred, target_names=['Stars', 'QSOs'])
              precision    recall  f1-score   support

       Stars       0.99      0.59      0.74    186721
        QSOs       0.14      0.95      0.25     13279

 avg / total       0.94      0.62      0.71    200000
```

We see that for Gaussian Naive Bayes, our QSO recall is fairly good: we are correctly identifying 95% of all quasars. The precision, on the other hand, is much worse. Of the points we label quasars, only 14% of them are correctly labeled. This low precision leads to an F1-score of only 0.25. This is not an optimal classification of our data. Apparently Naive Bayes is a bit too naive for this problem.

Later, in *Exercise #1* (page **??**), we will apply a more sophisticated learning method to this task, which will potentially improve on these results.

CHAPTER 5

Regression: Photometric Redshifts of Galaxies

Another important learning task in astronomy is the problem of determining redshifts[1] of distant galaxies. In the current standard cosmological model, the universe began nearly 14 billion years ago, in an explosive event commonly known as the Big Bang. Since then, the very fabric of space has been expanding[2], so that distant galaxies appear to be moving away from us at very high speeds. The uniformity of this expansion means that there is a relationship between the distance to a galaxy, and the speed that it appears to be receeding from us (this relationship is known as Hubble's Law[3], named after Edwin Hubble). This recession speed leads to a shift in the frequency of photons, very similar to the more familiar doppler shift[4] that causes the pitch of a siren to change as an emergency vehicle passes by. If a galaxy or star were moving toward us, its light would be shifted to higher frequencies, or *blue-shifted*. Because the universe is expanding away from us, distant galaxies appear to be *red-shifted*: their photons are shifted to lower frequencies.

In cosmology, the redshift is measured with the parameter z, defined in terms of the observed wavelength λ_{obs} and the emitted wavelength λ_{em}:

$$\lambda_{obs} = (1 + z)\lambda_{em}$$

When a spectrum can be obtained, determining the redshift is rather straight-forward: if you can localize the spectral fingerprint of a common element, such as hydrogen, then the redshift can be computed using simple arithmetic. But similarly to the case of Star/Quasar classification, the task becomes much more difficult when only photometric observations are available.

Because of the spectrum shift, an identical source at different redshifts will have a different color through each pair of filters. See the following figure:

At redshift $z = 0.0$, the spectrum is bright in the *u* and *g* filters, but dim in the *i* and *z* filters. At redshift $z = 0.8$, the opposite is the case. This suggests the possibility of determining redshift from photometry alone. The situation is complicated by the fact that each individual source has unique spectral characteristics, but nevertheless, these *photometric redshifts* are often used in astronomical applications.

[1] http://en.wikipedia.org/wiki/Redshift
[2] http://en.wikipedia.org/wiki/Expansion_of_space
[3] http://en.wikipedia.org/wiki/Hubble_expansion
[4] http://en.wikipedia.org/wiki/Doppler_shift

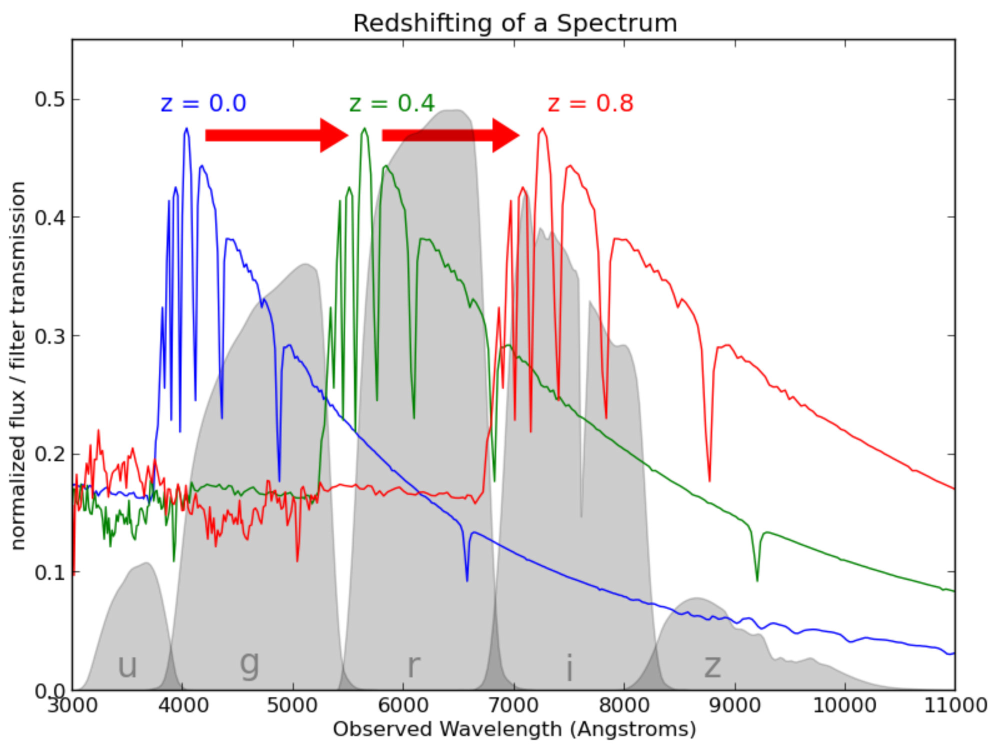


Figure 5.1: The spectrum of the star Vega (α-Lyr) at three different redshifts. The SDSS ugriz filters are shown in gray for reference.

5.1 Motivation: Dark Energy, Dark Matter, and the Fate of the Universe

The photometric redshift problem is very important. Future astronomical surveys hope to image trillions of very faint galaxies, and use this data to inform our view of the universe as a whole: its history, its geometry, and its fate. Obtaining an accurate estimate of the redshift to each of these galaxies is a pivotal part of this task. Because these surveys will image so many extremely faint galaxies, there is no possibility of obtaining a spectrum for each one. Thus sophisticated photometric redshift codes will be required to advance our understanding of the Universe, including more precisely understanding the nature of the dark energy that is currently accelerating the cosmic expansion.

5.2 A Simple Method: Decision Tree Regression

Note: The information in this section is available in an interactive notebook `08_regression_example.ipynb`, which can be viewed using iPython notebook[5].

Here we'll take an extremely simple approach to the photometric redshift problem, using a decision tree. In the folder `$TUTORIAL_HOME/data/sdss_photoz`, there is a script `fetch_data.py` which will download the colors of 400,000+ galaxies from the Sloan Digital Sky Survey. This script also includes a python implementation of the SQL query used to construct this data. This template can be modified to download more features if desired. Before executing the example below, run `fetch_data.py` to download the colors and redshifts.

[5]http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

First we will load this data, shuffle it in preparation for later, and arrange the colors in an array of shape `(n_samples, n_features)`:

```
>>> import numpy as np
>>> data = np.load('data/sdss_photoz/sdss_photoz.npy')
>>> N = len(data)
>>> X = np.zeros((N, 4))
>>> X[:, 0] = data['u'] - data['g']
>>> X[:, 1] = data['g'] - data['r']
>>> X[:, 2] = data['r'] - data['i']
>>> X[:, 3] = data['i'] - data['z']
>>> z = data['redshift']
```

Next we'll split the data into two samples: a training sample and a test sample which we'll use to evaluate our training:

```
>>> Ntrain = 3 * N / 4
>>> Xtrain = X[:Ntrain]
>>> ztrain = z[:Ntrain]
>>> Xtest = X[Ntrain:]
>>> ztest = z[Ntrain:]
```

Now we'll use the scikit-learn `DecisionTreeRegressor` method to train a model and predict redshifts for the test set based on a 20-level decision tree:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> clf = DecisionTreeRegressor(
>>> clf.fit(Xtrain, ztrain)
>>> zpred = clf.predict(Xtest)
```

To judge the efficacy of prediction, we can compute the root-mean-square difference between the true and predicted values:

```
>>> rms = np.sqrt(np.mean((ztest - zpred) ** 2))
>>> print rms
0.221409442926
```

Our RMS error is about 0.22. This is pretty good for such an unsophisticated learning algorithm, but better algorithms can improve on this. The biggest issue here are the *catastrophic errors*, where the predicted redshift is extremely far from the prediction:

```
>>> print len(ztest)
102798
>>> print np.sum(abs(ztest - zpred) > 1)
1538
```

About 1.5% of objects have redshift estimates which are off by greater than 1. This sort of error in redshift determination is very problematic for high-precision cosmological studies. This can be seen in a scatter plot of the predicted redshift versus the true redshift for the test data:

Later, in *Exercise #2* (page **??**), we will attempt to improve on this by optimizing the parameters of the decision tree.

In practice, the solutions to the photometric redshift problem can benefit from approaches that use physical intuition as well as machine learning tools. For example, some solutions involve the use of libraries of synthetic galaxy spectra which are known to be representative of the true galaxy distribution. This extra information can be used either directly, in a physically motivated analysis, or can be used to generate a larger suite of artificial training instances for a pure machine learning approach.

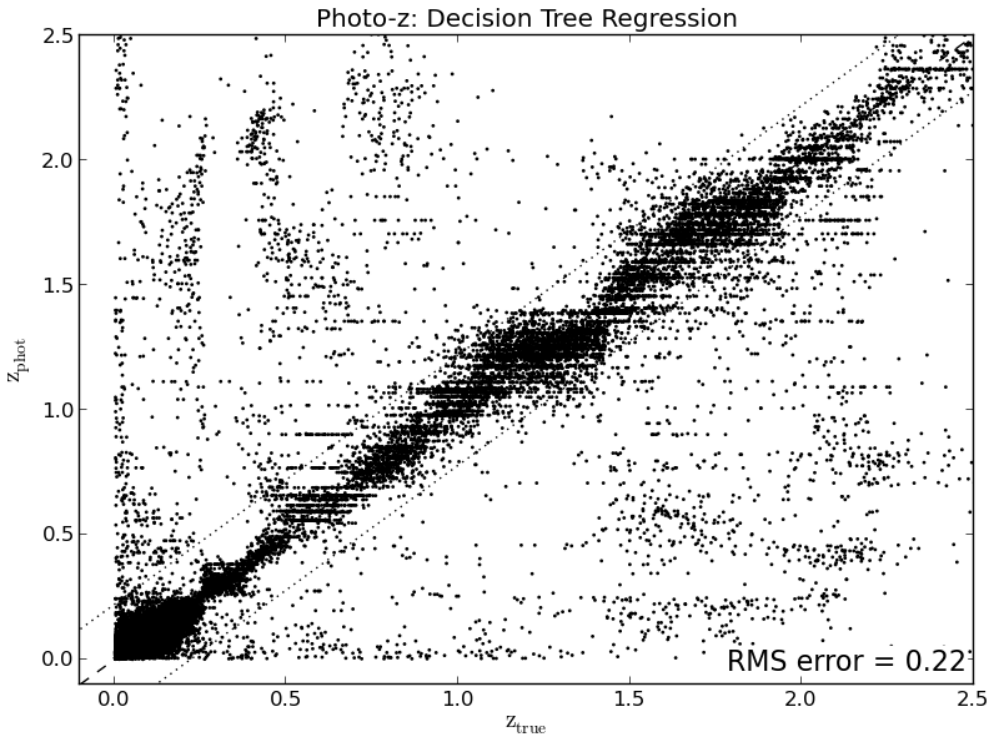


Figure 5.2: The true and predicted redshifts of 102,798 SDSS galaxies, using a simple decision tree regressor. Notice the presece of catastrophic outliers: those galaxies whose predicted redshifts are extremely far from the true value.

CHAPTER 6

Dimensionality Reduction of Astronomical Spectra

The Sloan Digital Sky Survey[1] is a photometric and spectroscopic survey which has operated since the year 2000, and has resulted in an unprecedented astronomical database. The database contains photometric observations like those we explored in the previous sections, but also includes a large number of spectra of various objects. These spectra are n-dimensional data vectors (generally, $n \approx 4000$) for each observed object, where each observation in the vector measures the flux of a particular wavelength of light.

Because of the large dimensionality of this data, visualization of the dataset becomes very challenging. This is where unsupervised dimensionality reduction methods can be useful. One of the most commonly used dimensionality reduction methods in astronomy is Principal Component Analysis (PCA)[2]. We won't go through the mathematical details here, but PCA essentially seeks dimensions of the input data which contain the bulk of the variability present in the data. The model has this form:

$$\vec{x_i} = \vec{\mu} + \sum_{j=1}^{n} a_{ij}\vec{v_j}$$

Here $\vec{x_i}$ represents an individual spectrum. $\vec{\mu}$ is the mean spectrum for the dataset. The remaining term encodes the contributions of each of the *eigenvectors* $\vec{v_j}$. The eigenvectors are generally arranged so that those with the smallest j contain the most signal-to-noise, and are the most important vectors in reconstructing the spectra. For this reason, truncating the sum at some $m < n$ can still result in a faithful representation of the input.

For astronomical spectra, the result is that the first few coefficientts a_{ij} of each spectrum $\vec{x_i}$ encode a good low-dimensional representation of the spectrum. We'll use this fact to visualize this data in a meaningful way.

6.1 SDSS Spectral Data

Note: The information in this section is available in an interactive notebook `09_dimensionality_example.ipynb`, which can be viewed using iPython notebook[3].

[1] http://www.sdss.org/
[2] http://scikit-learn.org/stable/modules/decompositions.html#principal-component-analysis-pca
[3] http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

In the directory `$TUTORIAL_HOME/data/sdss_spectra/`, you'll find the script which fetches a set of spectra from the Sloan Digital Sky Survey. Each individual spectrum is at a particular redshift, and can have data missing in certain spectral regions. So that this doesn't affect our analysis, the spectra have been de-redshifted and the gaps have been filled using the PCA-based algorithm described in [4]. In the process of shifting and gap-filling, the spectra have been down-sampled so that the number of attributes is $n = 1000$.

