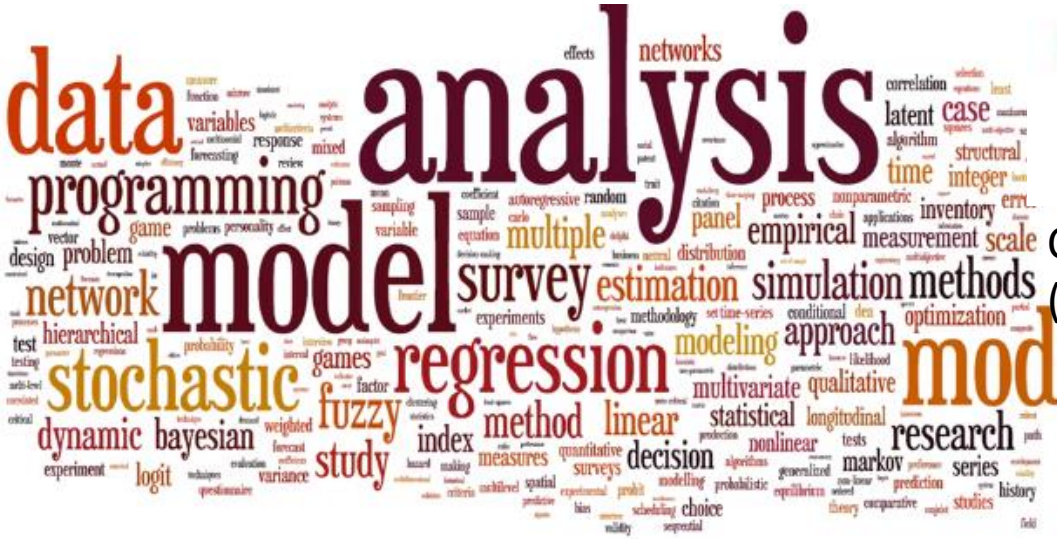
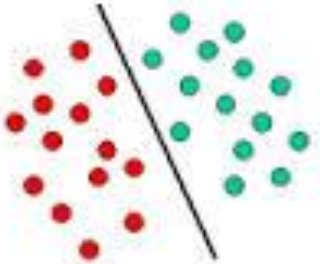
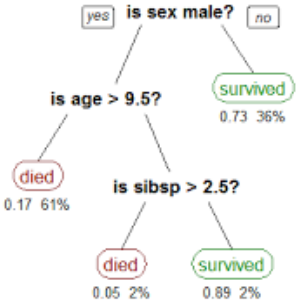
Regression and supervised classification with Python







Classification (LR, SVM…)

Trees

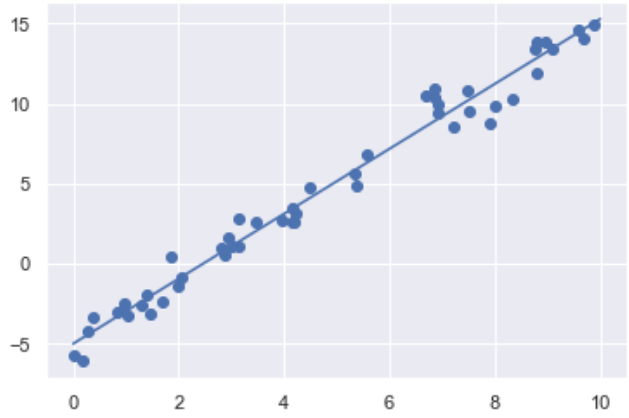
Luís Garmendia

Index

- Linear Regression
  - Multidimensional linear regression
  - Polynomical linear regression
  - Gaussian bases linear regression
- Naïve Bayes Classification
  - Gaussian Naïve Bayes
  - Multinomial Naïve Bayes
  - Classifying Text
- Support Vector Machines
  - Radial Basis Functions
  - SVM overlap
- Decision Trees and Random Forests
  - Decision trees over-fitting
  - Random Forest classification and regression
  - Classifying Digits
- Neuronal Networks
  - Multi-layer Perceptron (MLP)

Linear Regression

```
In [153]: from sklearn.linear_model import LinearRegression
          model = LinearRegression(fit_intercept=True)

          model.fit(x[:, np.newaxis], y)

          xfit = np.linspace(0, 10, 1000)
          yfit = model.predict(xfit[:, np.newaxis])

          plt.scatter(x, y)
          plt.plot(xfit, yfit);
```



Luís Garmendia

Linear regression

We will start with the most familiar linear regression, a straight-line fit to data. A straight-line fit is a model of the form

$$y = ax + b$$

where a is commonly known as the *slope*, and b is commonly known as the *intercept*.

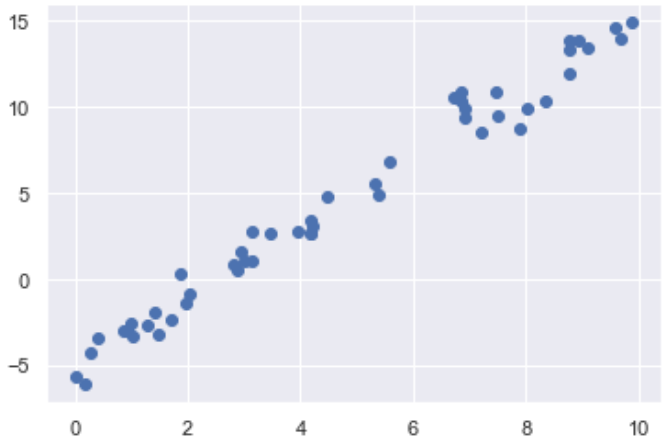
Linear regression

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

Consider the following data, which is scattered about a line with a slope of 2 and an intercept of -5:

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = 2 * x - 5 + rng.randn(50)
plt.scatter(x, y);
```

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = 2 * x - 5 + rng.randn(50)
plt.scatter(x, y);
```

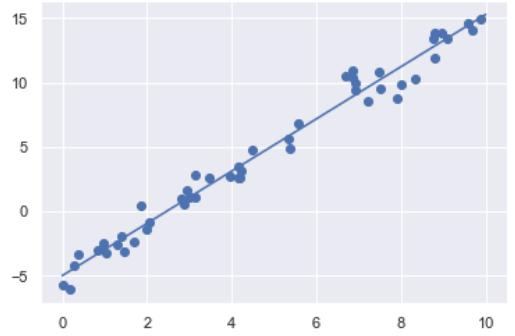


Linear regression

We can use Scikit-Learn's LinearRegression estimator to fit this data and construct the best-fit line:

```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)
model.fit(x[:, np.newaxis], y)
xfit = np.linspace(0, 10, 1000)
yfit = model.predict(xfit[:, np.newaxis])
plt.scatter(x, y)
plt.plot(xfit, yfit);
```

```
In [153]: from sklearn.linear_model import LinearRegression
          model = LinearRegression(fit_intercept=True)

          model.fit(x[:, np.newaxis], y)

          xfit = np.linspace(0, 10, 1000)
          yfit = model.predict(xfit[:, np.newaxis])

          plt.scatter(x, y)
          plt.plot(xfit, yfit);
```



Linear regression slope and intercept

The slope and intercept of the data are contained in the model's fit parameters, which in Scikit-Learn are always marked by a trailing underscore. Here the relevant parameters are coef_ and intercept_

```
print("Model slope: ", model.coef_[0])
print("Model intercept:", model.intercept_)
```

```
Model slope: 2.02720881036
Model intercept: -4.99857708555
```

Multidimensional linear regression

The LinearRegression estimator is much more capable than this, however—in addition to simple straight-line fits, it can also handle multidimensional linear models of the form

$$y=a_0+a_1x_1+a_2x_2+\cdots$$

```
rng = np.random.RandomState(1)
X = 10 * rng.rand(100, 3)
y = 0.5 + np.dot(X, [1.5, -2., 1.])
model.fit(X, y)
print(model.intercept_)
print(model.coef_)
```

```
0.5
[ 1.5 -2. 1. ]
```

Polynomial regression

polynomial regression:

$y=a_0+a_1x+a_2x^2+a_3x^3+ \ldots$

Notice that this is *still a linear model*—the linearity refers to the fact that the coefficients a_n never multiply or divide each other.

This polynomial projection is useful enough that it is built into Scikit-Learn, using the `PolynomialFeatures` transformer

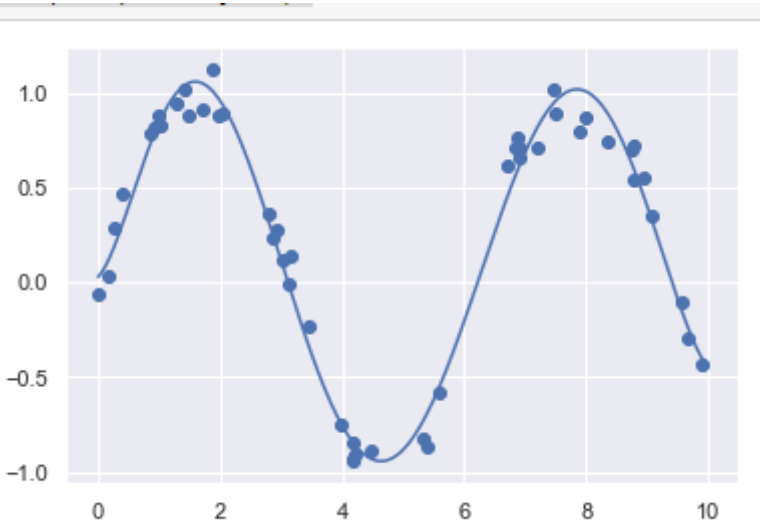
Polynomial regression Feature Engineering

```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:, None])
```

```
In [156]: from sklearn.preprocessing import PolynomialFeatures
          x = np.array([2, 3, 4])
          poly = PolynomialFeatures(3, include_bias=False)
          poly.fit_transform(x[:, None])

Out[156]: array([[ 2.,  4.,  8.],
                 [ 3.,  9., 27.],
                 [ 4., 16., 64.]])
```

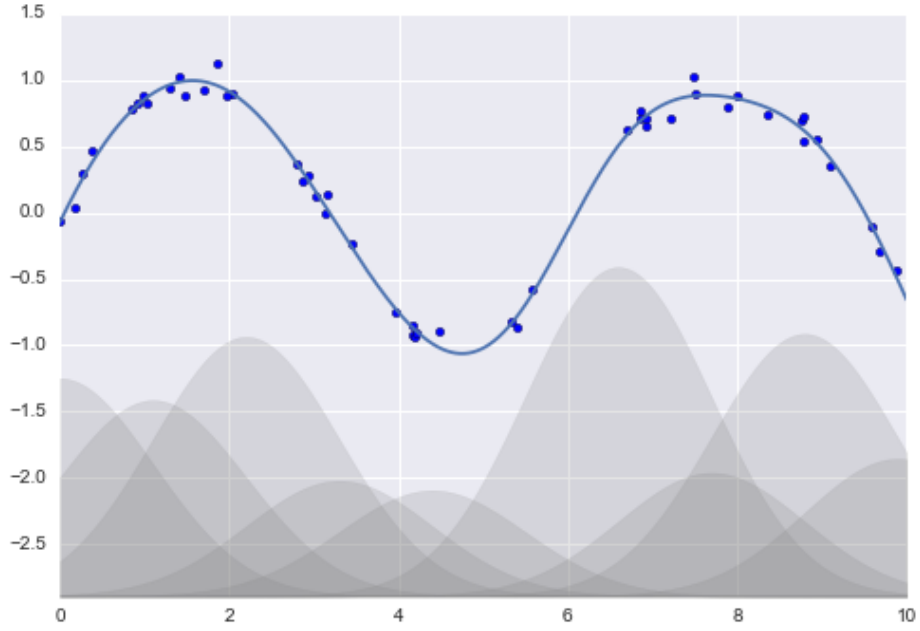
Polynomial regression Feature Engineering

```
from sklearn.pipeline import make_pipeline
poly_model = make_pipeline(PolynomialFeatures(7), LinearRegression())
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
poly_model.fit(x[:, np.newaxis], y)
yfit = poly_model.predict(xfit[:, np.newaxis])
plt.scatter(x, y)
plt.plot(xfit, yfit);
```



Gaussian basis functions

One useful pattern is to fit a model that is not a sum of polynomial bases, but a sum of Gaussian bases. The result might look something like the following figure:

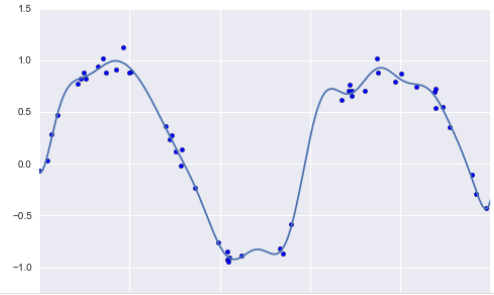


Gaussian basis functions

- These Gaussian basis functions are not built into Scikit-Learn, but we can write a custom transformer that will create

```
class GaussianFeatures(BaseEstimator, TransformerMixin):
  """Uniformly spaced Gaussian features for one-dimensional input"""
  def __init__(self, N, width_factor=2.0):
    self.N = N
    self.width_factor = width_factor
  @staticmethod
  def _gauss_basis(x, y, width, axis=None):
    arg = (x - y) / width
    return np.exp(-0.5 * np.sum(arg ** 2, axis))
  def fit(self, X, y=None):
    # create N centers spread along the data range
    self.centers_ = np.linspace(X.min(), X.max(), self.N)
    self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
    return self
  def transform(self, X):
    return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                             self.width_, axis=1)
gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])
plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```

```
In [9]: from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                 self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```

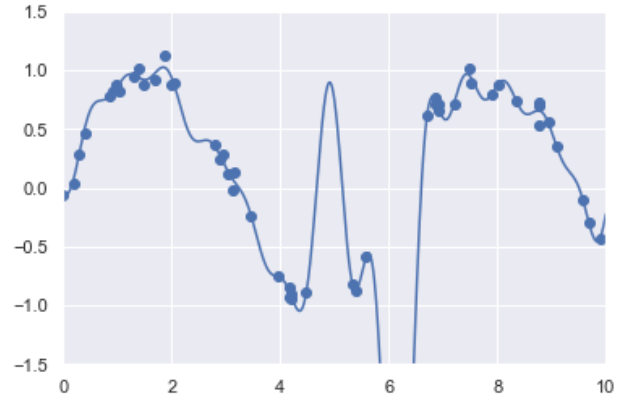


Gaussian basis functions

If we choose too many Gaussian basis functions, we end up with results that don't look so good:

```
model = make_pipeline(GaussianFeatures(30), LinearRegression())
model.fit(x[:, np.newaxis], y)
plt.scatter(x, y)
plt.plot(xfit, model.predict(xfit[:, np.newaxis]))
plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);
```

```
In [161]: model = make_pipeline(GaussianFeatures(30),
                                LinearRegression())
          model.fit(x[:, np.newaxis], y)

          plt.scatter(x, y)
          plt.plot(xfit, model.predict(xfit[:, np.newaxis]))

          plt.xlim(0, 10)
          plt.ylim(-1.5, 1.5);
```

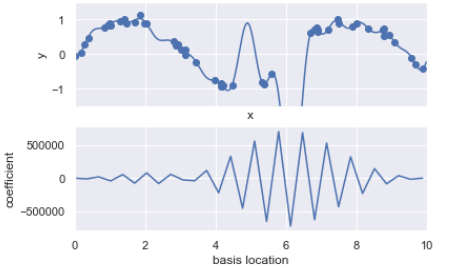


Gaussian basis functions coefficient of the Gaussian Bases

We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations:

```
def basis_plot(model, title=None):
    fig, ax = plt.subplots(2, sharex=True)
    model.fit(x[:, np.newaxis], y)
    ax[0].scatter(x, y)
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))
    if title:
        ax[0].set_title(title)
    ax[1].plot(model.steps[0][1].centers_, model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location', ylabel='coefficient', xlim=(0, 10))
model = make_pipeline(GaussianFeatures(30), LinearRegression())
basis_plot(model)
```

```
In [162]: def basis_plot(model, title=None):
              fig, ax = plt.subplots(2, sharex=True)
              model.fit(x[:, np.newaxis], y)
              ax[0].scatter(x, y)
              ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
              ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

              if title:
                  ax[0].set_title(title)

              ax[1].plot(model.steps[0][1].centers_,
                         model.steps[1][1].coef_)
              ax[1].set(xlabel='basis location',
                        ylabel='coefficient',
                        xlim=(0, 10))

          model = make_pipeline(GaussianFeatures(30), LinearRegression())
          basis_plot(model)
```



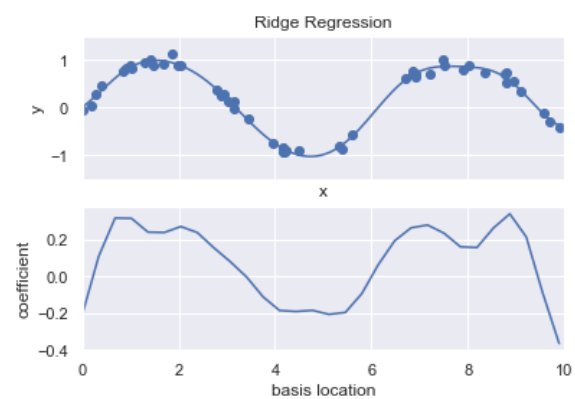
Ridge regression

Perhaps the most common form of regularization is known as *ridge regression*, sometimes also called *Tikhonov regularization*.This proceeds by penalizing the sum of squares (2-norms) of the model coefficients; in this case, the penalty on the model fit would be

$$P = \alpha \sum_{n=1}^{N} \theta_n^2$$

```
from sklearn.linear_model import Ridge
model = make_pipeline(GaussianFeatures(30),
            Ridge(alpha=0.1))
basis_plot(model, title='Ridge Regression')
```

```
In [163]: from sklearn.linear_model import Ridge
          model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
          basis_plot(model, title='Ridge Regression')
```

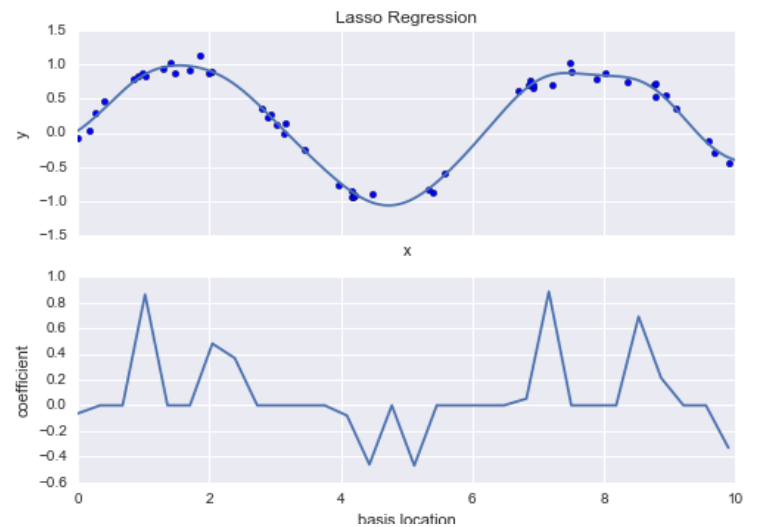


Lasso regression

Another very common type of regularization is known as lasso, and involves penalizing the sum of absolute values (1-norms) of regression coefficients

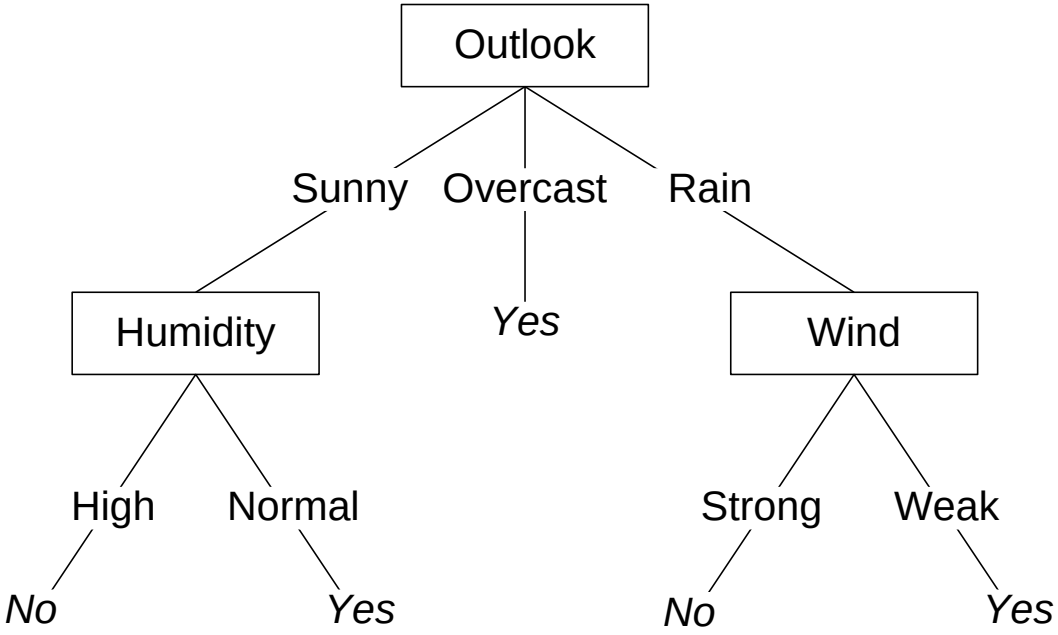
$$P = \alpha \sum_{n=1}^{N} |\theta_n|$$

```
from sklearn.linear_model import Lasso
model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
basis_plot(model, title='Lasso Regression')
```




```
from sklearn.linear_model import Lasso
model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
basis_plot(model, title='Lasso Regression')
```

Naive Bayes Classification



Day	*Outlook*	*Temperature*	*Humidity*	*Wind*	*PlayTennis*
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Luís Garmendia

Naïve Bayes

Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being very useful as a quick-and-dirty baseline for a classification problem.