Once the dataset is downloaded, it can be read-in as follows:

```
>>> import numpy as np
>>> data = np.load('spec4000_corrected.npz')
>>> data.files
['y', 'X', 'z', 'wavelengths']
>>> data['X'].shape
(4000, 1000)
>>> print data['labels']
['unknown' 'star' 'absorption galaxy' 'galaxy' 'emission galaxy'
 'narrow-line QSO' 'broad-line QSO' 'sky' 'Hi-z QSO' 'Late-type star']
```

The variable `X` contains 4000 spectra, each with 1000 attributes. In addition, the file includes a classification code `y`, and redshift `z` for each spectrum, and an array `wavelengths` which can aid in plotting spectra. Let's plot a few of these to see what they look like, making sure to choose a representative of each of the interesting classes:

```
>>> import pylab as pl
>>> def plot_spectral_types(data):
...     X = data['X']
...     y = data['y']
...     wavelengths = data['wavelengths']
...     labels = data['labels']

...     for i_class in (2, 3, 4, 5, 6):
...         i = np.where(y == i_class)[0][0]
...         l = pl.plot(wavelengths, X[i] + 20 * i_class)
...         c = l[0].get_color()
...         pl.text(6800, 2 + 20 * i_class, labels[i_class], color=c)

...     pl.subplots_adjust(hspace=0)
...     pl.xlabel('wavelength (Angstroms)')
...     pl.ylabel('flux + offset')
...     pl.title('Sample of Spectra')
```

calling `plot_spectral_types(data)` gives the following figure:

There are 4000 spectra in this file, each with 1000 attributes. Visualizing a dataset of this size can be very difficult. We could plot all 4000 spectra as we did above, but the plot would quickly become too complicated. As a first step, it might be helpful to ask what the average spectrum looks like. To do this, we'll plot the mean, but first we'll normalize the spectra. Because the spectra represent galaxies at distances that range over several hundreds of light-years, their total flux varies greatly. Thus it will help if we normalize the spectra. For this we'll use the scikit-learn preprocessing module. We'll then plot both the mean and standard deviation to give us an idea of the data we're working with:

```
>>> from sklearn.preprocessing import normalize
>>>
>>> def plot_mean_std(data):
...     X = data['X']
...     wavelengths = data['wavelengths']
...
...     X = normalize(X)
```

[4] C.W. Yip et al. Spectral Classification of Quasars in the Sloan Digital Sky Survey: Eigenspectra, Redshift, and Luminosity Effects. Astronomical Journal 128:6, 2004.

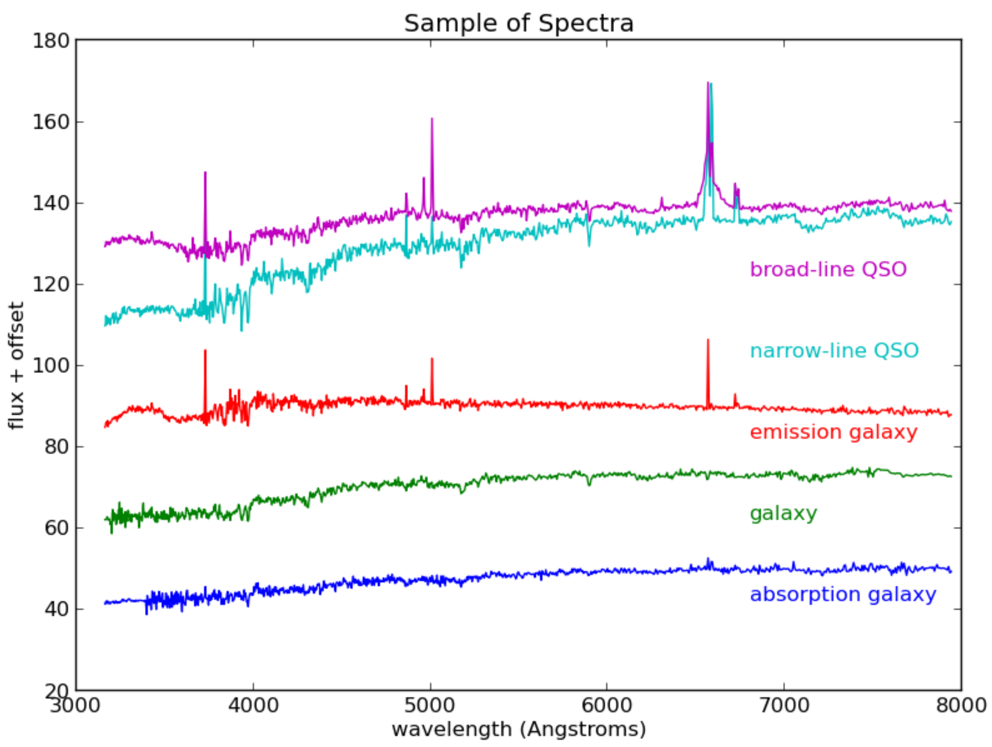


Figure 6.1: A sample from among the most interesting classes of object. These objects are classified based on their emission and absorption characteristics.

```
...     mu = X.mean(0)
...     std = X.std(0)
...     pl.plot(wavelengths, mu, color='black')
...     pl.fill_between(wavelengths, mu - std, mu + std, color='#CCCCCC')
...     pl.xlim(wavelengths[0], wavelengths[-1])
...     pl.ylim(0, 0.06)
...     pl.xlabel('wavelength (Angstroms)')
...     pl.ylabel('scaled flux')
...     pl.title('Mean Spectrum')
```

Calling `plot_mean_std(data)` gives the following figure:

The interesting part of the data is in the gray shaded regions: how do spectra vary from the mean, and how can this variation tell us about their physical properties? One option to visualize this would be to scatter-plot random pairs of observations from each spectrum. We'll create a function to visualize this:

```
>>> def plot_random_projection(data, rseed=25255):
...     # rseed is chosen to emphasize correlation
...     np.random.seed(rseed)
...     i1, i2 = np.random.randint(1000, size=2)
...
...     # create a formatter which works for our labels
...     format = pl.FuncFormatter(lambda i, *args: labels[i].replace(' ', '\n'))
...
...     X = normalize(data['X'])
...     y = data['y']
...     labels = data['labels']
...     wavelengths = data['wavelengths']
```

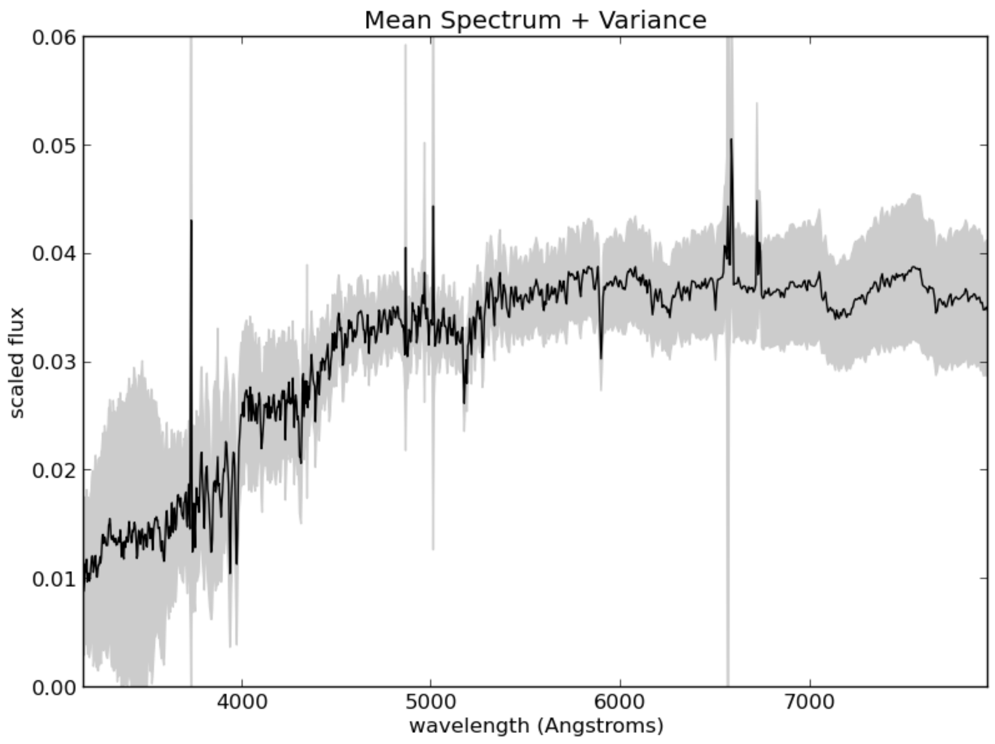


Figure 6.2: The mean and standard deviation of the normalized spectra. Some of the largest variation is found at wavelengths at which Hydrogen absorbs and emits photons (Hydrogen is by far the most abundant atom in the universe). For example, the line at 6563 is known as Hydrogen-α, and is often seen in emission (spiking up) in quasars and other active galaxies.

```
...
...     pl.scatter(X[:, i1], X[:, i2], c=y, s=4, lw=0,
...                vmin=2, vmax=6, cmap=pl.cm.jet)
...     pl.colorbar(ticks = range(2, 7), format=format)
...     pl.xlabel('wavelength = %.1f' % wavelengths[i1])
...     pl.ylabel('wavelength = %.1f' % wavelengths[i2])
...     pl.title('Random Pair of Spectra Bins')
```

Calling `plot_random_projection(data)` gives the following plot:

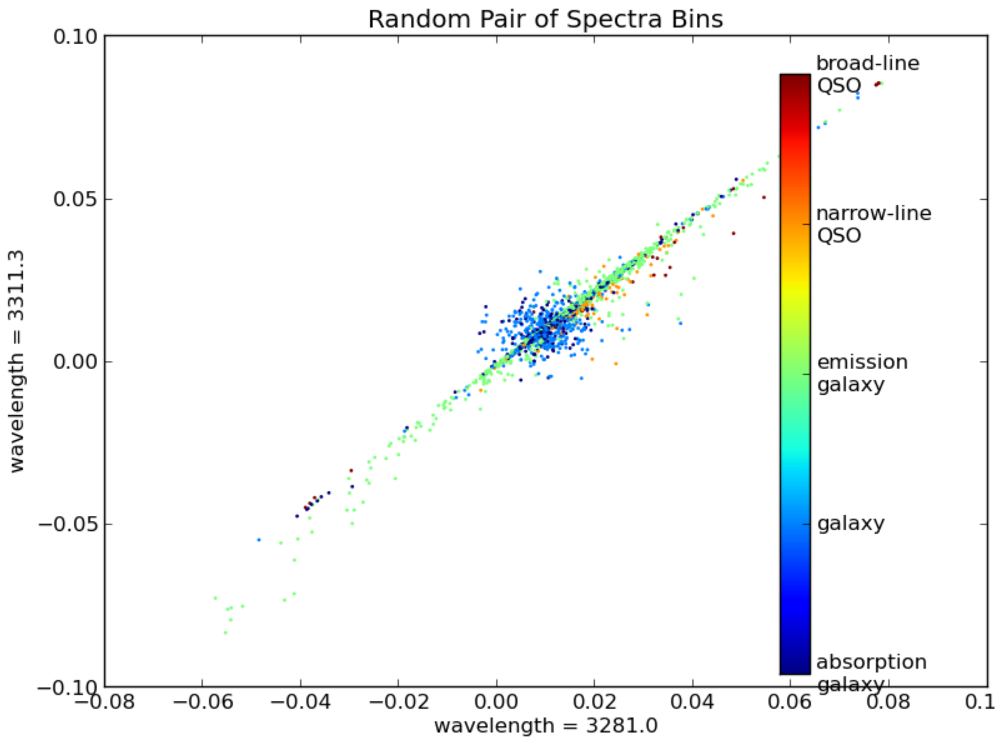


Figure 6.3: A scatter-plot of a random pair of observations from the data. These show a clear correlation.

There is a clear correlation between these two measurements. That is, if you know the value of one, then you could quite accurately predict the value of the other. This shows us that some of the spectral bins do not add much information, and can be ignored. One could imagine proceeding by trial and error, plotting pairs of points and seeing which ones provide the most interesting information, but this would be very tedious. Instead, we can use an automated technique for dimensionality reduction, one well-known example of which is Principal Component Analysis.

6.2 Principal Component Analysis

Principal Component Analysis (PCA) is an often-used tool in astronomy and other data-intensive sciences. In a sense, it automates the trial-and-error process discussed in the previous section, and finds the most interesting linear combinations of attributes, so that high-dimensional data can be visualized in a 2D or 3D plot. Scikit-learn has methods to compute PCA and several variants. Classic PCA (`sklearn.decomposition.PCA`[5]) is based on an eigenvalue decomposition of the data covariance, so that for N points, the computational cost grows as $\mathcal{O}[N^3]$. This means that for large datasets like the current one, the fit can be very slow. You can try it as follows, but the computation may take up to several minutes for this dataset:

[5]http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html#sklearn.decomposition.PCA

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=4)
>>> X_projected = pca.fit_transform(X)  # warning: this takes a long time!
```

Fortunately, scikit-learn has an alternative method that is much faster. The speed comes at a price: it is based on random projections, so the results are not as robust as the normal method. But for tasks such as ours where we are seeking only a few of a large number of eigenvectors, it performs fairly well. To keep our results consistent between runs, we'll explicitly set the random seed for the fit. You should repeat this with several different random seeds to convince yourself that the results are consistent:

```
>>> from sklearn.decomposition import RandomizedPCA
>>> rpca = RandomizedPCA(n_components=4, random_state=0)
>>> X_proj = rpca.fit_transform(X)
>>> X_proj.shape
(4000, 4)
```

`X_proj` is now a reduced-dimension representation of `X`, where the lower-index columns are the most important dimensions. We can visualize the spectra now using the first two columns:

```
>>> def plot_PCA_projection(data, rpca):
...     y = data['y']
...
...     # create a formatter which works for our labels
...     labels = data['labels']
...     format = pl.FuncFormatter(lambda i, *args: labels[i].replace(' ', '\n'))
...
...     X_proj = rpca.transform(X)

...     pl.scatter(X_proj[:, 0], X_proj[:, 1], c=y, s=4, lw=0, vmin=2, vmax=6, cmap=pl.cm.jet)
...     pl.colorbar(ticks = range(2, 7), format=format)
...     pl.xlabel('coefficient 1')
...     pl.ylabel('coefficient 2')
...     pl.title('PCA projection of Spectra')
```

Calling this function as `plot_PCA_projection(data, rpca)` gives the following plot:

We now have a two-dimensional visualization, but what does this tell us? Looking at the PCA model in the equation above, we see that each component is associated with an eigenvector, and this plot is showing a_{i1} and a_{i2} where

$$\vec{s_i} \approx \vec{\mu} + a_{i1}\vec{v_1} + a_{i2}\vec{v_2}$$

Visualizing the *eigenvectors* $\vec{v_j}$ can give insight into what these components mean:

```
>>> def plot_eigenvectors(data, rpca):
...     wavelengths = data['wavelengths']
...
...     l = pl.plot(wavelengths, rpca.mean_ - 0.15)
...     c = l[0].get_color()
...     pl.text(7000, -0.16, "mean", color=c)
...
...     for i in range(4):
...         l = pl.plot(wavelengths, rpca.components_[i] + 0.15 * i)
...         c = l[0].get_color()
...         pl.text(7000, -0.01 + 0.15 * i, "component %i" % (i + 1), color=c)
...
...     pl.ylim(-0.2, 0.6)
...     pl.xlabel('wavelength (Angstroms)')
```

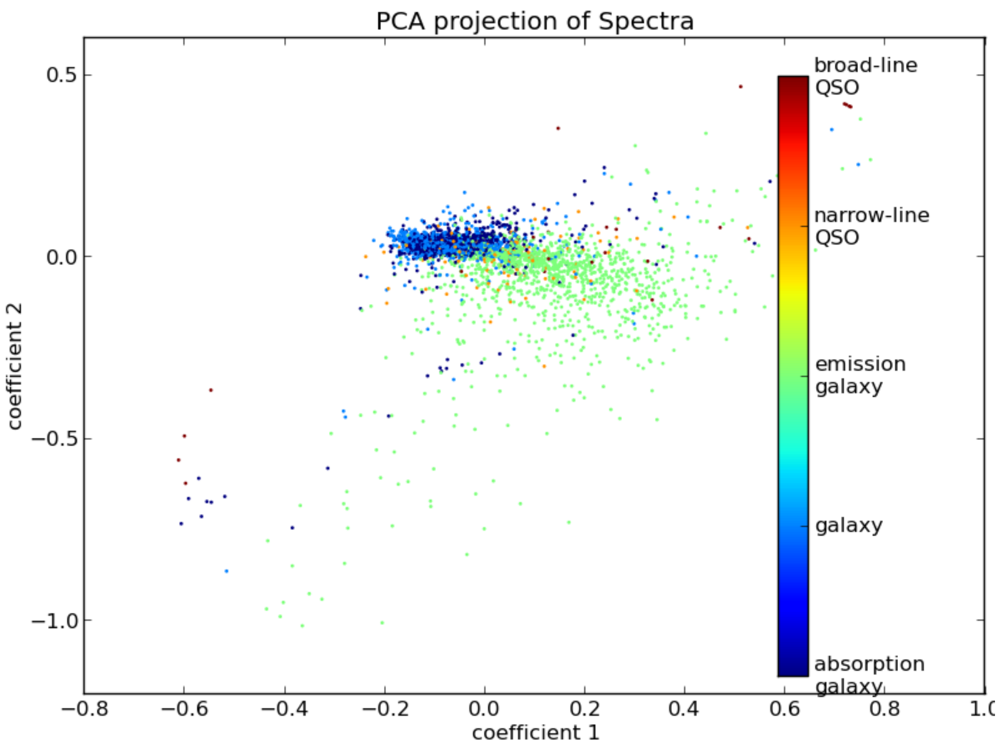


Figure 6.4: A scatter-plot of the first two principal components of the spectra.

```
...     pl.ylabel('scaled flux + offset')
...     pl.title('Mean Spectrum and Eigen-spectra')
```

Calling this function as `plot_eivenvectors(data, rpca)` gives the following plot:

We see that the first eigenspectrum (component 1) tells us about the relative difference in flux between low wavelengths and high wavelengths - that is, the color of the spectrum. Component 2 tells us a lot about the emission and absorption characteristics in the various lines, and also in the so-called "4000 angstrom break" due to Hydrogen absorption. Detailed analysis of these components and eigenspectra can lead to much physical insight about the galaxies in the fit (See, for example [1]).

Nevertheless, there are some weaknesses here. First of all, PCA does not do a good job of separating out galaxies with different emission characteristics. We'd hope for a projection which reflects the fact that narrow spectral features are very important in the classification. PCA does not do this. Later, in *Exercise 3* (page **??**), we'll explore some alternative nonlinear dimensionality reduction techniques which will address this deficiency of PCA.

6.3 References

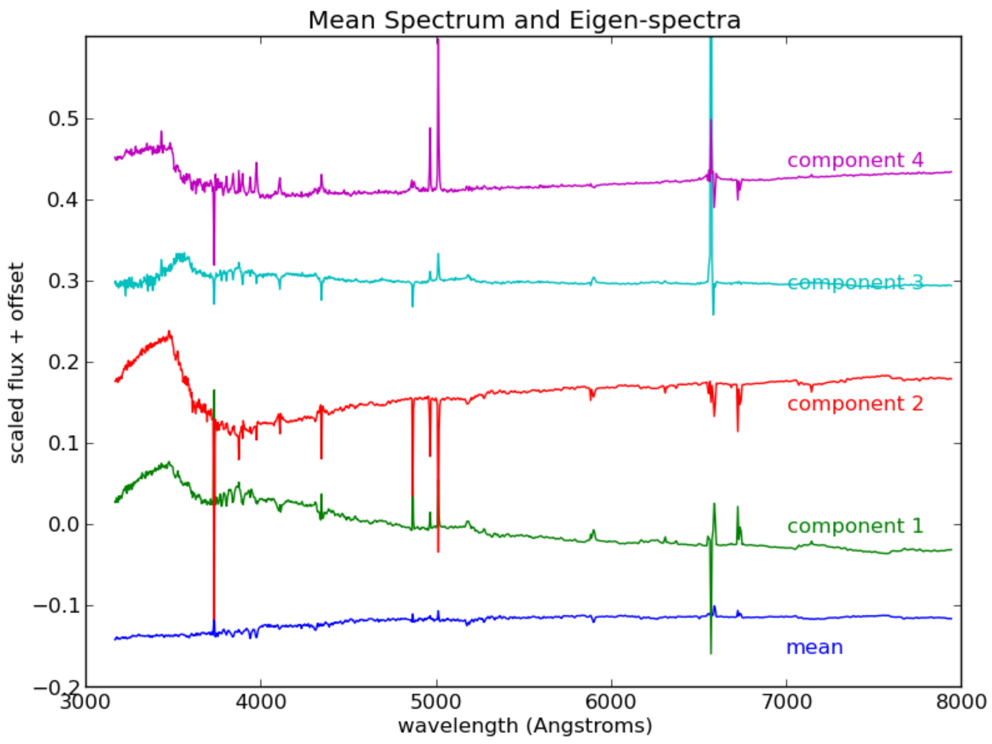


Figure 6.5: The mean spectrum and the first four eigenvectors of the spectral data.

CHAPTER 7

Exercises: Taking it a step further

Here we describe three exercises that will walk you through more advanced approaches to the classification, regression, and dimensionality reduction tasks outlined in the previous sections.

Before beginning the exercises, be sure you have downloaded the tutorial source as described in the setup section. For each exercise, there are three files. In `$TUTORIAL_HOME/skeletons`, there is a skeleton script which can be filled-in to complete the exercise. In `$TUTORIAL_HOME/solutions` you can find a completed version of the exercise files. `$TUTORIAL_HOME/notebooks` has iPython notebook[1] files which can be used to interactively complete the tutorials.

Using the notebook files is the preferred way to complete the tutorial. If ipython notebook is unavailable, the files in the `skeletons` and `solutions` directory can be used. Begin by copying the content of the `skeletons` folder to your own workspace:

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

You can then edit the contents of `workspace` without losing the original files. You must also make sure you have run the `fetch_data.py` scripts in the appropriate subdirectory of `$TUTORIAL_HOME/data`. Next start the ipython interpreter and run the script with:

```
In [1]: %run workspace/exercise*.py datadir
```

If an exception is triggered, you can type `%debug` to fire-up an ipdb session.

Each exercise contains the necessary import statements, data loading, and code to evaluate the predictive accuracy of the model being used. The remaining code must be filled-in by the user. These places are marked by the comment `#TODO`. Each exercise also has in-line descriptions which go over the details of each sub-task.

7.1 Exercise 1: Photometric Classification with GMM

Note: The information in this section is available in an interactive notebook `10_exercise01.ipynb`, which can be viewed using iPython notebook[2]. This requires the sdss colors data, which can be downloaded as described in *Tutorial Setup and Installation* (page **??**).

[1]http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html
[2]http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

In this exercise, you will improve on the results of the classification example described in the classification section. We previously used a simple Gaussian Naive Bayes classifier to distinguish between stars and quasars. Here we will use Gaussian Mixture Models[3] to recreate the Gaussian Naive Bayes classifier, and then tweak the parameters of these mixture models, evaluating them using a cross-validation set, and attempt to arrive at a better classifier.

The overall goal of the task is to improve the quasar classifier, measuring its performance using the precision, recall, and F1-score.

This task is broken into several parts:

1. Re-implement Gaussian Naive Bayes using Gaussian Mixture models. This involves training two single-component GMMs, calculating the priors for each class, and using these to compute likelihoods. This can be compared directly with the Gaussian Naive Bayes results to check that the implementation is correct.
2. Experiment with various covariance types and numbers of components to optimize the classifier using the cross-validation set.
3. Once you've converged on a good set of GMM parameters, predict the labels for the test set, and compare this with the labels from the literature.

the ipython command is:

```
In [1]: %run workspace/exercise_01.py data/sdss_colors
```

Althernatively, the ipython notebook file is found in

```
notebooks/10_exercise01.ipynb
```

7.2 Exercise 2: Photometric redshifts with Decision Trees

Note: The information in this section is available in an interactive notebook `11_exercise02.ipynb`, which can be viewed using iPython notebook[4]. This requires the sdss photoz data, which can be downloaded as described in *Tutorial Setup and Installation* (page **??**).

In this exercise, you will seek to improve the results of the photometric redshift regression problem described in the regression section. This exercise will draw heavily on the concepts of bias and variance, using the learning curve plots introduced in section 3 of this tutorial.

There are two goals of this exercise:

1. Find the best possible decision tree classifier for the data
2. Decide what the best use of future resources is. Should the astronomers seek to observe more objects (i.e. increase the number of training samples), or record more observations for each object (i.e. increase the number of features)?

The exercise is broken into the following tasks:

1. Compute the training error and cross-validation error as a function of the `max_depth` parameter used in the Decision Tree Classifier.
2. Compute the training error and cross-validation error as a function of the number of training samples.
3. Repeat these two tasks, recording the outlier rate rather than the rms error.

[3]http://scikit-learn.org/0.6/modules/mixture.html
[4]http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html

4. Analyze these results: should future observations focus on increasing the number of samples, or increasing the number of features? Does this answer change depending on whether the rms error or outlier rate is the metric used?

the ipython command is:

```
In [1]: %run workspace/exercise_02.py data/sdss_photoz/
```

Althernatively, the ipython notebook file is found in

```
notebooks/11_exercise02.ipynb
```

7.3 Exercise 3: Dimensionality Reduction of Spectra

Note: The information in this section is available in an interactive notebook `12_exercise03.ipynb`, which can be viewed using iPython notebook[5]. This requires the sdss spectra data, which can be downloaded as described in *Tutorial Setup and Installation* (page **??**).

In this exercise, you will use several dimensionality reduction techniques to view low-dimensional projections of galaxy & quasar spectra from the Sloan Digital Sky Survey. This exercise is much less quantitative than the previous ones: it mainly will help you to get a qualitative sense of the characteristics of these learning methods.

There is a programming section, followed by an experimentation section. The skeleton is set up to use command-line options to compare different sets of parameters

7.3.1 Programming

The file has several places with "TODO" marked. In these, you will use the specified unsupervised method to project the data `X` into the lower-dimensional `X_proj`.

1. Use `sklearn.decomposition.RandomizedPCA`[6] to project the data

   the ipython command is:

   ```
   In [1]: %run workspace/exercise_03.py data/sdss_spectra/ -m pca
   ```

   Note the argument `-m` which specifies the method to use.

2. Use `sklearn.manifold.LocallyLinearEmbedding`[7] with `method='standard'` to project the data.

   the ipython command is:

   ```
   In [1]: %run workspace/exercise_03.py data/sdss_spectra/ -m lle
   ```

3. Use `sklearn.manifold.LocallyLinearEmbedding`[8] with `method='standard'` to project the data.

   the ipython command is:

   ```
   In [1]: %run workspace/exercise_03.py data/sdss_spectra/ -m mlle
   ```

[5] http://ipython.org/ipython-doc/stable/interactive/htmlnotebook.html
[6] http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.RandomizedPCA.html#sklearn.decomposition.RandomizedPCA
[7] http://scikit-learn.org/stable/modules/generated/sklearn.manifold.LocallyLinearEmbedding.html#sklearn.manifold.LocallyLinearEmbedding
[8] http://scikit-learn.org/stable/modules/generated/sklearn.manifold.LocallyLinearEmbedding.html#sklearn.manifold.LocallyLinearEmbedding

4. Use `sklearn.manifold.Isomap`[9] to project the data.

   the ipython command is:

   ```
   In [1]: %run workspace/exercise_03.py data/sdss_spectra/ -m isomap
   ```

7.3.2 Experimentation

Your goal is to find a projection that does a good job of separating the various classes of spectra, and lays them out in a way that might allow intuitive evaluation of the relationships between points. The script is set-up as a command-line interface. You should address the following questions:

1. How sensitive is PCA to the set of data used? To the number of training points? You can test this out as follows:

   ```
   In [1]: %run workspace/exercise_03.py data/sdss_spectra -m pca -n 1000 -s
   ```

   This will perform PCA on a subset of 1000 points. `-s` indicates that the data should be shuffled, so that the set of points is different every time. How stable is the projection between different subsets of the data? How does the projection change as the number of points is increased?

2. Address the same questions with LLE, MLLE, and Isomap. Which of these manifold methods appears to give the most stable results?

3. Now we can vary the number of neighbors used with LLE, MLLE, and Isomap. This is accomplished as follows:

   ```
   In [1]: %run workspace/exercise_03.py data/sdss_spectra -m lle -k 20
   ```

   This call will execute LLE with 20 neighbors. Try this for several values of *k*. How does the number of neighbors change the projection? Among LLE, MLLE, and Isomap, which produces the most stable results as the number of neighbors are changed?

4. Finally, we'll test the effects of normalization. This can be done as follows:

   ```
   In [1]: %run workspace/exercise_03.py data/sdss_spectra -N l2
   ```

   this will perform PCA with L2-normalization. The other options are `-N l1` for L1-normalization, and `-N none` for no normalization. Normalization has the effect of bringing all the spectra closer together: unnormalized spectra may be very bright (for nearby objects) or very dim (for far away objects). Normalization corrects for this source of variance in the data. How do the projected results change as you vary the normalization?

5. By now, you should have an idea of which method and which combination of parameters give the best qualitative separation between the points. Re-run this method using the full ‘n‘=4000 dataset now:

   ```
   In [1]: %run python workspace/exercise_03.py data/sdss_spectra -n 4000 -m [method] [other option
   ```

   This should give you a projection of the data that gives a good visualization of the relationship between points. An astronomer may go further and try to develop rough cut-offs that would give a broad classification to an unlabeled test point. This sort of procedure could be used as the first step of a physically-motivated classification pipeline, or to flag potentially interesting objects for quick followup.

[9]http://scikit-learn.org/stable/modules/generated/sklearn.manifold.Isomap.html#sklearn.manifold.Isomap

CHAPTER 8

Code examples

Applications to experiment with scikit-learn tools.

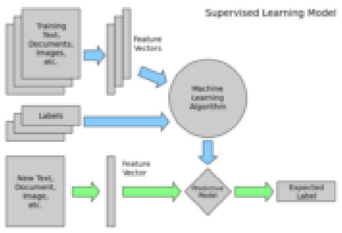
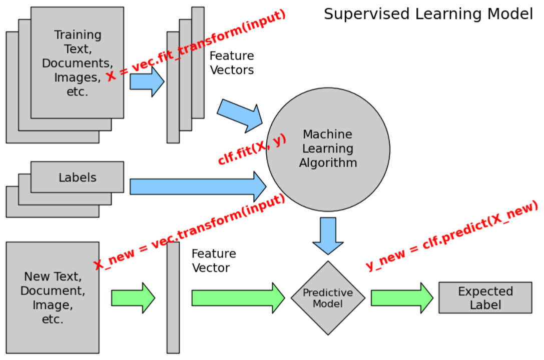
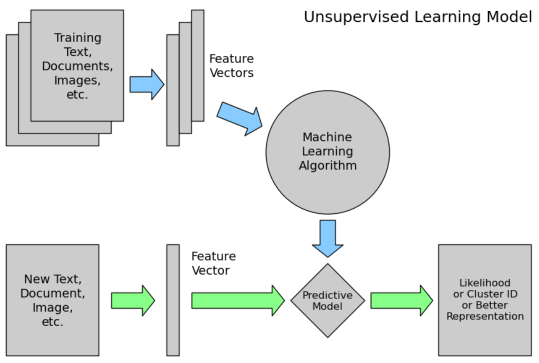
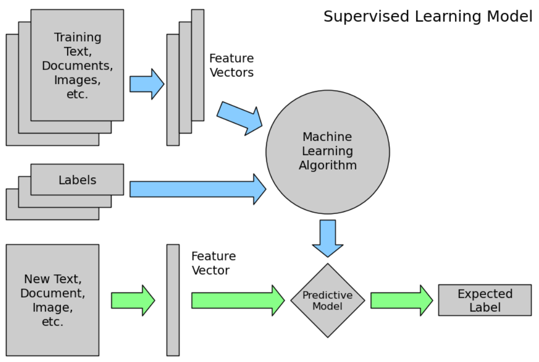


Figure 8.1: *Tutorial Diagrams* (page **??**)

8.1 Tutorial Diagrams

This script plots the flow-charts used in the scikit-learn tutorials.

- 
Supervised Learning Model
Training Text, Documents, Images, etc.
X = vec.fit_transform(input)
Feature Vectors
Machine Learning Algorithm
clf.fit(X, y)
Labels
X_new = vec.transform(input)
Feature Vector
New Text, Document, Image, etc.
Predictive Model
y_new = clf.predict(X_new)
Expected Label

- 
Unsupervised Learning Model
Training Text, Documents, Images, etc.
Feature Vectors
Machine Learning Algorithm
New Text, Document, Image, etc.
Feature Vector
Predictive Model
Likelihood or Cluster ID or Better Representation
- 
Supervised Learning Model
Training Text, Documents, Images, etc.
Feature Vectors
Labels
Machine Learning Algorithm
New Text, Document, Image, etc.
Feature Vector
Predictive Model
Expected Label

Python source code: `plot_ML_flow_chart.py`