Bayesian Classification

Naive Bayes classifiers are built on Bayesian classification methods. These rely on Bayes's theorem, which is an equation describing the relationship of conditional probabilities of statistical quantities.

$$P(L \mid features) = P(features \mid L)\, P(L) \,/\, P(features)$$

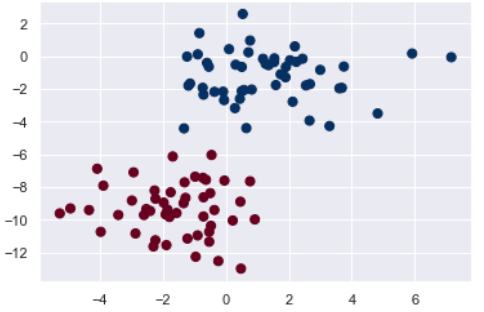
Such a model is called a ***generative*** *model* because it specifies the hypothetical random process that generates the data. Specifying this generative model for each label is the main piece of the training of such a Bayesian classifier.

This is where the "naive" in "naive Bayes" comes in: if we make very naive assumptions about the generative model for each label, we can find a rough approximation of the generative model for each class, and then proceed with the Bayesian classification.

Gaussian Naïve Bayes

Perhaps the easiest naive Bayes classifier to understand is Gaussian naive Bayes. In this classifier, the assumption is that *data from each label is drawn from a simple Gaussian distribution*.

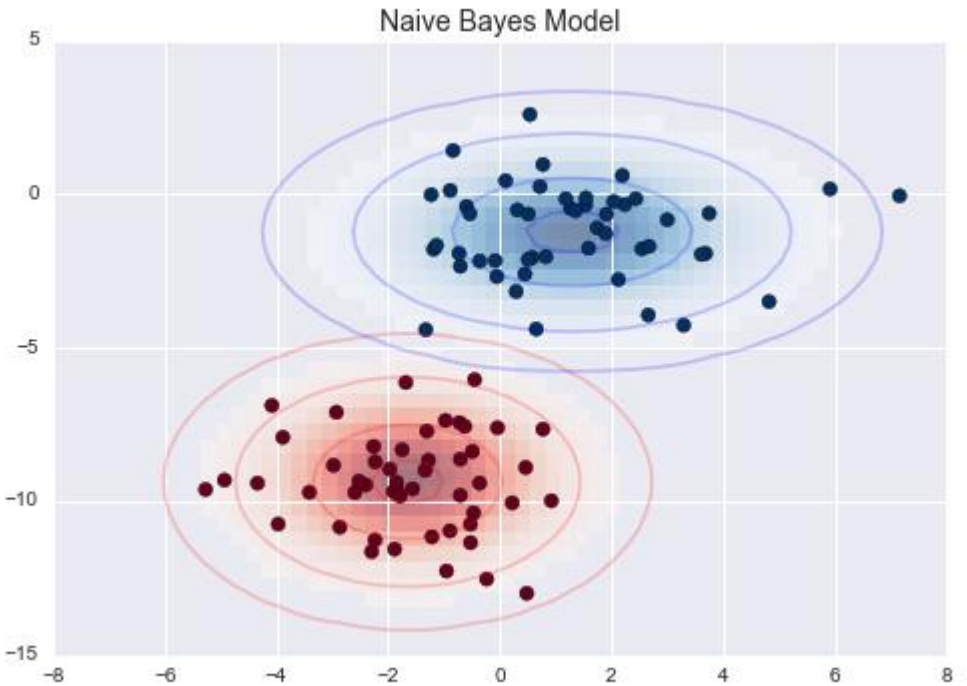
```
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
```




```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
```

Gaussian Naïve Bayes

One extremely fast way to create a simple model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions. This model can be fit by simply finding the mean and standard deviation of the points within each label



Gaussian Naïve Bayes

This procedure is implemented in Scikit-Learn's sklearn.naive_bayes.GaussianNB estimator.

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X, y);
```

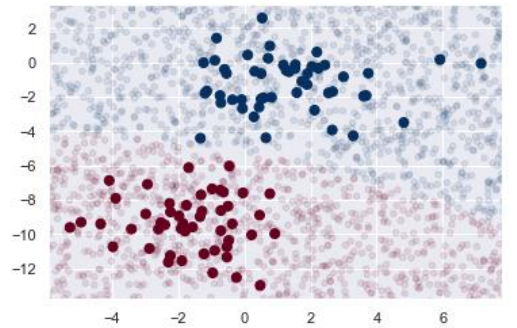
Gaussian Naïve Bayes

Now let's generate some new data and predict the label:

```
rng = np.random.RandomState(0)
Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
ynew = model.predict(Xnew)
```

```
In [141]: ynew
Out[141]: array([1, 1, 1, ..., 0, 1, 1])
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
lim = plt.axis()
plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu', alpha=0.1)
plt.axis(lim);
```



Now we can plot this new data to get an idea of where the decision boundary is:

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
lim = plt.axis()
plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu', alpha=0.1)
plt.axis(lim);
```

Gaussian Naïve Bayes predict_proba

It naturally allows for probabilistic classification, which we can compute using the predict_proba method:

```
In [135]: yprob = model.predict_proba(Xnew)
          yprob[-8:].round(2)

Out[135]: array([[0.89, 0.11],
                 [1.  , 0.  ],
                 [1.  , 0.  ],
                 [1.  , 0.  ],
                 [1.  , 0.  ],
                 [1.  , 0.  ],
                 [0.  , 1.  ],
                 [0.15, 0.85]])
```

The columns give the posterior probabilities of 0 and 1.

Suppose you are working on cancer diagnosis problem and you want to be very sure with your results. So in that case you can use predict_proba which will give you class probability values and you can set some threshold like if predict_proba_value > .98 return class 1 else 0. So basically with the help of predit_proba we can set threshold as per our needs.

Multinomial Naïve Bayes

Another useful example is multinomial naive Bayes, where the features are assumed to be generated from a simple multinomial distribution. The multinomial distribution describes the probability of observing counts among a number of categories, and thus multinomial naive Bayes is most appropriate for features that represent counts or count rates.

Classifying Text

```
from sklearn.datasets import fetch_20newsgroups
data = fetch_20newsgroups()
data.target_names
```

```
['alt.atheism',
'comp.graphics',
'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware',
'comp.sys.mac.hardware',
'comp.windows.x', …,
'talk.politics.misc', 'talk.religion.misc']
```

Classifying Text

For simplicity here, we will select just a few of these categories, and download the training and testing set:

```
categories = ['talk.religion.misc', 'soc.religion.christian', 'sci.space', 'comp.graphics']
train = fetch_20newsgroups(subset='train', categories=categories)
test = fetch_20newsgroups(subset='test', categories=categories)
```

Here is a representative entry from the data:

```
print(train.data[5])
```

```
In [144]: print(train.data[5])

From: dmcgee@uluhe.soest.hawaii.edu (Don McGee)
Subject: Federal Hearing
Originator: dmcgee@uluhe
Organization: School of Ocean and Earth Science and Technology
Distribution: usa
Lines: 10


Fact or rumor....?  Madalyn Murray O'Hare an atheist who eliminated the
use of the bible reading and prayer in public schools 15 years ago is now
going to appear before the FCC with a petition to stop the reading of the
Gospel on the airways of America.  And she is also campaigning to remove
Christmas programs, songs, etc from the public schools.  If it is true
then mail to Federal Communications Commission 1919 H Street Washington DC
20054 expressing your opposition to her request.  Reference Petition number
```

Classifying Text

In order to use this data for machine learning, we need to be able to convert the content of each string into a vector of numbers. For this we will use the TF-IDF vectorizer and create a pipeline that attaches it to a multinomial naïve Bayes classifier:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

Classifying Text

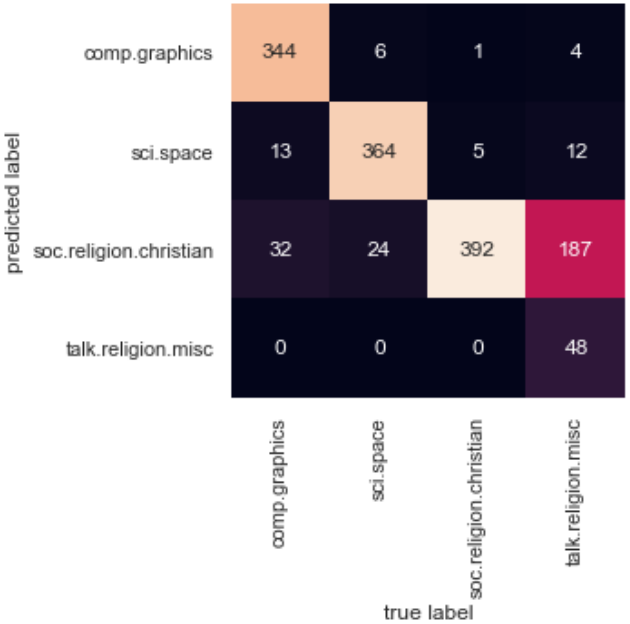
With this pipeline, we can apply the model to the training data, and predict labels for the test data:

```
model.fit(train.data, train.target)
labels = model.predict(test.data)
```

Classifying Text

Now that we have predicted the labels for the test data, we can evaluate them to learn about the performance of the estimator. For example, here is the confusion matrix between the true and predicted labels for the test data:

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(test.target, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
    xticklabels=train.target_names, yticklabels=train.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```



Classifying Text

Evidently, even this very simple classifier can successfully separate space talk from computer talk, but it gets confused between talk about religion and talk about Christianity. This is perhaps an expected area of confusion!