```
import numpy as np
import pylab as pl
from matplotlib.patches import Circle, Rectangle, Polygon, Arrow, FancyArrow

def create_base(box_bg = '#CCCCCC',
                arrow1 = '#88CCFF',
                arrow2 = '#88FF88',
                supervised=True):
    fig = pl.figure(figsize=(9, 6), facecolor='w')
    ax = pl.axes((0, 0, 1, 1),
                 xticks=[], yticks=[], frameon=False)
    ax.set_xlim(0, 9)
    ax.set_ylim(0, 6)

    patches = [Rectangle((0.3, 3.6), 1.5, 1.8, zorder=1, fc=box_bg),
               Rectangle((0.5, 3.8), 1.5, 1.8, zorder=2, fc=box_bg),
               Rectangle((0.7, 4.0), 1.5, 1.8, zorder=3, fc=box_bg),

               Rectangle((2.9, 3.6), 0.2, 1.8, fc=box_bg),
               Rectangle((3.1, 3.8), 0.2, 1.8, fc=box_bg),
               Rectangle((3.3, 4.0), 0.2, 1.8, fc=box_bg),

               Rectangle((0.3, 0.2), 1.5, 1.8, fc=box_bg),

               Rectangle((2.9, 0.2), 0.2, 1.8, fc=box_bg),

               Circle((5.5, 3.5), 1.0, fc=box_bg),

               Polygon([[5.5, 1.7],
                        [6.1, 1.1],
                        [5.5, 0.5],
                        [4.9, 1.1]], fc=box_bg),

               FancyArrow(2.3, 4.6, 0.35, 0, fc=arrow1,
                          width=0.25, head_width=0.5, head_length=0.2),

               FancyArrow(3.75, 4.2, 0.5, -0.2, fc=arrow1,
                          width=0.25, head_width=0.5, head_length=0.2),

               FancyArrow(5.5, 2.4, 0, -0.4, fc=arrow1,
                          width=0.25, head_width=0.5, head_length=0.2),

               FancyArrow(2.0, 1.1, 0.5, 0, fc=arrow2,
                          width=0.25, head_width=0.5, head_length=0.2),

               FancyArrow(3.3, 1.1, 1.3, 0, fc=arrow2,
                          width=0.25, head_width=0.5, head_length=0.2),

               FancyArrow(6.2, 1.1, 0.8, 0, fc=arrow2,
                          width=0.25, head_width=0.5, head_length=0.2)]

    if supervised:
        patches += [Rectangle((0.3, 2.4), 1.5, 0.5, zorder=1, fc=box_bg),
                    Rectangle((0.5, 2.6), 1.5, 0.5, zorder=2, fc=box_bg),
                    Rectangle((0.7, 2.8), 1.5, 0.5, zorder=3, fc=box_bg),
                    FancyArrow(2.3, 2.9, 2.0, 0, fc=arrow1,
                               width=0.25, head_width=0.5, head_length=0.2),
                    Rectangle((7.3, 0.85), 1.5, 0.5, fc=box_bg)]
    else:
        patches += [Rectangle((7.3, 0.2), 1.5, 1.8, fc=box_bg)]

    for p in patches:
        ax.add_patch(p)

    pl.text(1.45, 4.9, "Training\nText,\nDocuments,\nImages,\netc.",
            ha='center', va='center', fontsize=14)

    pl.text(3.6, 4.9, "Feature\nVectors",
            ha='left', va='center', fontsize=14)

    pl.text(5.5, 3.5, "Machine\nLearning\nAlgorithm",
            ha='center', va='center', fontsize=14)

    pl.text(1.05, 1.1, "New Text,\nDocument,\nImage,\netc.",
            ha='center', va='center', fontsize=14)

    pl.text(3.3, 1.7, "Feature\nVector",
            ha='left', va='center', fontsize=14)

    pl.text(5.5, 1.1, "Predictive\nModel",
            ha='center', va='center', fontsize=12)

    if supervised:
        pl.text(1.45, 3.05, "Labels",
                ha='center', va='center', fontsize=14)

        pl.text(8.05, 1.1, "Expected\nLabel",
                ha='center', va='center', fontsize=14)
```

```
        pl.text(8.8, 5.8, "Supervised Learning Model",
                ha='right', va='top', fontsize=18)

    else:
        pl.text(8.05, 1.1,
                "Likelihood\nor Cluster ID\nor Better\nRepresentation",
                ha='center', va='center', fontsize=12)
        pl.text(8.8, 5.8, "Unsupervised Learning Model",
                ha='right', va='top', fontsize=18)



def plot_supervised(annotate=False):
    create_base(supervised=True)
    if annotate:
        fontdict = dict(color='r', weight='bold', size=14)
        pl.text(1.9, 4.55, 'X = vec.fit_transform(input)',
                fontdict=fontdict,
                rotation=20, ha='left', va='bottom')
        pl.text(3.7, 3.2, 'clf.fit(X, y)',
                fontdict=fontdict,
                rotation=20, ha='left', va='bottom')
        pl.text(1.7, 1.5, 'X_new = vec.transform(input)',
                fontdict=fontdict,
                rotation=20, ha='left', va='bottom')
        pl.text(6.1, 1.5, 'y_new = clf.predict(X_new)',
                fontdict=fontdict,
                rotation=20, ha='left', va='bottom')

def plot_unsupervised():
    create_base(supervised=False)


plot_supervised(False)
plot_supervised(True)
plot_unsupervised()
pl.show()
```

Total running time of the example: 1 seconds

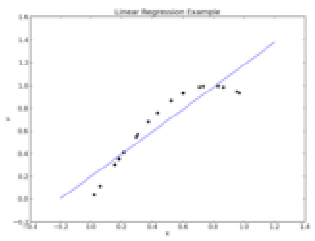
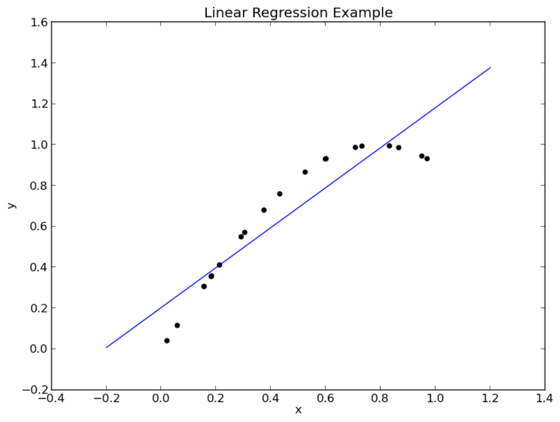
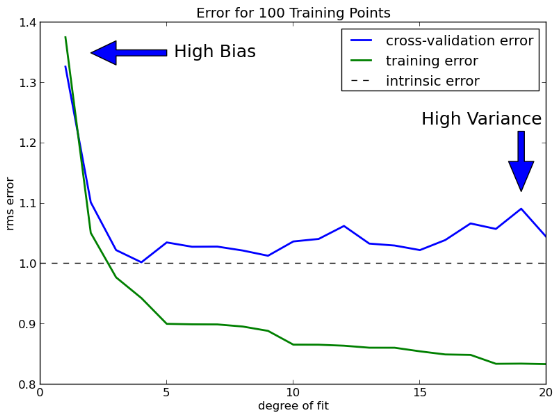


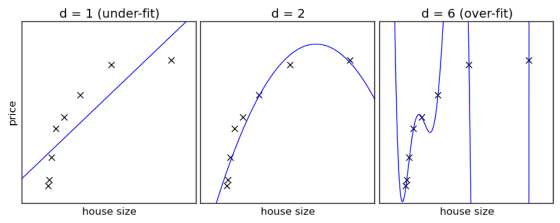
Figure 8.2: *Bias and Variance* (page **??**)

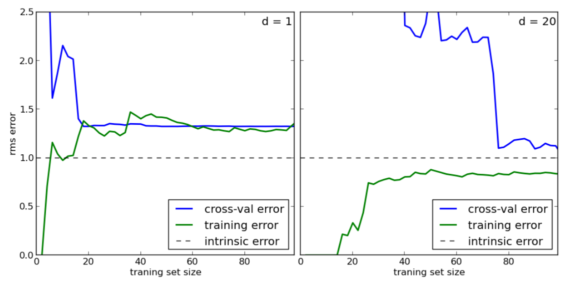
8.2 Bias and Variance

This script plots some simple examples of how model complexity affects bias and variance.

- 

- 

- 

- 

Python source code: `plot_bias_variance_examples.py`

```
import numpy as np
import pylab as pl
from matplotlib import ticker
from matplotlib.patches import FancyArrow

# Suppress warnings from Polyfit (ill-conditioned fit)
import warnings
warnings.filterwarnings('ignore', message='Polyfit*')
```

```
np.random.seed(42)

def test_func(x, err=0.5):
    return np.random.normal(10 - 1. / (x + 0.1), err)


def compute_error(x, y, p):
    yfit = np.polyval(p, x)
    return np.sqrt(np.mean((y - yfit) ** 2))


#------------------------------------------------------------
# Plot linear regression example
np.random.seed(42)
x = np.random.random(20)
y = np.sin(2 * x)
p = np.polyfit(x, y, 1)  # fit a 1st-degree polynomial to the data

xfit = np.linspace(-0.2, 1.2, 10)
yfit = np.polyval(p, xfit)

pl.scatter(x, y, c='k')
pl.plot(xfit, yfit)
pl.xlabel('x')
pl.ylabel('y')
pl.title('Linear Regression Example')

#------------------------------------------------------------
# Plot example of over-fitting and under-fitting

N = 8
np.random.seed(42)
x = 10 ** np.linspace(-2, 0, N)
y = test_func(x)

xfit = np.linspace(-0.2, 1.2, 1000)

titles = ['d = 1 (under-fit)', 'd = 2', 'd = 6 (over-fit)']
degrees = [1, 2, 6]

pl.figure(figsize = (9, 3.5))
for i, d in enumerate(degrees):
    pl.subplot(131 + i, xticks=[], yticks=[])
    pl.scatter(x, y, marker='x', c='k', s=50)

    p = np.polyfit(x, y, d)
    yfit = np.polyval(p, xfit)
    pl.plot(xfit, yfit, '-b')

    pl.xlim(-0.2, 1.2)
    pl.ylim(0, 12)
    pl.xlabel('house size')
    if i == 0:
        pl.ylabel('price')

    pl.title(titles[i])

pl.subplots_adjust(left = 0.06, right=0.98,
                   bottom=0.15, top=0.85,
                   wspace=0.05)

#------------------------------------------------------------
# Plot training error and cross-val error
#   as a function of polynomial degree