The very cool thing here is that we now have the tools to determine the category for *any* string, using the predict() method of this pipeline. Here's a quick utility function that will return the prediction for a single string:

```
def predict_category(s, train=train, model=model):
    pred = model.predict([s])
    return train.target_names[pred[0]]
predict_category('sending a payload to the ISS')
predict_category('discussing islam vs atheism')
predict_category('determining the screen resolution')
```

Classifying Text

```
In [148]: def predict_category(s, train=train, model=model):
              pred = model.predict([s])
              return train.target_names[pred[0]]

In [149]: predict_category('sending a payload to the ISS')

Out[149]: 'sci.space'

In [150]: predict_category('discussing islam vs atheism')

Out[150]: 'soc.religion.christian'

In [151]: predict_category('determining the screen resolution')

Out[151]: 'comp.graphics'
```

When to use Naïve Bayes

Evidently, even this very simple classifier Because naive Bayesian classifiers make such stringent assumptions about data, they will generally not perform as well as a more complicated model. That said, they have several advantages:

- They are extremely fast for both training and prediction
- They provide straightforward probabilistic prediction
- They are often very easily interpretable
- They have very few (if any) tunable parameters

When to use Naïve Bayes

These advantages mean a naive Bayesian classifier is often a good choice as an initial baseline classification. If it performs suitably, then congratulations: you have a very fast, very interpretable classifier for your problem. If it does not perform well, then you can begin exploring more sophisticated models, with some baseline knowledge of how well they should perform.

Naive Bayes classifiers tend to perform especially well in one of the following situations:

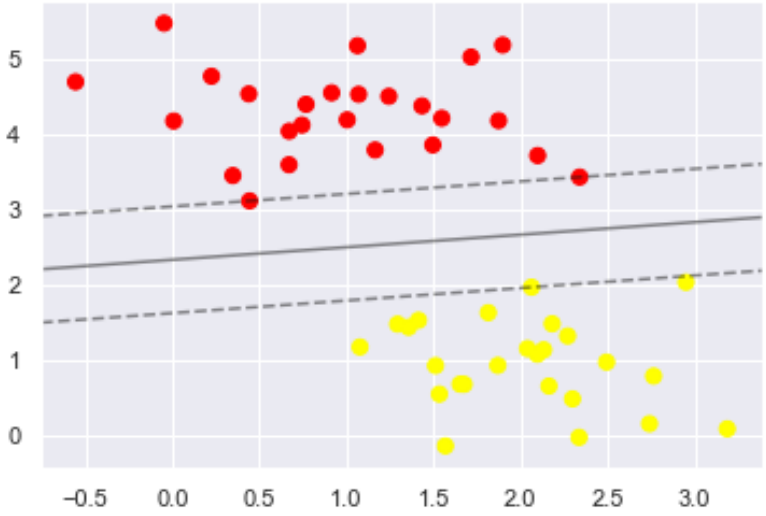
- When the naive assumptions actually match the data (very rare in practice)
- For very well-separated categories, when model complexity is less important
- For very high-dimensional data, when model complexity is less important

Classifiers like naive Bayes tend to work as well or better than more complicated classifiers as the dimensionality grows: once you have enough data, even a simple model can be very powerful.

Support Vector Machines

Classification and regression

```
In [171]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
          plot_svc_decision_function(model);
```



Luís Garmendia

Support vector machines (SVMs)

Support vector machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# use seaborn plotting defaults
import seaborn as sns; sns.set()
```

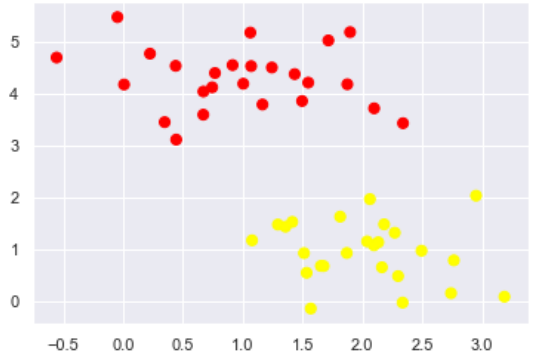
Support vector machines (SVMs)

We will consider instead ***discriminative classification***: rather than modeling each class (generative classification).

We find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

As an example of this, consider the simple case of a classification task, in which the two classes of points are well separated:

```
from sklearn.datasets.samples_generator import make_blobs
X, y = make_blobs(n_samples=50, centers=2,
        random_state=0, cluster_std=0.60)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```



Support vector machines (SVMs)

A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and thereby create a model for classification.

For the time being, we will use a linear kernel and set the `C` parameter to a very large number

```
from sklearn.svm import SVC # "Support vector classifier"
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)
```

Support vector machines (SVMs)

To better visualize what's happening here, let's create a quick convenience function that will plot SVM decision boundaries for us:

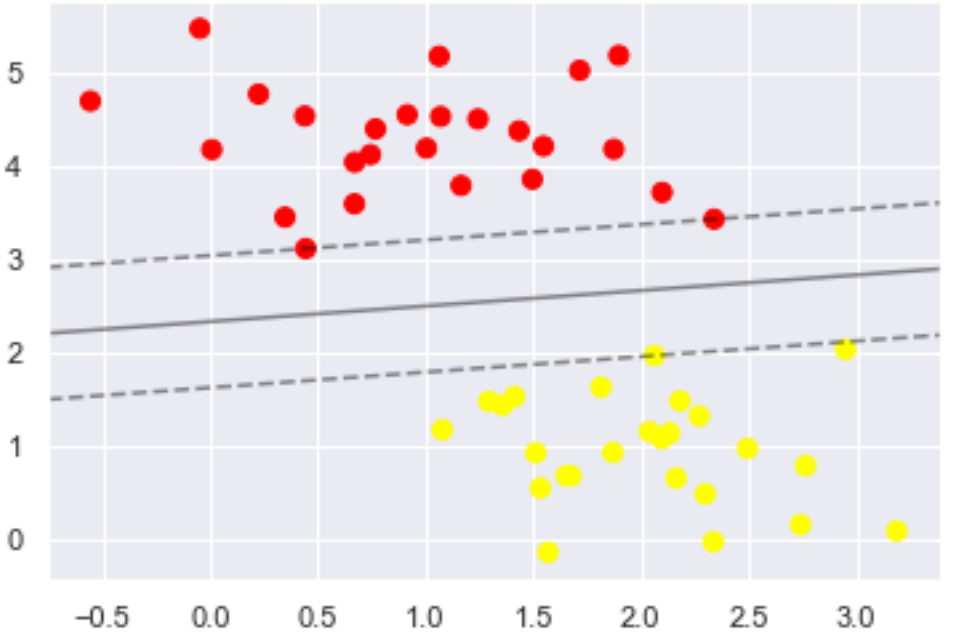
```
def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()
    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)
    # plot decision boundary and margins
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])
    # plot support vectors
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=300, linewidth=1, facecolors='none');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
```

Support vector machines (SVMs)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

plot_svc_decision_function(model);

```
In [171]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
          plot_svc_decision_function(model);
```



Support vector machines support_vectors_

The dividing line maximizes the margin between the two sets of points.

Notice that a few of the training points just touch the margin: they are indicated by the black circles in this figure.

These points are the pivotal elements of this fit, and are known as the support vectors, and give the algorithm its name.

In Scikit-Learn, the identity of these points are stored in the support_vectors_ attribute of the classifier:

```
model.support_vectors_
```

```
array([[ 0.44359863, 3.11530945],
[ 2.33812285, 3.43116792],
[ 2.06156753, 1.96918596]])
```

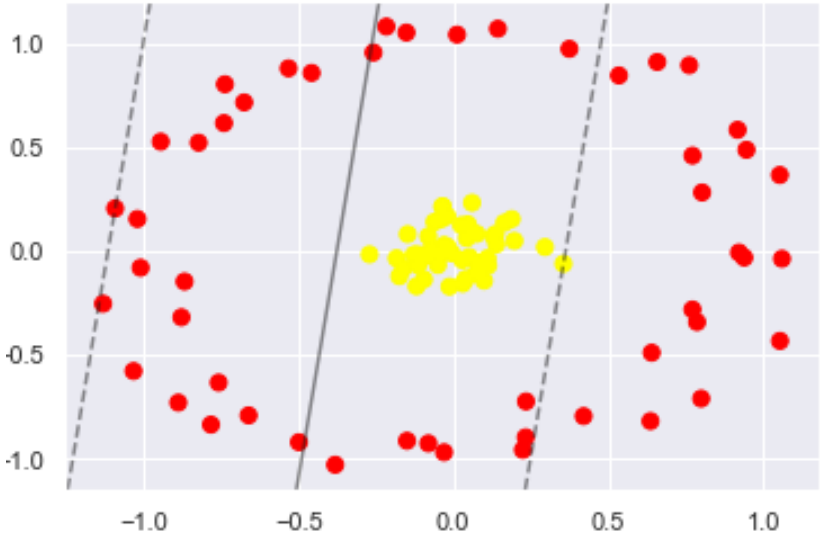
Support vector machines (SVMs)

A key to this classifier's success is that for the fit, only the position of the support vectors matter; any points further from the margin which are on the correct side do not modify the fit! Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

Kernel SVM

To motivate the need for kernels, let's look at some data that is not linearly separable:.

```
from sklearn.datasets  import make_circles
X, y = make_circles(100, factor=.1, noise=.1)
clf = SVC(kernel='linear').fit(X, y)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf, plot_support=False);
```



Kernel SVM
RBF

One simple projection we could use would be to compute a *radial basis function* centered on the middle clump.

```
r = np.exp(-(X ** 2).sum(1))
```