Ntrain = 100
Ncrossval = 100
error = 1.0

np.random.seed(0)
x = np.random.random(Ntrain + Ncrossval)
y = test_func(x, error)

xtrain = x[:Ntrain]
ytrain = y[:Ntrain]

xcrossval = x[Ntrain:]
ycrossval = y[Ntrain:]

degrees = np.arange(1, 21)
train_err = np.zeros(len(degrees))
crossval_err = np.zeros(len(degrees))

for i, d in enumerate(degrees):
    p = np.polyfit(xtrain, ytrain, d)

    train_err[i] = compute_error(xtrain, ytrain, p)
    crossval_err[i] = compute_error(xcrossval, ycrossval, p)

pl.figure()
pl.title('Error for 100 Training Points')
pl.plot(degrees, crossval_err, lw=2, label = 'cross-validation error')
pl.plot(degrees, train_err, lw=2, label = 'training error')
pl.plot([0, 20], [error, error], '--k', label='intrinsic error')
pl.legend()
pl.xlabel('degree of fit')
pl.ylabel('rms error')

pl.gca().add_patch(FancyArrow(5, 1.35, -3, 0, width = 0.01,
                              head_width=0.04, head_length=1.0,
                              length_includes_head=True))
pl.text(5.3, 1.35, "High Bias", fontsize=18, va='center')

pl.gca().add_patch(FancyArrow(19, 1.22, 0, -0.1, width = 0.25,
                              head_width=1.0, head_length=0.05,
                              length_includes_head=True))
pl.text(19.8, 1.23, "High Variance", ha='right', fontsize=18)

#------------------------------------------------------------
# Plot training error and cross-val error
#   as a function of training set size

Ntrain = 100
Ncrossval = 100
error = 1.0
```

```
np.random.seed(0)
x = np.random.random(Ntrain + Ncrossval)
y = test_func(x, error)

xtrain = x[:Ntrain]
ytrain = y[:Ntrain]

xcrossval = x[Ntrain:]
ycrossval = y[Ntrain:]

sizes = np.linspace(2, Ntrain, 50).astype(int)
train_err = np.zeros(sizes.shape)
crossval_err = np.zeros(sizes.shape)

pl.figure(figsize=(10, 5))

for j,d in enumerate((1, 20)):
    for i, size in enumerate(sizes):
        p = np.polyfit(xtrain[:size], ytrain[:size], d)
        crossval_err[i] = compute_error(xcrossval, ycrossval, p)
        train_err[i] = compute_error(xtrain[:size], ytrain[:size], p)

    ax = pl.subplot(121 + j)
    pl.plot(sizes, crossval_err, lw=2, label='cross-val error')
    pl.plot(sizes, train_err, lw=2, label='training error')
    pl.plot([0, Ntrain], [error, error], '--k', label='intrinsic error')

    pl.xlabel('traning set size')
    if j == 0:
        pl.ylabel('rms error')
    else:
        ax.yaxis.set_major_formatter(ticker.NullFormatter())

    pl.legend(loc = 4)

    pl.ylim(0.0, 2.5)
    pl.xlim(0, 99)

    pl.text(98, 2.45, 'd = %i' % d, ha='right', va='top', fontsize='large')

pl.subplots_adjust(wspace = 0.02, left=0.07, right=0.95,
                   bottom=0.1, top=0.9)
pl.show()
```

Total running time of the example: 3 seconds

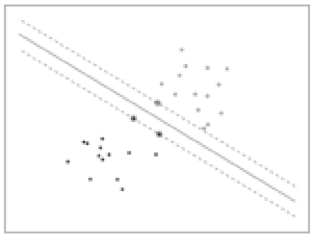
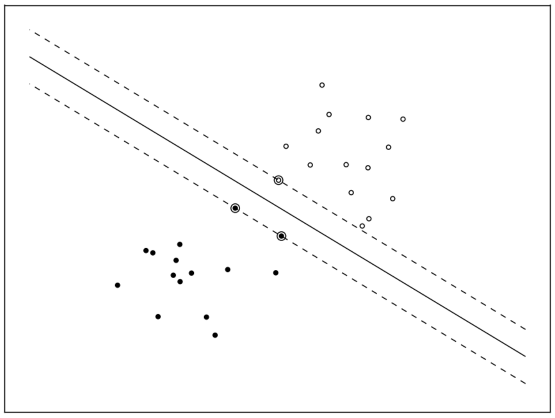
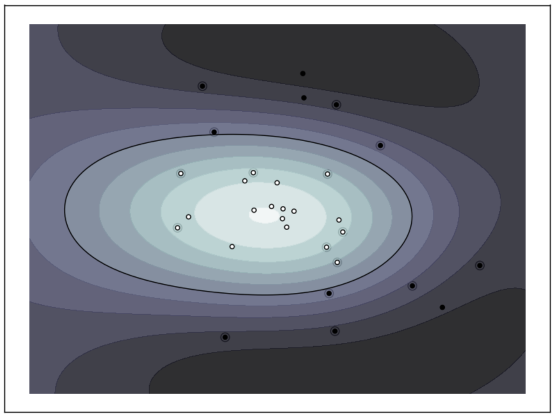


Figure 8.3: *Linear Model Example* (page **??**)

8.3 Linear Model Example

This is an example plot from the tutorial which accompanies an explanation of the support vector machine GUI.

- 
- 

Python source code: `plot_gui_example.py`

```
import numpy as np
import pylab as pl
import matplotlib

from sklearn import svm


def linear_model(rseed=42, Npts=30):
    np.random.seed(rseed)


    data = np.random.normal(0, 10, (Npts, 2))
    data[:Npts / 2] -= 15
    data[Npts / 2:] += 15

    labels = np.ones(Npts)
    labels[:Npts / 2] = -1

    return data, labels


def nonlinear_model(rseed=42, Npts=30):
    radius = 40 * np.random.random(Npts)
    far_pts = radius > 20
    radius[far_pts] *= 1.2
    radius[~far_pts] *= 1.1

    theta = np.random.random(Npts) * np.pi * 2

    data = np.empty((Npts, 2))
    data[:, 0] = radius * np.cos(theta)
    data[:, 1] = radius * np.sin(theta)

    labels = np.ones(Npts)
    labels[far_pts] = -1

    return data, labels

#------------------------------------------------------------
# Linear model
X, y = linear_model()
clf = svm.SVC(kernel='linear',
              gamma=0.01, coef0=0, degree=3)
clf.fit(X, y)

fig = pl.figure()
ax = pl.subplot(111, xticks=[], yticks=[])
ax.scatter(X[:, 0], X[:, 1], c=y, cmap=pl.cm.bone)

ax.scatter(clf.support_vectors_[:, 0],
           clf.support_vectors_[:, 1],
           s=80, edgecolors="k", facecolors="none")

delta = 1
y_min, y_max = -50, 50
x_min, x_max = -50, 50
x = np.arange(x_min, x_max + delta, delta)
y = np.arange(y_min, y_max + delta, delta)
X1, X2 = np.meshgrid(x, y)
Z = clf.decision_function(np.c_[X1.ravel(), X2.ravel()])
Z = Z.reshape(X1.shape)

levels = [-1.0, 0.0, 1.0]
linestyles = ['dashed', 'solid', 'dashed']
colors = 'k'
ax.contour(X1, X2, Z, levels,
           colors=colors,
           linestyles=linestyles)


#------------------------------------------------------------
# RBF model
X, y = nonlinear_model()
clf = svm.SVC(kernel='rbf',
              gamma=0.001, coef0=0, degree=3)
clf.fit(X, y)

fig = pl.figure()
ax = pl.subplot(111, xticks=[], yticks=[])
ax.scatter(X[:, 0], X[:, 1], c=y, cmap=pl.cm.bone, zorder=2)
ax.scatter(clf.support_vectors_[:, 0],
           clf.support_vectors_[:, 1],
           s=80, edgecolors="k", facecolors="none")

delta = 1
y_min, y_max = -50, 50
x_min, x_max = -50, 50
x = np.arange(x_min, x_max + delta, delta)
y = np.arange(y_min, y_max + delta, delta)
X1, X2 = np.meshgrid(x, y)
Z = clf.decision_function(np.c_[X1.ravel(), X2.ravel()])
Z = Z.reshape(X1.shape)

levels = [-1.0, 0.0, 1.0]
linestyles = ['dashed', 'solid', 'dashed']
colors = 'k'

ax.contourf(X1, X2, Z, 10,
            cmap=matplotlib.cm.bone,
            origin='lower',
            alpha=0.85, zorder=1)
ax.contour(X1, X2, Z, [0.0],
           colors='k',
           linestyles=['solid'], zorder=1)

pl.show()
```

Total running time of the example: 2 seconds

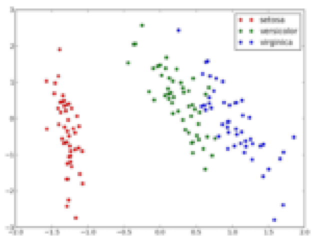
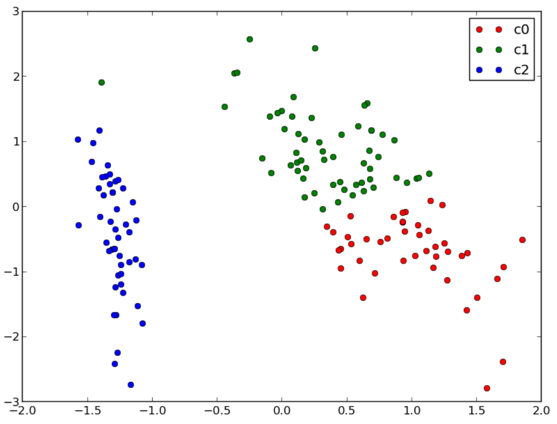
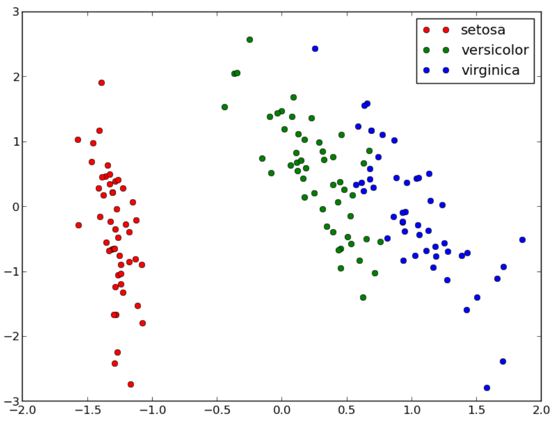


Figure 8.4: *Iris Projections* (page **??**)

8.4 Iris Projections

This code generates the Iris projection example plots found in the tutorial

- 

- 

Python source code: `plot_iris_projections.py`

```
from itertools import cycle
import pylab as pl

from sklearn.datasets import load_iris
from sklearn.decomposition import PCA


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.plot(data[target == i, 0],
                data[target == i, 1], 'o',
                c=c, label=label)
    pl.legend(target_names)

#----------------------------------------------------------------------
# Load iris data
iris = load_iris()
X, y = iris.data, iris.target


#----------------------------------------------------------------------
# First figure: PCA
pca = PCA(n_components=2, whiten=True).fit(X)
X_pca = pca.transform(X)
```

```
plot_2D(X_pca, iris.target, iris.target_names)


#-----------------------------------------------------------------------
# Second figure: Kmeans labels
from sklearn.cluster import KMeans
from numpy.random import RandomState
rng = RandomState(42)
kmeans = KMeans(3, random_state=rng).fit(X_pca)
plot_2D(X_pca, kmeans.labels_, ["c0", "c1", "c2"])


pl.show()
```

Total running time of the example: 1 seconds

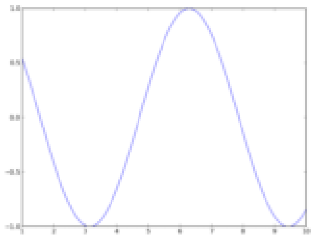
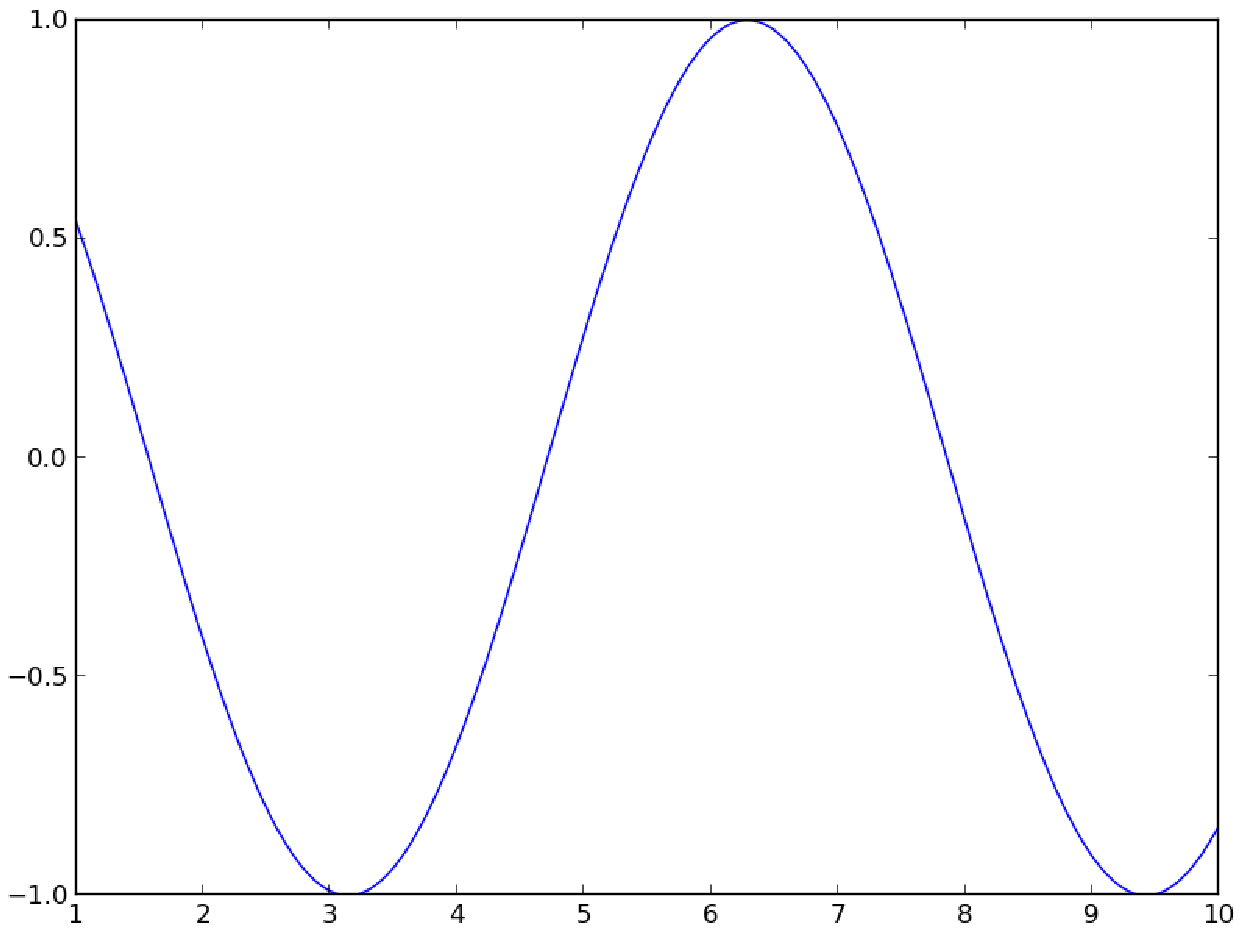


Figure 8.5: *Basic numerics and plotting with Python* (page **??**)

8.5 Basic numerics and plotting with Python



Python source code: `plot_python_101.py`