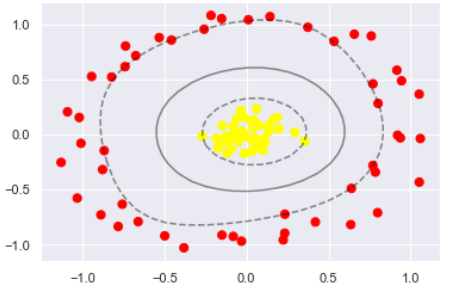
In Scikit-Learn, we can apply kernelized SVM simply by changing our linear kernel to an RBF (**radial basis function**) kernel, using the `kernel` model hyperparameter:

```
clf =SVC(kernel='rbf', C=1E6)
clf.fit(X, y)
```

Kernel SVM
RBF

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf)
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=300, lw=1,
facecolors='none');
```

```
In [179]: clf = SVC(kernel='rbf', C=1E6)
          clf.fit(X, y)

Out[179]: SVC(C=1000000.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
              decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
              max_iter=-1, probability=False, random_state=None, shrinking=True,
              tol=0.001, verbose=False)

In [180]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
          plot_svc_decision_function(clf)
          plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
                      s=300, lw=1, facecolors='none');
```

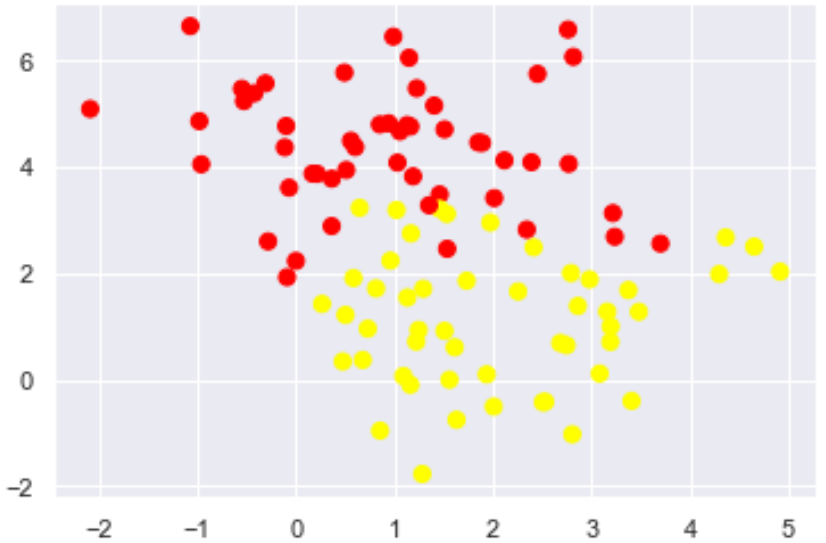


SVM overlap

X, y = make_blobs(n_samples=100, centers=2, random_state=0, cluster_std=1.2)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');

```
In [181]: X, y = make_blobs(n_samples=100, centers=2,
                            random_state=0, cluster_std=1.2)
          plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

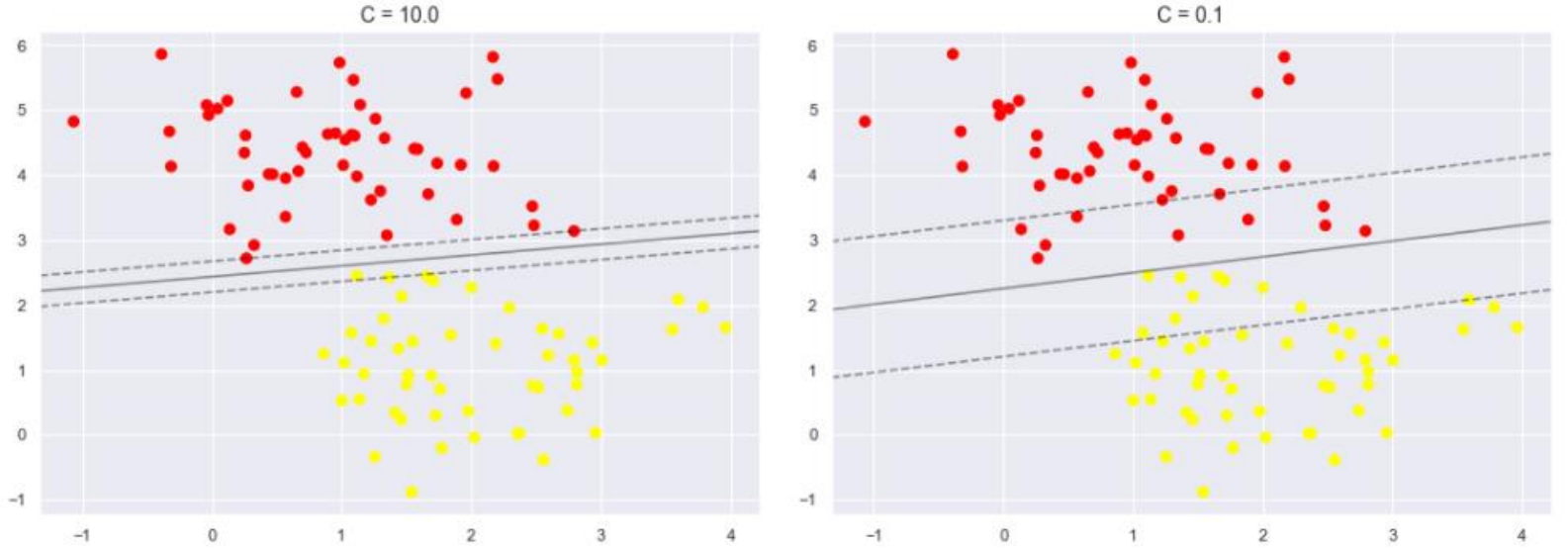


SVM overlap
C parameter

The hardness of the margin is controlled by a tuning parameter, most often known as C.

For very large C, the margin is hard, and points cannot lie in it.

For smaller C, the margin is softer, and can grow to encompass some points.



SVM overlap
C parameter

The plot shown below gives a visual picture of how a changing C parameter affects the final fit, via the softening of the margin

```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=0.8)
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for axi, C in zip(ax, [10.0, 0.1]):
    model = SVC(kernel='linear', C=C).fit(X, y)
    axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    plot_svc_decision_function(model, axi)
    axi.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=300, lw=1, facecolors='none');
    axi.set_title('C = {0:.1f}'.format(C), size=14)
```

Support Verctor Machine Summary

These methods are a powerful classification method for a number of reasons:

- Their dependence on relatively few support vectors means that they are very compact models, and take up very little memory.
- Once the model is trained, the prediction phase is very fast.
- Because they are affected only by points near the margin, they work well with high-dimensional data—even data with more dimensions than samples, which is a challenging regime for other algorithms.
- Their integration with kernel methods makes them very versatile, able to adapt to many types of data.
- Parameter C must be carefully chosen via cross-validation, which can be expensive as datasets grow in size.

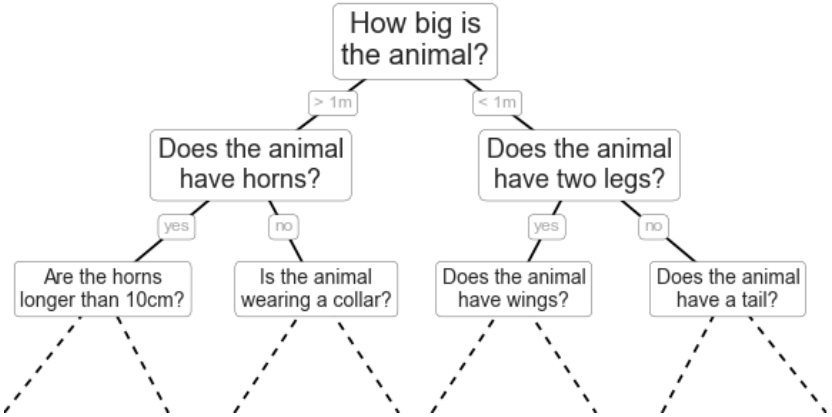
Support Vector Machine Summary

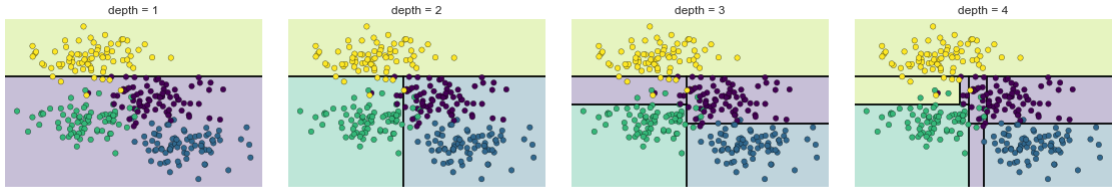
However, SVMs have several disadvantages as well:

- For large numbers of training samples, this computational cost can be prohibitive.
- The results are strongly dependent on a suitable choice for the softening parameter C. This must be carefully chosen via cross-validation, which can be expensive as datasets grow in size.
- The results do not have a direct probabilistic interpretation. This can be estimated via an internal cross-validation (see the probability parameter of SVC), but this extra estimation is costly.

With those traits in mind, turn to SVMs once other simpler, faster, and less tuning-intensive methods have been shown to be insufficient for my needs. Nevertheless, if you have the CPU cycles to commit to training and cross-validating an SVM on your data, the method can lead to excellent results.

Decision Trees
Random Forest





Luís Garmendia

Random Forest

Random forests are an example of an ***ensemble*** method, meaning that it relies on aggregating the results of an ensemble of simpler estimators.

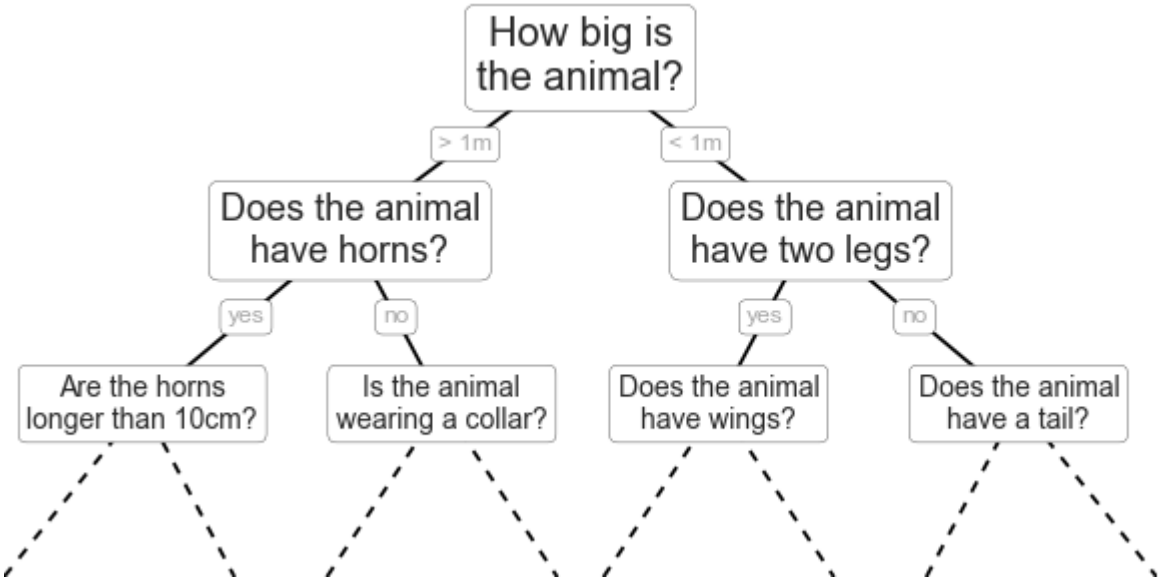
The somewhat surprising result with such ensemble methods is that the sum can be greater than the parts: that is, a majority vote among a number of estimators can end up being better than any of the individual estimators doing the voting!