```
# import numpy: the module providing numerical arrays
import numpy as np
t = np.linspace(1, 10, 2000)

# import pylab: the module for scientific plotting
import pylab as pl
pl.plot(t, np.cos(t))
```

Total running time of the example: 0 seconds

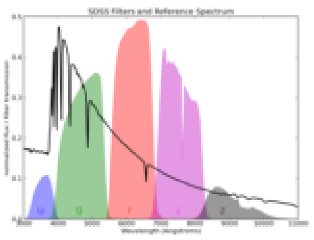
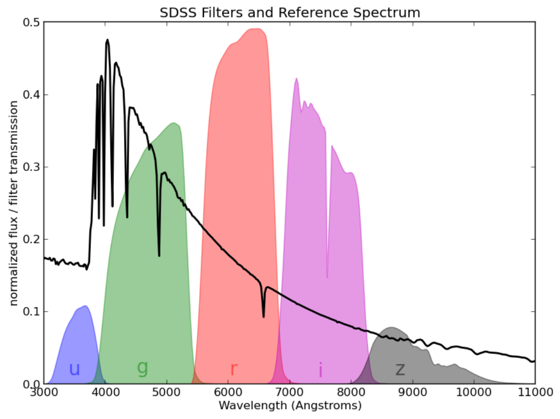
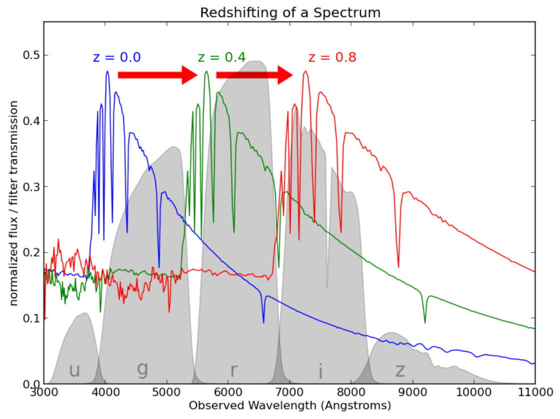


Figure 8.6: *SDSS Filters* (page **??**)

8.6 SDSS Filters

This example downloads and plots the filters from the Sloan Digital Sky Survey, along with a reference spectrum.

- 

- 

Python source code: `plot_sdss_filters.py`

```
import os
import urllib2

import numpy as np
import pylab as pl
from matplotlib.patches import Arrow

REFSPEC_URL = 'ftp://ftp.stsci.edu/cdbs/current_calspec/1732526_nic_002.ascii'
URL = 'http://www.sdss.org/dr7/instruments/imager/filters/%s.dat'

def fetch_filter(filt):
    assert filt in 'ugriz'
    url = URL % filt

    if not os.path.exists('downloads'):
        os.makedirs('downloads')

    loc = os.path.join('downloads', '%s.dat' % filt)
    if not os.path.exists(loc):
        print "downloading from %s" % url
        F = urllib2.urlopen(url)
        open(loc, 'w').write(F.read())
    F = open(loc)

    data = np.loadtxt(F)
    return data

def fetch_vega_spectrum():
    if not os.path.exists('downloads'):
        os.makedirs('downloads')

    refspec_file = os.path.join('downloads', REFSPEC_URL.split('/')[-1])

    if  not os.path.exists(refspec_file):
        print "downloading from %s" % REFSPEC_URL
        F = urllib2.urlopen(REFSPEC_URL)
        open(refspec_file, 'w').write(F.read())

    F = open(refspec_file)

    data = np.loadtxt(F)
    return data


Xref = fetch_vega_spectrum()
Xref[:, 1] /= 2.1 * Xref[:, 1].max()

#----------------------------------------------------------------------
# Plot filters in color with a single spectrum
pl.figure()
pl.plot(Xref[:, 0], Xref[:, 1], '-k', lw=2)

for f,c in zip('ugriz', 'bgrmk'):
    X = fetch_filter(f)
    pl.fill(X[:, 0], X[:, 1], ec=c, fc=c, alpha=0.4)

kwargs = dict(fontsize=20, ha='center', va='center', alpha=0.5)
pl.text(3500, 0.02, 'u', color='b', **kwargs)
pl.text(4600, 0.02, 'g', color='g', **kwargs)
pl.text(6100, 0.02, 'r', color='r', **kwargs)
pl.text(7500, 0.02, 'i', color='m', **kwargs)
pl.text(8800, 0.02, 'z', color='k', **kwargs)

pl.xlim(3000, 11000)

pl.title('SDSS Filters and Reference Spectrum')
pl.xlabel('Wavelength (Angstroms)')
pl.ylabel('normalized flux / filter transmission')

#----------------------------------------------------------------------
# Plot filters in gray with several redshifted spectra
pl.figure()

redshifts = [0.0, 0.4, 0.8]
colors = 'bgr'

for z, c in zip(redshifts, colors):
    pl.plot((1. + z) * Xref[:, 0], Xref[:, 1], color=c)

pl.gca().add_patch(Arrow(4200, 0.47, 1300, 0, lw=0, width=0.05, color='r'))
pl.gca().add_patch(Arrow(5800, 0.47, 1250, 0, lw=0, width=0.05, color='r'))

pl.text(3800, 0.49, 'z = 0.0', fontsize=14, color=colors[0])
pl.text(5500, 0.49, 'z = 0.4', fontsize=14, color=colors[1])
pl.text(7300, 0.49, 'z = 0.8', fontsize=14, color=colors[2])

for f in 'ugriz':
    X = fetch_filter(f)
    pl.fill(X[:, 0], X[:, 1], ec='k', fc='k', alpha=0.2)

kwargs = dict(fontsize=20, color='gray', ha='center', va='center')
pl.text(3500, 0.02, 'u', **kwargs)
pl.text(4600, 0.02, 'g', **kwargs)
pl.text(6100, 0.02, 'r', **kwargs)
pl.text(7500, 0.02, 'i', **kwargs)
pl.text(8800, 0.02, 'z', **kwargs)

pl.xlim(3000, 11000)
pl.ylim(0, 0.55)

pl.title('Redshifting of a Spectrum')
pl.xlabel('Observed Wavelength (Angstroms)')
pl.ylabel('normalized flux / filter transmission')

pl.show()
```

Total running time of the example: 1 seconds

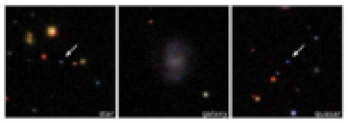
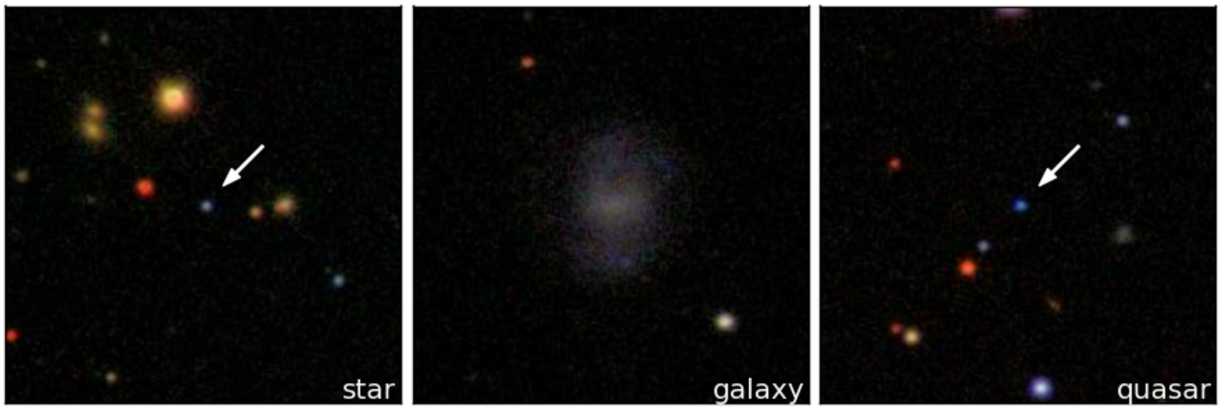


Figure 8.7: *SDSS Images* (page **??**)

8.7 SDSS Images

This script plots an example quasar, star, and galaxy image for use in the tutorial.



Python source code: `plot_sdss_images.py`

```
import os
import urllib2

import pylab as pl
from matplotlib import image

def _fetch(outfile, RA, DEC, scale=0.2, width=400, height=400):
    """Fetch the image at the given RA, DEC from the SDSS server"""
    url = ("http://casjobs.sdss.org/ImgCutoutDR7/"
           "getjpeg.aspx?ra=%.8f&dec=%.8f&scale=%.2f&width=%i&height=%i"
           % (RA, DEC, scale, width, height))
    print "downloading %s" % url
    print " -> %s" % outfile
    fhandle = urllib2.urlopen(url)
    open(outfile, 'w').write(fhandle.read())


def fetch_image(object_type):
    """Return the data array for the image of object type"""
    if not os.path.exists('downloads'):
        os.makedirs('downloads')

    filename = os.path.join('downloads', '%s_image.jpg' % object_type)
    if not os.path.exists(filename):
        RA = image_locations[object_type]['RA']
        DEC = image_locations[object_type]['DEC']
        _fetch(filename, RA, DEC)

    return image.imread(filename)


image_locations = dict(star=dict(RA=180.63040108,
                                 DEC=64.96767375),
                       galaxy=dict(RA=197.51943983,
                                   DEC=0.94881436),
                       quasar=dict(RA=226.18451462,
                                   DEC=4.07456639))


# Plot the images
fig = pl.figure(figsize=(9, 3))

# Check that PIL is installed for jpg support
if 'jpg' not in fig.canvas.get_supported_filetypes():
    raise ValueError("PIL required to load SDSS jpeg images")

object_types = ['star', 'galaxy', 'quasar']

for i, object_type in enumerate(object_types):
    ax = pl.subplot(131 + i, xticks=[], yticks=[])
    I = fetch_image(object_type)
    ax.imshow(I)
    if object_type != 'galaxy':
        pl.arrow(0.65, 0.65, -0.1, -0.1, width=0.005, head_width=0.03,
                 length_includes_head=True,
                 color='w', transform=ax.transAxes)
    pl.text(0.99, 0.01, object_type, fontsize='large', color='w', ha='right',
            transform=ax.transAxes)
```

```
pl.subplots_adjust(bottom=0.04, top=0.94, left=0.02, right=0.98, wspace=0.04)

pl.show()
```

Total running time of the example: 1 seconds

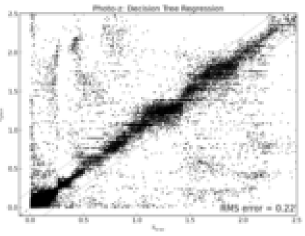
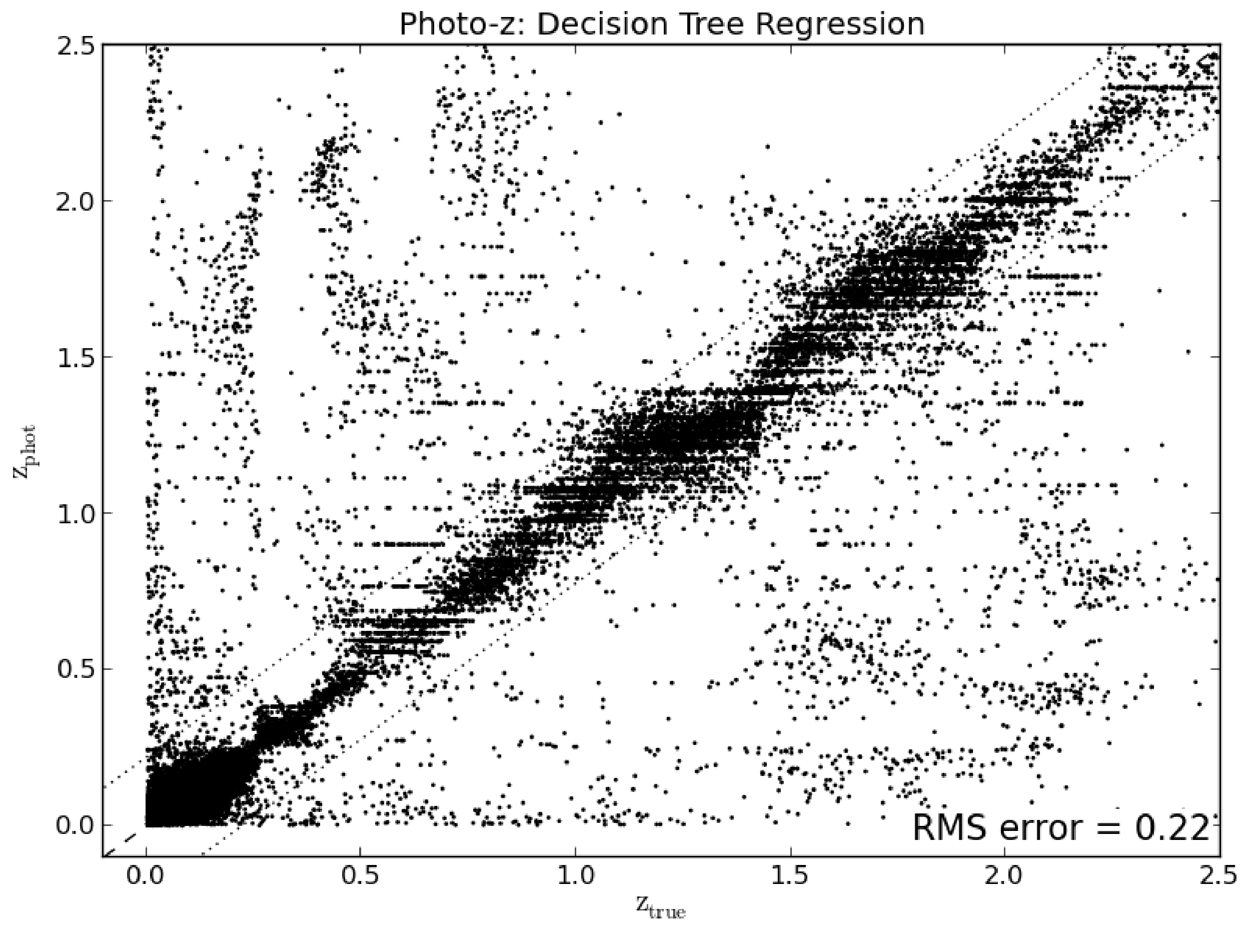


Figure 8.8: *SDSS Photometric Redshifts* (page **??**)

8.8 SDSS Photometric Redshifts

This example shows how a Decision tree can be used to learn redshifts of galaxies in the Sloan Digital Sky Survey.



Python source code: `plot_sdss_photoz.py`