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

Decision Tree

Decision trees are extremely intuitive ways to classify or label objects: you simply ask a series of questions designed to zero-in on the classification.

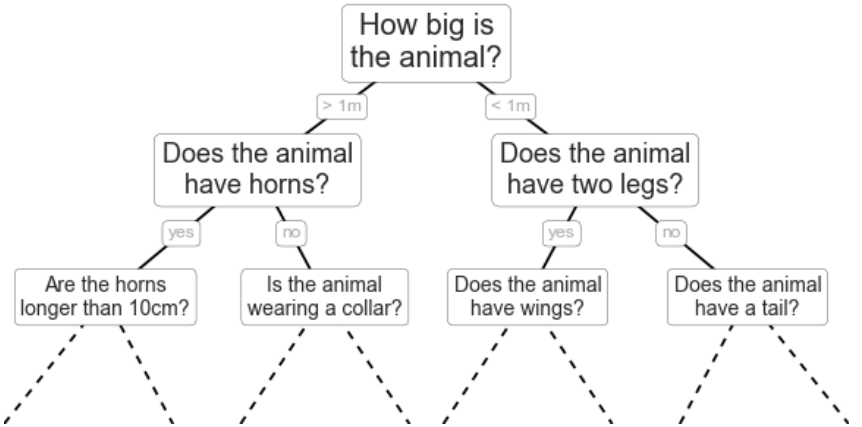
For example, if you wanted to build a decision tree to classify an animal you come across while on a hike, you might construct the one shown here:



Decision Tree

The binary splitting makes this extremely efficient: in a well-constructed tree, each question will cut the number of options by approximately half, very quickly narrowing the options even among a large number of classes.

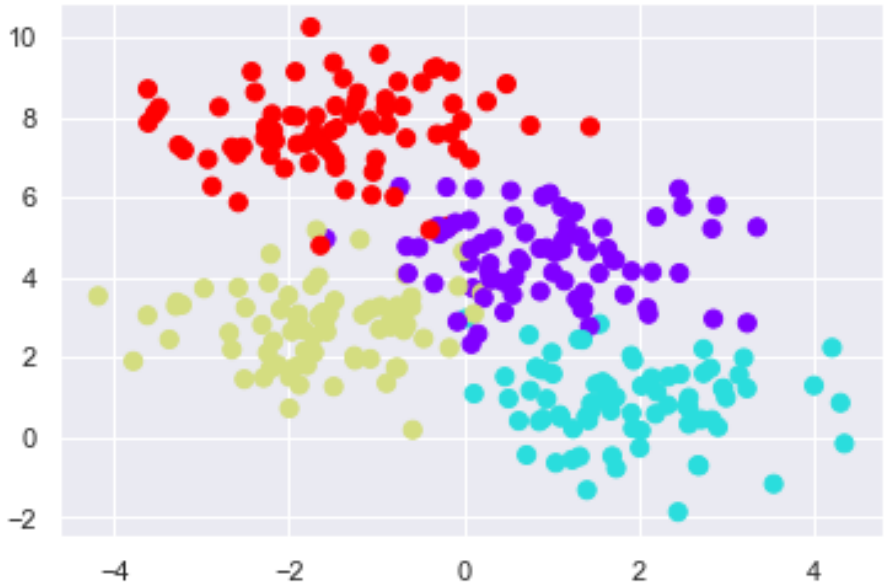
The trick, of course, comes in deciding which questions to ask at each step. In machine learning implementations of decision trees, the questions generally take the form of axis-aligned splits in the data: that is, each node in the tree splits the data into two groups using a cutoff value within one of the features.



Decision Tree

Consider the following two-dimensional data, which has one of four class labels:

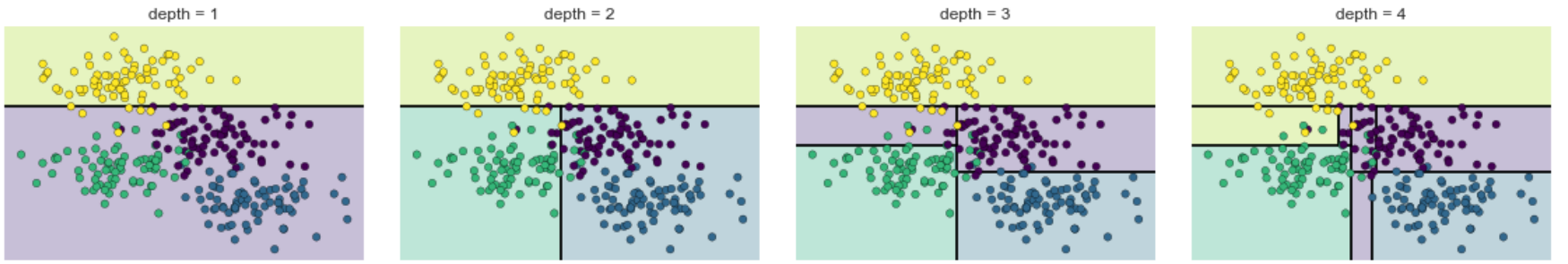
```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=300, centers=4, random_state=0, cluster_std=1.0)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```



Decision Tree

A simple decision tree built on this data will iteratively split the data along one or the other axis according to some quantitative criterion, and at each level assign the label of the new region according to a majority vote of points within it.

This figure presents a visualization of the first four levels of a decision tree classifier for this data:



Notice that after the first split, every point in the upper branch remains unchanged, so there is no need to further subdivide this branch.

Decision Tree

This process of fitting a decision tree to our data can be done in Scikit-Learn with the DecisionTreeClassifier estimator:

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier().fit(X, y)
```

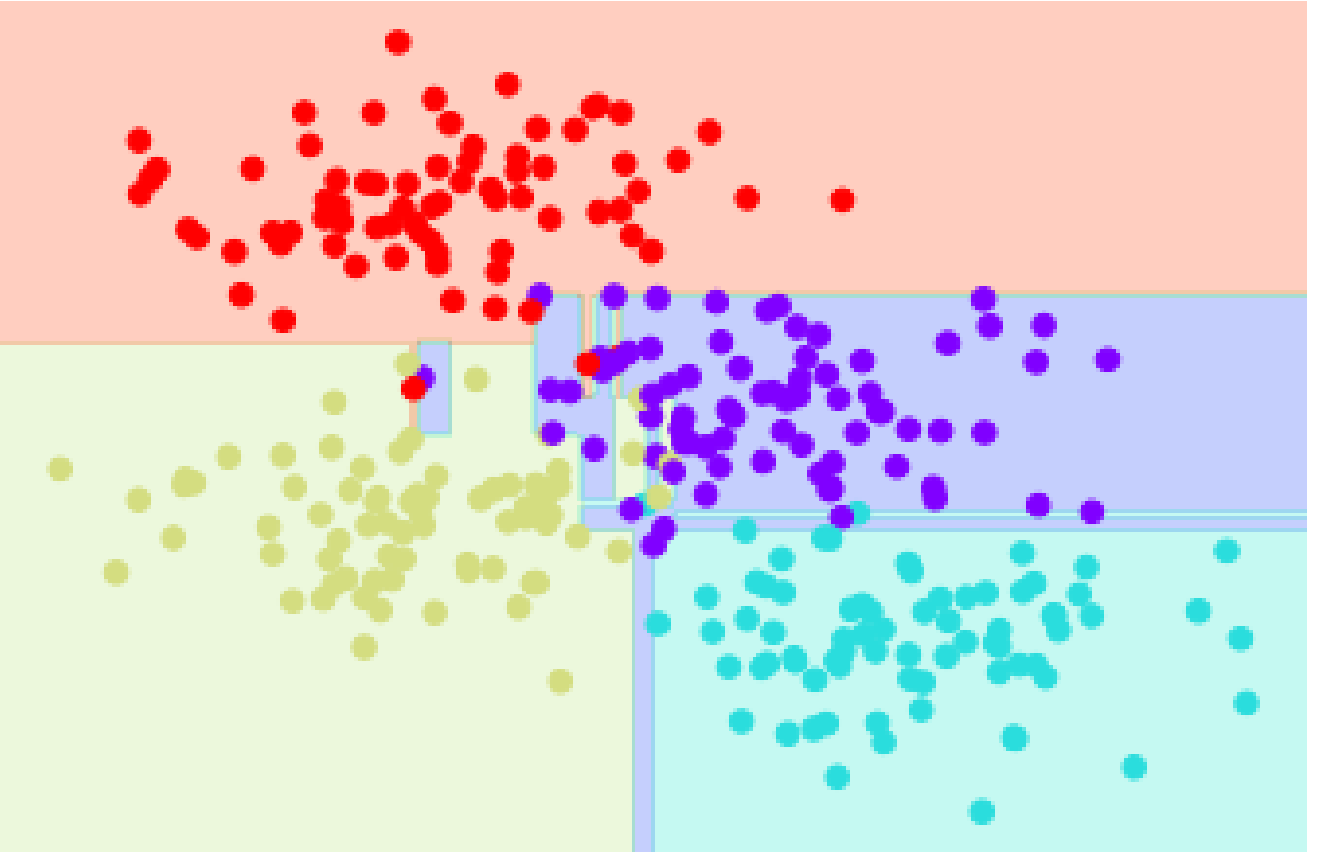
Decision Tree

This process of fitting a decision tree to our data can be done in Scikit-Learn with the DecisionTreeClassifier estimator:

```
def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
  ax = ax or plt.gca()
  ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap, clim=(y.min(), y.max()), zorder=3)
  ax.axis('tight')
  ax.axis('off')
  xlim = ax.get_xlim()
  ylim = ax.get_ylim()
  # fit the estimator
  model.fit(X, y)
  xx, yy = np.meshgrid(np.linspace(*xlim, num=200), np.linspace(*ylim, num=200))
  Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
  # Create a color plot with the results
  n_classes = len(np.unique(y))
  contours = ax.contourf(xx, yy, Z, alpha=0.3, levels=np.arange(n_classes + 1) - 0.5, cmap=cmap, zorder=1)
  ax.set(xlim=xlim, ylim=ylim)
```

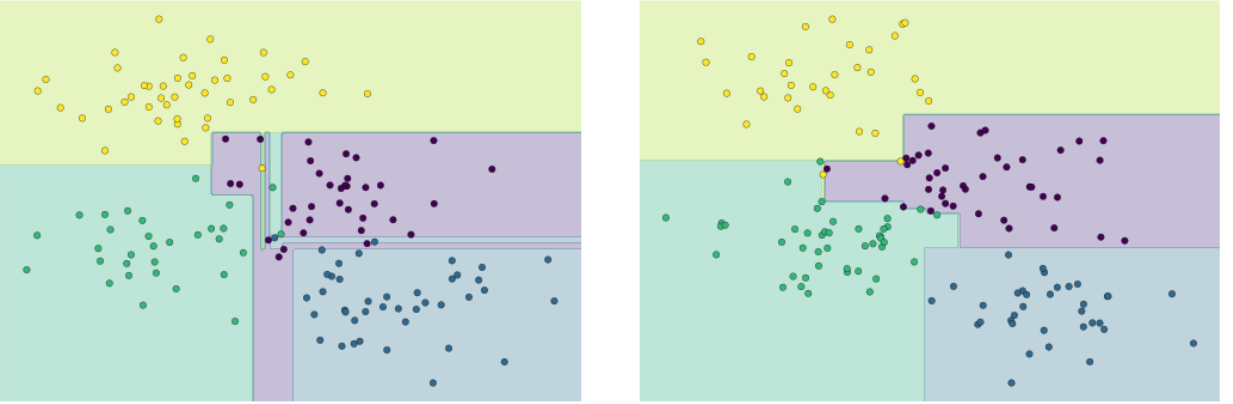
Decision Tree

visualize_classifier(DecisionTreeClassifier(), X, y)



Decision Tree and over-fitting

Look at models trained on different subsets of the data—for example, in this figure we train two different trees, each on half of the original data:



The two trees produce consistent results (e.g., in the four corners), while in other places, the two trees give very different classifications (e.g., in the regions between any two clusters). The key observation is that the inconsistencies tend to happen where the classification is less certain, and thus by using information from both of these trees, we might come up with a better result!

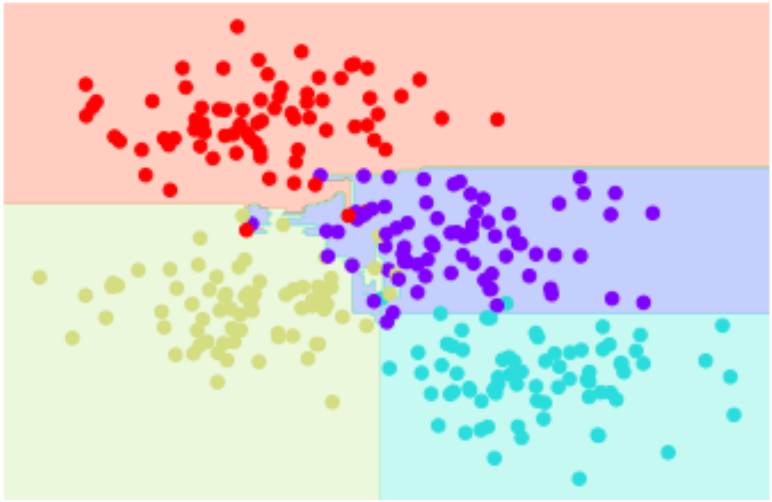
Random Forest

This notion—that multiple overfitting estimators can be combined to reduce the effect of this overfitting—is what underlies an ensemble method called ***bagging***. Bagging makes use of an ensemble (a grab bag, perhaps) of parallel estimators, each of which over-fits the data, and **averages the results** to find a better classification. An ensemble of randomized decision trees is known as a *random forest*.

This type of bagging classification can be done manually using Scikit-Learn's **BaggingClassifier** meta-estimator.

Random Forest

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
tree = DecisionTreeClassifier()
bag = BaggingClassifier(tree, n_estimators=100,
        max_samples=0.8, random_state=1)
bag.fit(X, y)
visualize_classifier(bag, X, y)
```



we have randomized the data by fitting each estimator with a random subset of 80% of the training points.

Random Forest

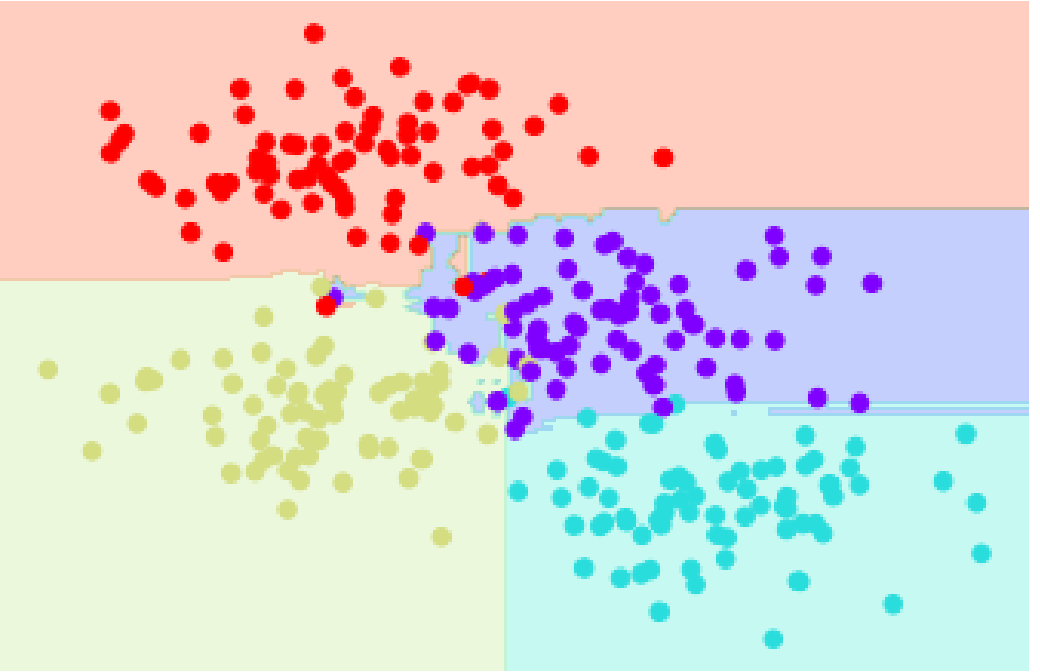
In practice, decision trees are more effectively randomized by injecting some stochasticity in how the splits are chosen: this way all the data contributes to the fit each time, but the results of the fit still have the desired randomness.

For example, when determining which feature to split on, the randomized tree might select from among the top several features.

In Scikit-Learn, such an optimized ensemble of randomized decision trees is implemented in the **RandomForestClassifier** estimator, which takes care of all the randomization automatically. All you need to do is select a number of estimators, and it will very quickly.

Random Forest

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100, random_state=0)
visualize_classifier(model, X, y);
```



Random Forest Regression

In the previous section we considered random forests within the context of classification.

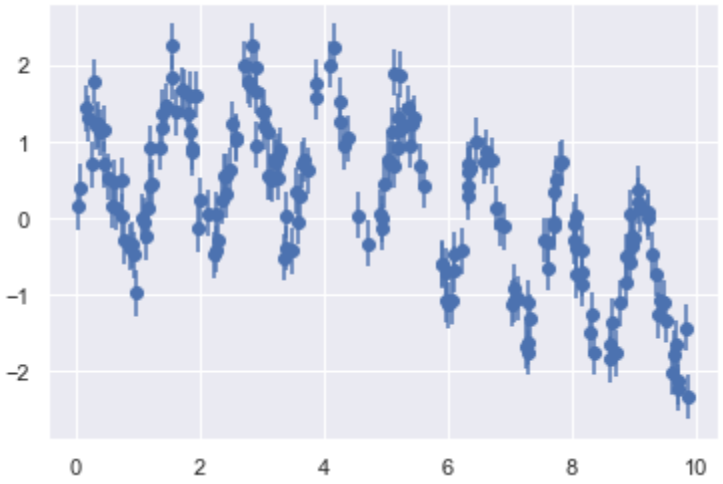
Random forests can also be made to work in the case of **regression** (that is**, continuous rather than categorical variables**).

The estimator to use for this is the `RandomForestRegressor`, and the syntax is very similar to what we saw earlier.