```
import os
import urllib2
import numpy as np
import pylab as pl

from sklearn.datasets import get_data_home
from sklearn.tree import DecisionTreeRegressor

DATA_URL = ('http://www.astro.washington.edu/users/'
            'vanderplas/pydata/sdss_photoz.npy')
LOCAL_FILE = 'sdss_photoz.npy'

def fetch_photoz_data():
    if not os.path.exists('downloads'):
        os.makedirs('downloads')

    local_file = os.path.join('downloads', LOCAL_FILE)

    if not os.path.exists(local_file):
        # data directory is password protected so the public can't access it
        password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()
        password_mgr.add_password(None, DATA_URL, 'pydata', 'astroML')
        handler = urllib2.HTTPBasicAuthHandler(password_mgr)
        opener = urllib2.build_opener(handler)
        fhandle = opener.open(DATA_URL)
        open(local_file, 'w').write(fhandle.read())

    return np.load(local_file)

data = fetch_photoz_data()

N = len(data)

# put colors in a matrix
X = np.zeros((N, 4))
X[:, 0] = data['u'] - data['g']
X[:, 1] = data['g'] - data['r']
X[:, 2] = data['r'] - data['i']
X[:, 3] = data['i'] - data['z']
z = data['redshift']

# divide into training and testing data
Ntrain = 3 * N / 4
Xtrain = X[:Ntrain]
ztrain = z[:Ntrain]

Xtest = X[Ntrain:]
ztest = z[Ntrain:]


clf = DecisionTreeRegressor(max_depth=20)
clf.fit(Xtrain, ztrain)
zpred = clf.predict(Xtest)

axis_lim = np.array([-0.1, 2.5])

rms = np.sqrt(np.mean((ztest - zpred) ** 2))
print rms
print len(ztest)
print np.sum(abs(ztest - zpred) > 1)

ax = pl.axes()
pl.scatter(ztest, zpred, c='k', lw=0, s=4)
pl.plot(axis_lim, axis_lim, '--k')
pl.plot(axis_lim, axis_lim + rms, ':k')
pl.plot(axis_lim, axis_lim - rms, ':k')
pl.xlim(axis_lim)
pl.ylim(axis_lim)

pl.text(0.99, 0.02, "RMS error = %.2g" % rms,
        ha='right', va='bottom', transform=ax.transAxes,
        bbox=dict(ec='w', fc='w'), fontsize=16)

pl.title('Photo-z: Decision Tree Regression')
pl.xlabel(r'$\mathrm{z_{true}}$', fontsize=14)
pl.ylabel(r'$\mathrm{z_{phot}}$', fontsize=14)
pl.show()
```

Script output:

```
0.221409443127
102798
1538
```

Total running time of the example: 46 seconds

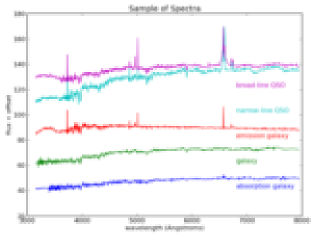
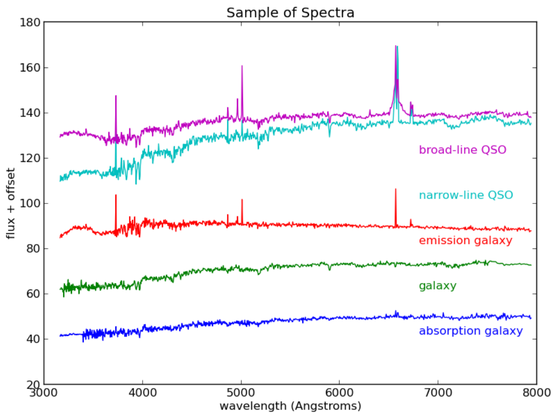
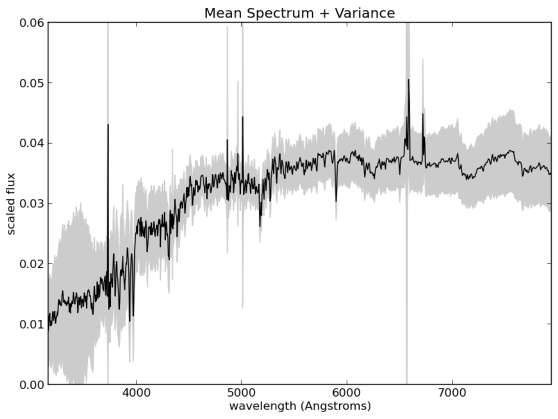


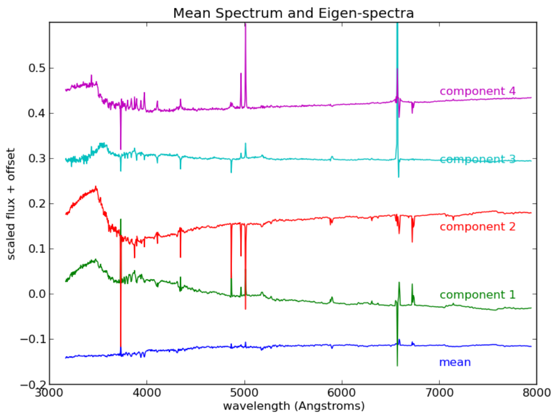
Figure 8.9: *SDSS Spectra Plots* (page **??**)

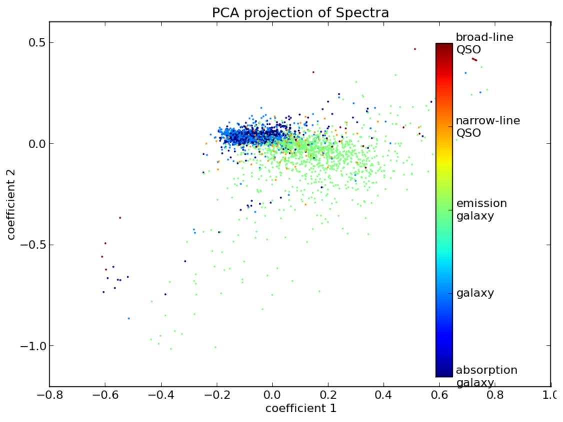
8.9 SDSS Spectra Plots

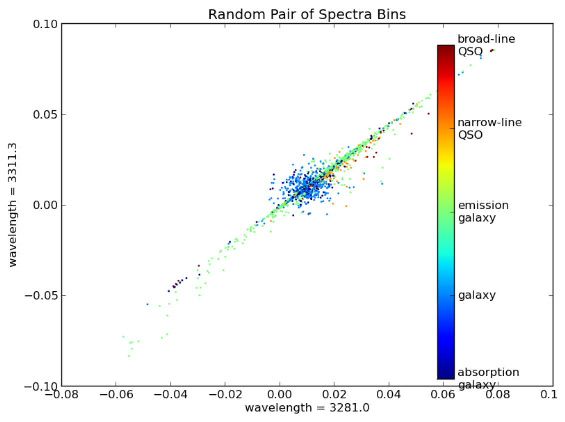
This plots some of the SDSS spectra examples for the astronomy tutorial

- 
Sample of Spectra
flux + offset
wavelength (Angstroms)
broad-line QSO
narrow-line QSO
emission galaxy
galaxy
absorption galaxy

- 
Mean Spectrum + Variance
scaled flux
wavelength (Angstroms)

- 
Mean Spectrum and Eigen-spectra
scaled flux + offset
wavelength (Angstroms)
component 4
component 3
component 2
component 1
mean

- 
PCA projection of Spectra
coefficient 1
coefficient 2
broad-line QSO
narrow-line QSO
emission galaxy
galaxy
absorption galaxy

- 

Python source code: `plot_sdss_specPCA.py`

```
import os
import urllib2

import numpy as np
import pylab as pl

from sklearn import preprocessing
from sklearn.decomposition import RandomizedPCA

DATA_URL = ('http://www.astro.washington.edu/users/'
            'vanderplas/pydata/spec4000_corrected.npz')

def fetch_sdss_spec_data():
    if not os.path.exists('downloads'):
        os.makedirs('downloads')

    local_file = os.path.join('downloads', os.path.basename(DATA_URL))

    # data directory is password protected so the public can't access it
    password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()
    password_mgr.add_password(None, DATA_URL, 'pydata', 'astroML')
    handler = urllib2.HTTPBasicAuthHandler(password_mgr)
    opener = urllib2.build_opener(handler)

    # download training data
    if not os.path.exists(local_file):
        fhandle = opener.open(DATA_URL)
        open(local_file, 'w').write(fhandle.read())

    return np.load(local_file)

#----------------------------------------------------------------------
#
#  Load the data
data = fetch_sdss_spec_data()

wavelengths = data['wavelengths']
X = data['X']
y = data['y']
labels = data['labels']

from matplotlib.ticker import FuncFormatter
```

```
format = FuncFormatter(lambda i, *args: labels[i].replace(' ', '\n'))

#------------------------------------------------------------
#
#  Plot the first few spectra, offset so they don't overlap
#
pl.figure()

for i_class in (2, 3, 4, 5, 6):
    i = np.where(y == i_class)[0][0]
    l = pl.plot(wavelengths, X[i] + 20 * i_class)
    c = l[0].get_color()
    pl.text(6800, 2 + 20 * i_class, labels[i_class], color=c)

pl.subplots_adjust(hspace=0)
pl.xlabel('wavelength (Angstroms)')
pl.ylabel('flux + offset')
pl.title('Sample of Spectra')

#------------------------------------------------------------
#
#  Plot the mean spectrum
#
X = preprocessing.normalize(X, 'l2')

pl.figure()

mu = X.mean(0)
std = X.std(0)

pl.plot(wavelengths, mu, color='black')
pl.fill_between(wavelengths, mu - std, mu + std, color='#CCCCCC')
pl.xlim(wavelengths[0], wavelengths[-1])
pl.ylim(0, 0.06)
pl.xlabel('wavelength (Angstroms)')
pl.ylabel('scaled flux')
pl.title('Mean Spectrum + Variance')

#------------------------------------------------------------
#
#  Plot a random pair of digits
#
pl.figure()
np.random.seed(25255)
i1, i2 = np.random.randint(1000, size=2)

pl.scatter(X[:, i1], X[:, i2], c=y, s=4, lw=0,
           vmin=2, vmax=6, cmap=pl.cm.jet)
pl.colorbar(ticks = range(2, 7), format=format)
pl.xlabel('wavelength = %.1f' % wavelengths[i1])
pl.ylabel('wavelength = %.1f' % wavelengths[i2])
pl.title('Random Pair of Spectra Bins')

#------------------------------------------------------------
#
#  Perform PCA
#
```

```
rpca = RandomizedPCA(n_components=4, random_state=0)
X_proj = rpca.fit_transform(X)

#----------------------------------------------------------------------
#
#  Plot PCA components
#

pl.figure()
pl.scatter(X_proj[:, 0], X_proj[:, 1], c=y, s=4, lw=0,
           vmin=2, vmax=6, cmap=pl.cm.jet)
pl.colorbar(ticks = range(2, 7), format=format)
pl.xlabel('coefficient 1')
pl.ylabel('coefficient 2')
pl.title('PCA projection of Spectra')

#----------------------------------------------------------------------
#
#  Plot PCA eigenspectra
#

pl.figure()

l = pl.plot(wavelengths, rpca.mean_ - 0.15)
c = l[0].get_color()
pl.text(7000, -0.16, "mean" % i, color=c)

for i in range(4):
    l = pl.plot(wavelengths, rpca.components_[i] + 0.15 * i)
    c = l[0].get_color()
    pl.text(7000, -0.01 + 0.15 * i, "component %i" % (i + 1), color=c)
pl.ylim(-0.2, 0.6)
pl.xlabel('wavelength (Angstroms)')
pl.ylabel('scaled flux + offset')
pl.title('Mean Spectrum and Eigen-spectra')

pl.show()
```

Total running time of the example: 8 seconds

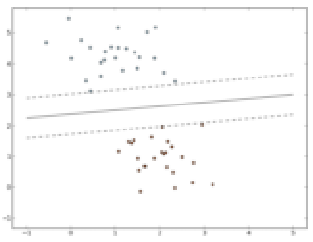
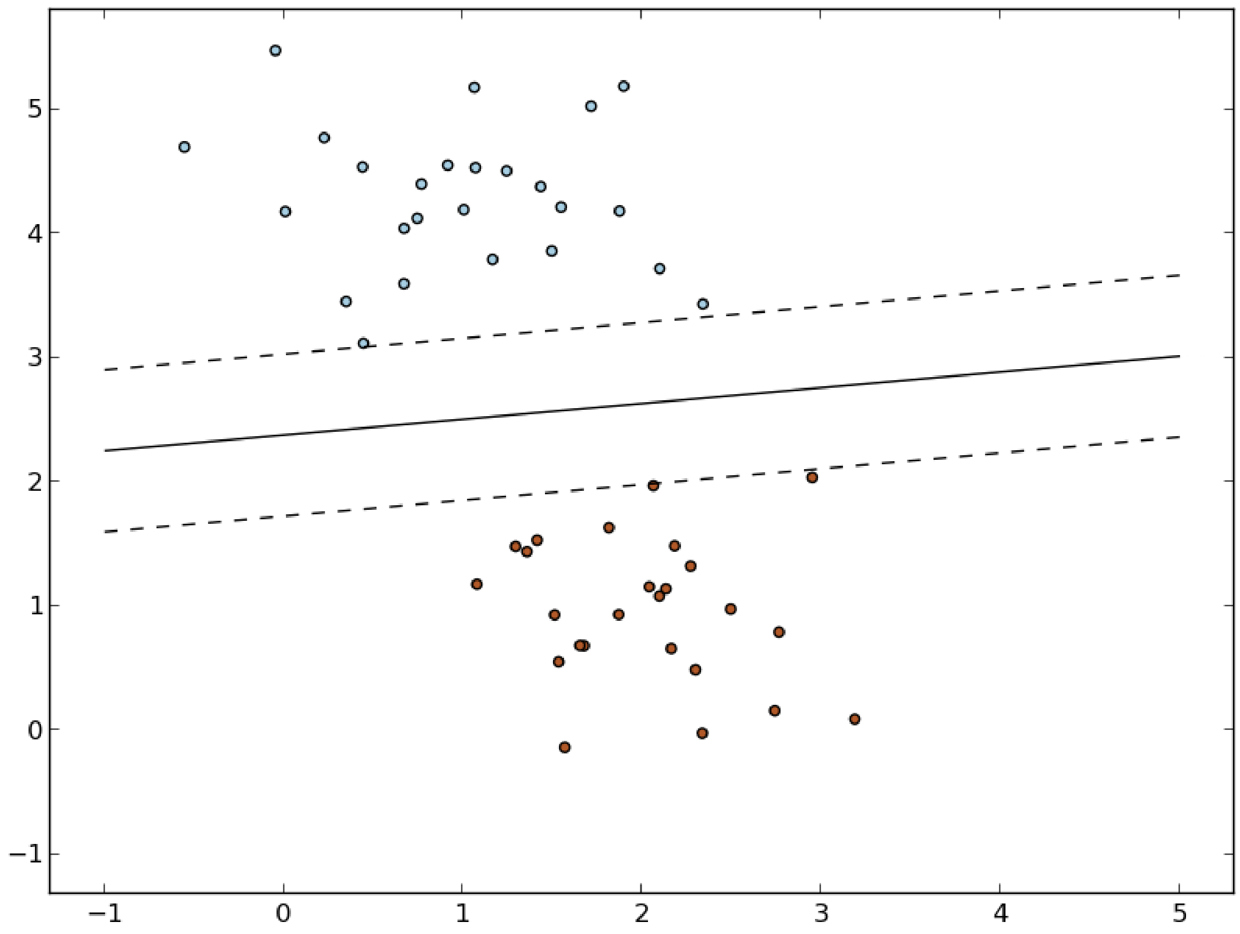


Figure 8.10: *SGD: Maximum margin separating hyperplane* (page **??**)

8.10 SGD: Maximum margin separating hyperplane

Plot the maximum margin separating hyperplane within a two-class separable dataset using a linear Support Vector Machines classifier trained using SGD.



Python source code: `plot_sgd_separating_hyperplane.py`

```
print __doc__

import numpy as np
import pylab as pl
from sklearn.linear_model import SGDClassifier
from sklearn.datasets.samples_generator import make_blobs

# we create 50 separable points
X, Y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=0.60)

# fit the model
clf = SGDClassifier(loss="hinge", alpha=0.01, n_iter=200, fit_intercept=True)
clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
xx = np.linspace(-1, 5, 10)
yy = np.linspace(-1, 5, 10)

X1, X2 = np.meshgrid(xx, yy)
Z = np.empty(X1.shape)
for (i, j), val in np.ndenumerate(X1):
    x1 = val
    x2 = X2[i, j]
    p = clf.decision_function([x1, x2])
    Z[i, j] = p[0]
levels = [-1.0, 0.0, 1.0]
linestyles = ['dashed', 'solid', 'dashed']
colors = 'k'
pl.contour(X1, X2, Z, levels, colors=colors, linestyles=linestyles)
pl.scatter(X[:, 0], X[:, 1], c=Y, cmap=pl.cm.Paired)

pl.axis('tight')
pl.show()
```

Total running time of the example: 0 seconds

Figure 8.11: *Libsvm GUI* (page **??**)

8.11 Libsvm GUI

A simple graphical frontend for Libsvm mainly intended for didactic purposes. You can create data points by point and click and visualize the decision region induced by different kernels and parameter settings.

To create positive examples click the left mouse button; to create negative examples click the right button.

If all examples are from the same class, it uses a one-class SVM.

Python source code: `svm_gui.py`

```
from __future__ import division

print __doc__

# Author: Peter Prettenhoer <peter.prettenhofer@gmail.com>
#
# License: BSD Style.

import matplotlib
matplotlib.use('TkAgg')

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.backends.backend_tkagg import NavigationToolbar2TkAgg
from matplotlib.figure import Figure
from matplotlib.contour import ContourSet

import Tkinter as Tk
import sys
import numpy as np

from sklearn import svm
from sklearn.datasets import dump_svmlight_file

y_min, y_max = -50, 50
x_min, x_max = -50, 50


class Model(object):
    """The Model which hold the data. It implements the
    observable in the observer pattern and notifies the
    registered observers on change event.
    """

    def __init__(self):
        self.observers = []
        self.surface = None
        self.data = []
        self.cls = None
        self.surface_type = 0

    def changed(self, event):
        """Notify the observers. """
        for observer in self.observers:
            observer.update(event, self)

    def add_observer(self, observer):
        """Register an observer. """
        self.observers.append(observer)

    def set_surface(self, surface):
        self.surface = surface

    def dump_svmlight_file(self, file):
        data = np.array(self.data)
        X = data[:, 0:2]
        y = data[:, 2]
        dump_svmlight_file(X, y, file)


class Controller(object):
    def __init__(self, model):
        self.model = model
        self.kernel = Tk.IntVar()
        self.surface_type = Tk.IntVar()
        # Whether or not a model has been fitted
        self.fitted = False

    def fit(self):
        print "fit the model"
        train = np.array(self.model.data)
        X = train[:, 0:2]
        y = train[:, 2]

        C = float(self.complexity.get())
        gamma = float(self.gamma.get())
        coef0 = float(self.coef0.get())
        degree = int(self.degree.get())
        kernel_map = {0: "linear", 1: "rbf", 2: "poly"}
        if len(np.unique(y)) == 1:
            clf = svm.OneClassSVM(kernel=kernel_map[self.kernel.get()],
                        gamma=gamma, coef0=coef0, degree=degree)
            clf.fit(X)
        else:
            clf = svm.SVC(kernel=kernel_map[self.kernel.get()], C=C,
                            gamma=gamma, coef0=coef0, degree=degree)
            clf.fit(X, y)
        if hasattr(clf, 'score'):
            print "Accuracy:", clf.score(X, y) * 100
        X1, X2, Z = self.decision_surface(clf)
        self.model.clf = clf
        self.model.set_surface((X1, X2, Z))
        self.model.surface_type = self.surface_type.get()
        self.fitted = True
        self.model.changed("surface")

    def decision_surface(self, cls):
        delta = 1
        x = np.arange(x_min, x_max + delta, delta)
        y = np.arange(y_min, y_max + delta, delta)
        X1, X2 = np.meshgrid(x, y)
        Z = cls.decision_function(np.c_[X1.ravel(), X2.ravel()])
        Z = Z.reshape(X1.shape)
        return X1, X2, Z

    def clear_data(self):
        self.model.data = []
        self.fitted = False
        self.model.changed("clear")

    def add_example(self, x, y, label):
        self.model.data.append((x, y, label))
        self.model.changed("example_added")

        # update decision surface if already fitted.
        self.refit()

    def refit(self):
        """Refit the model if already fitted. """
        if self.fitted:
            self.fit()


class View(object):
    """Test docstring. """
    def __init__(self, root, controller):
        f = Figure()
        ax = f.add_subplot(111)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_xlim((x_min, x_max))
        ax.set_ylim((y_min, y_max))
        canvas = FigureCanvasTkAgg(f, master=root)
        canvas.show()
        canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
        canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
        canvas.mpl_connect('button_press_event', self.onclick)
        toolbar = NavigationToolbar2TkAgg(canvas, root)
        toolbar.update()
        self.controllbar = ControllBar(root, controller)
        self.f = f
        self.ax = ax
        self.canvas = canvas
        self.controller = controller
        self.contours = []
        self.c_labels = None
        self.plot_kernels()

    def plot_kernels(self):
        self.ax.text(-50, -60, "Linear: $u^T v$")
        self.ax.text(-20, -60, "RBF: $\exp (-\gamma \| u-v \|^2)$")
        self.ax.text(10, -60, "Poly: $(\gamma \, u^T v + r)^d$")

    def onclick(self, event):
        if event.xdata and event.ydata:
            if event.button == 1:
                self.controller.add_example(event.xdata, event.ydata, 1)
            elif event.button == 3:
                self.controller.add_example(event.xdata, event.ydata, -1)

    def update_example(self, model, idx):
        x, y, l = model.data[idx]
        if l == 1:
            color = 'w'
        elif l == -1:
            color = 'k'
        self.ax.plot([x], [y], "%so" % color, scalex=0.0, scaley=0.0)

    def update(self, event, model):
        if event == "examples_loaded":
            for i in xrange(len(model.data)):
                self.update_example(model, i)

        if event == "example_added":
            self.update_example(model, -1)

        if event == "clear":
            self.ax.clear()
            self.ax.set_xticks([])
            self.ax.set_yticks([])
            self.contours = []
            self.c_labels = None
            self.plot_kernels()

        if event == "surface":
            self.remove_surface()
            self.plot_support_vectors(model.clf.support_vectors_)
            self.plot_decision_surface(model.surface, model.surface_type)

        self.canvas.draw()

    def remove_surface(self):
        """Remove old decision surface."""
        if len(self.contours) > 0:
            for contour in self.contours:
                if isinstance(contour, ContourSet):
```

```
                for lineset in contour.collections:
                    lineset.remove()
            else:
                contour.remove()
        self.contours = []

    def plot_support_vectors(self, support_vectors):
        """Plot the support vectors by placing circles over the
        corresponding data points and adds the circle collection
        to the contours list."""
        cs = self.ax.scatter(support_vectors[:, 0], support_vectors[:, 1],
                             s=80, edgecolors="k", facecolors="none")
        self.contours.append(cs)

    def plot_decision_surface(self, surface, type):
        X1, X2, Z = surface
        if type == 0:
            levels = [-1.0, 0.0, 1.0]
            linestyles = ['dashed', 'solid', 'dashed']
            colors = 'k'
            self.contours.append(self.ax.contour(X1, X2, Z, levels,
                                                 colors=colors,
                                                 linestyles=linestyles))
        elif type == 1:
            self.contours.append(self.ax.contourf(X1, X2, Z, 10,
                                             cmap=matplotlib.cm.bone,
                                             origin='lower',
                                             alpha=0.85))
            self.contours.append(self.ax.contour(X1, X2, Z, [0.0],
                                                 colors='k',
                                                 linestyles=['solid']))
        else:
            raise ValueError("surface type unknown")


class ControllBar(object):
    def __init__(self, root, controller):
        fm = Tk.Frame(root)
        kernel_group = Tk.Frame(fm)
        Tk.Radiobutton(kernel_group, text="Linear", variable=controller.kernel,
                       value=0, command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(kernel_group, text="RBF", variable=controller.kernel,
                       value=1, command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(kernel_group, text="Poly", variable=controller.kernel,
                       value=2, command=controller.refit).pack(anchor=Tk.W)
        kernel_group.pack(side=Tk.LEFT)

        valbox = Tk.Frame(fm)
        controller.complexity = Tk.StringVar()
        controller.complexity.set("1.0")
        c = Tk.Frame(valbox)
        Tk.Label(c, text="C:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(c, width=6, textvariable=controller.complexity).pack(
            side=Tk.LEFT)
        c.pack()

        controller.gamma = Tk.StringVar()
        controller.gamma.set("0.01")
        g = Tk.Frame(valbox)
        Tk.Label(g, text="gamma:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(g, width=6, textvariable=controller.gamma).pack(side=Tk.LEFT)
        g.pack()

        controller.degree = Tk.StringVar()
        controller.degree.set("3")
        d = Tk.Frame(valbox)
        Tk.Label(d, text="degree:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(d, width=6, textvariable=controller.degree).pack(side=Tk.LEFT)
        d.pack()

        controller.coef0 = Tk.StringVar()
        controller.coef0.set("0")
        r = Tk.Frame(valbox)
        Tk.Label(r, text="coef0:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(r, width=6, textvariable=controller.coef0).pack(side=Tk.LEFT)
        r.pack()
        valbox.pack(side=Tk.LEFT)

        cmap_group = Tk.Frame(fm)
        Tk.Radiobutton(cmap_group, text="Hyperplanes",
                       variable=controller.surface_type, value=0,
                       command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(cmap_group, text="Surface",
                       variable=controller.surface_type, value=1,
                       command=controller.refit).pack(anchor=Tk.W)

        cmap_group.pack(side=Tk.LEFT)

        train_button = Tk.Button(fm, text='Fit', width=5,
                                 command=controller.fit)
        train_button.pack()
        fm.pack(side=Tk.LEFT)
        Tk.Button(fm, text='Clear', width=5,
                  command=controller.clear_data).pack(side=Tk.LEFT)


def get_parser():
    from optparse import OptionParser
    op = OptionParser()
    op.add_option("--output",
              action="store", type="str", dest="output",
              help="Path where to dump data.")
    return op


def main(argv):
    op = get_parser()
    opts, args = op.parse_args(argv[1:])
    root = Tk.Tk()
    model = Model()
    controller = Controller(model)
    root.wm_title("Scikit-learn Libsvm GUI")
    view = View(root, controller)
    model.add_observer(view)
    Tk.mainloop()
    if opts.output:
        model.dump_svmlight_file(opts.output)

if __name__ == "__main__":
    main(sys.argv)
```

CHAPTER 9

People

This tutorial is brought to you by the scikit-learn[1] folks, in particular:

- Jake Vanderplas[2]
- Olivier Grisel
- Jaques Grobler
- Gael Varoquaux[3]

[1] http://scikit-learn.org/
[2] http://www.astro.washington.edu/vanderplas
[3] http://gael-varoquaux.info/blog/

CHAPTER 10

Citing the scikit-learn

A huge amount of work goes in the scikit-learn. Researchers that invest their time in developing and maintaining the package deserve recognition with citations. In addition, the Parietal team needs the citations to the paper in order to justify paying a software engineer on the project. To garanty the future of the toolkit, if you use it, please cite it.

See the scikit-learn documentation on how to cite[1].

[1] http://scikit-learn.org/stable/about.html#citing-scikit-learn