Random Forest Regression

Prepare some data, drawn from the combination of a fast and slow oscillation:

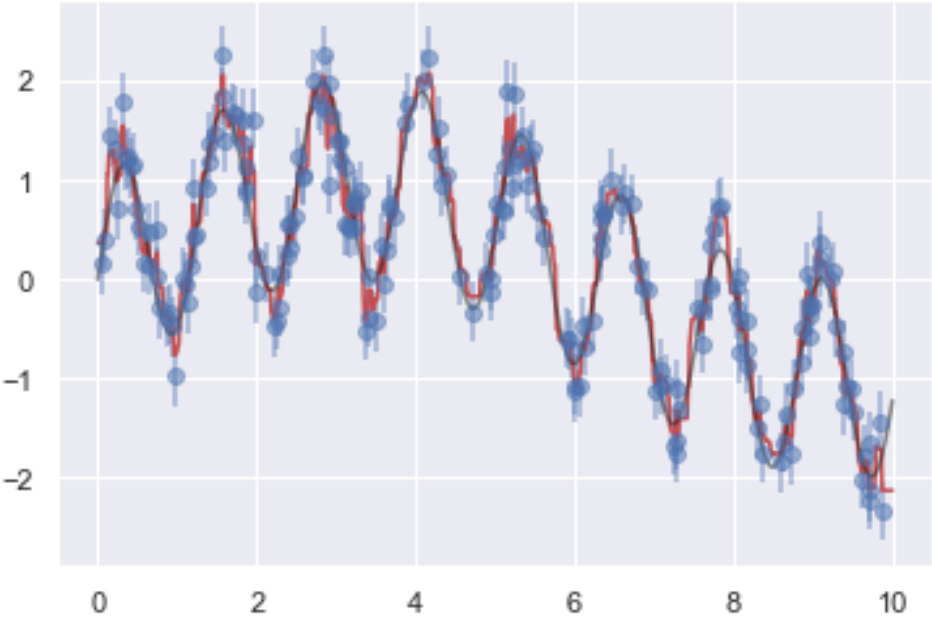
```
rng = np.random.RandomState(42)
x = 10 * rng.rand(200)
def model(x, sigma=0.3):
    fast_oscillation = np.sin(5 * x)
    slow_oscillation = np.sin(0.5 * x)
    noise = sigma * rng.randn(len(x))
    return slow_oscillation + fast_oscillation + noise
y = model(x)
plt.errorbar(x, y, 0.3, fmt='o');
```



Random Forest Regression

Using the random forest regressor, we can find the best fit curve

```
from sklearn.ensemble import RandomForestRegressor
forest = RandomForestRegressor(200)
forest.fit(x[:, None], y)
xfit = np.linspace(0, 10, 1000)
yfit = forest.predict(xfit[:, None])
ytrue = model(xfit, sigma=0)
plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)
plt.plot(xfit, yfit, '-r');
plt.plot(xfit, ytrue, '-k', alpha=0.5);
```



As you can see, the non-parametric random forest model is flexible enough to fit the multi-period data, without us needing to specifying a multi-period model!

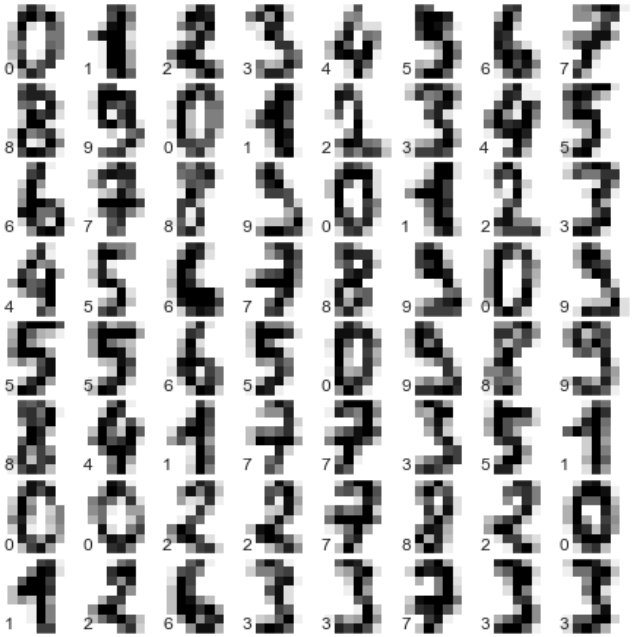
Classifying digits

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.keys()

 from sklearn.datasets import load_digits
digits = load_digits()
# set up the figure
fig = plt.figure(figsize=(6, 6)) # figure size in inches
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
# plot the digits: each image is 8x8 pixels
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')
    # label the image with the target value
    ax.text(0, 7, str(digits.target[i]))
```

Classifying digits

```
In [206]: from sklearn.datasets import load_digits
          digits = load_digits()
          digits.keys()

Out[206]: dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
```

```
In [207]: # set up the figure
          fig = plt.figure(figsize=(6, 6))  # figure size in inches
          fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

          # plot the digits: each image is 8x8 pixels
          for i in range(64):
              ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
              ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

              # label the image with the target value
              ax.text(0, 7, str(digits.target[i]))
```



Classifying digits

We can quickly classify the digits using a random forest as follows:

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(digits.data, digits.target,
random_state=0)
model = RandomForestClassifier(n_estimators=1000)
model.fit(Xtrain, ytrain)
ypred = model.predict(Xtest)
```

Classifying digits

We can take a look at the classification report for this classifier:

```
from sklearn import metrics
print(metrics.classification_report(ypred, ytest))
```

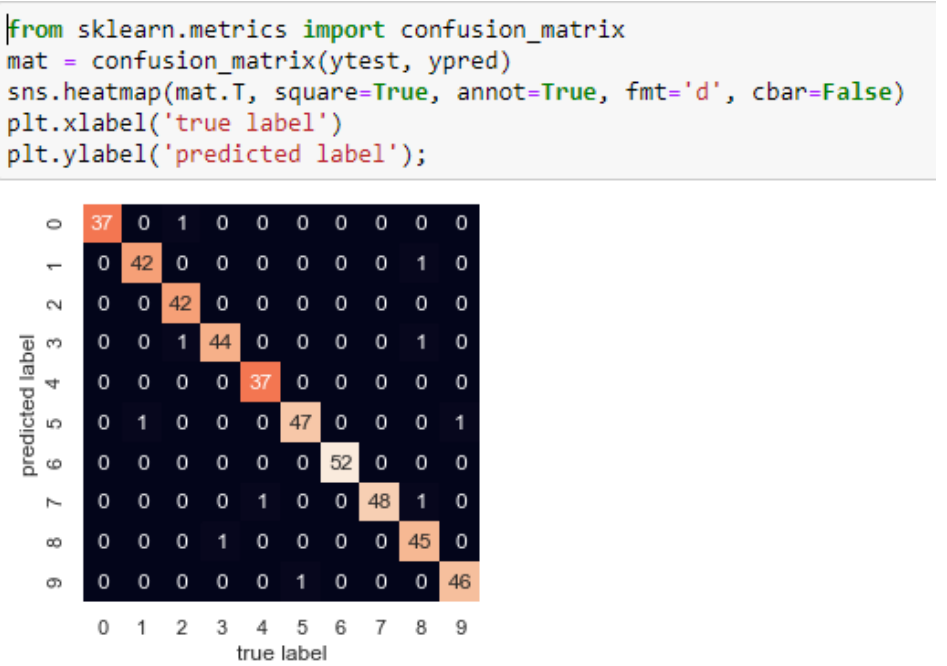
```
from sklearn import metrics
print(metrics.classification_report(ypred, ytest))
```

	precision	recall	f1-score	support
0	1.00	0.97	0.99	38
1	0.98	0.98	0.98	43
2	0.95	1.00	0.98	42
3	0.98	0.96	0.97	46
4	0.97	1.00	0.99	37
5	0.98	0.96	0.97	49
6	1.00	1.00	1.00	52
7	1.00	0.96	0.98	50
8	0.94	0.98	0.96	46
9	0.98	0.98	0.98	47
accuracy			0.98	450
macro avg	0.98	0.98	0.98	450
weighted avg	0.98	0.98	0.98	450

Classifying digits

And for good measure, plot the confusion matrix:

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(ytest, ypred)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False)
plt.xlabel('true label')
plt.ylabel('predicted label');
```



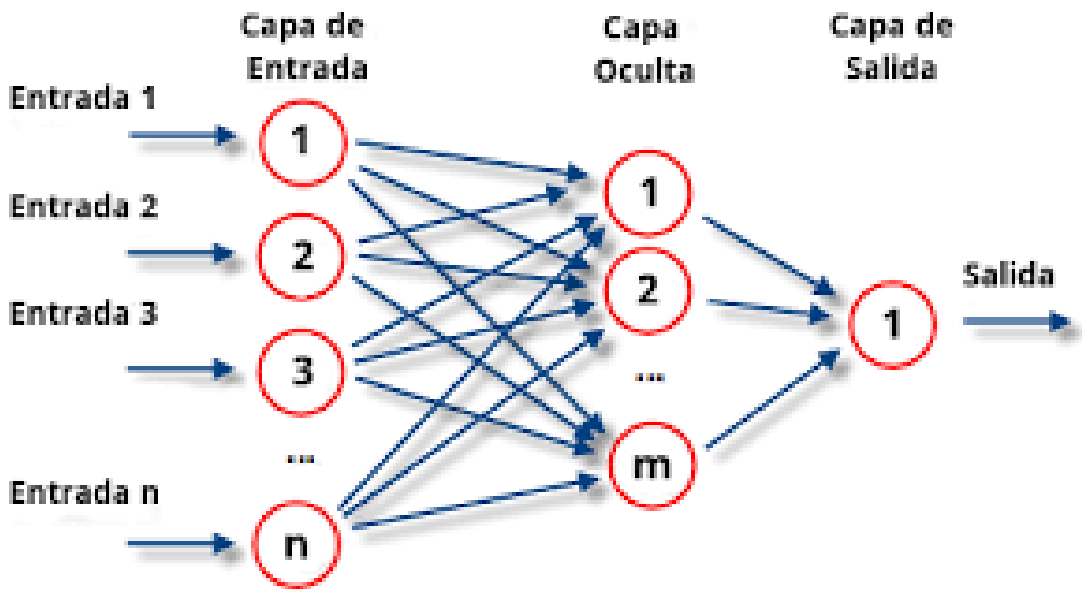
Random Forest Summary

Random forests are a powerful method with several advantages:

- Both training and prediction are very fast, because of the simplicity of the underlying decision trees. In addition, both tasks can be straightforwardly parallelized, because the individual trees are entirely independent entities.
- The multiple trees allow for a probabilistic classification: a majority vote among estimators gives an estimate of the probability (accessed in Scikit-Learn with the predict_proba() method).
- The nonparametric model is extremely flexible, and can thus perform well on tasks that are under-fit by other estimators.

A primary disadvantage of random forests is that the results are not easily interpretable: that is, if you would like to draw conclusions about the *meaning* of the classification model, random forests may not be the best choice.

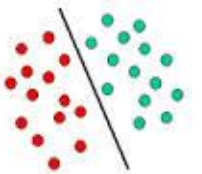
Neuronal Networks



https://scikit-learn.org/stable/modules/neural_networks_supervised.html

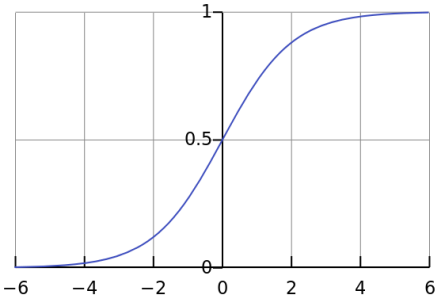
Luís Garmendia

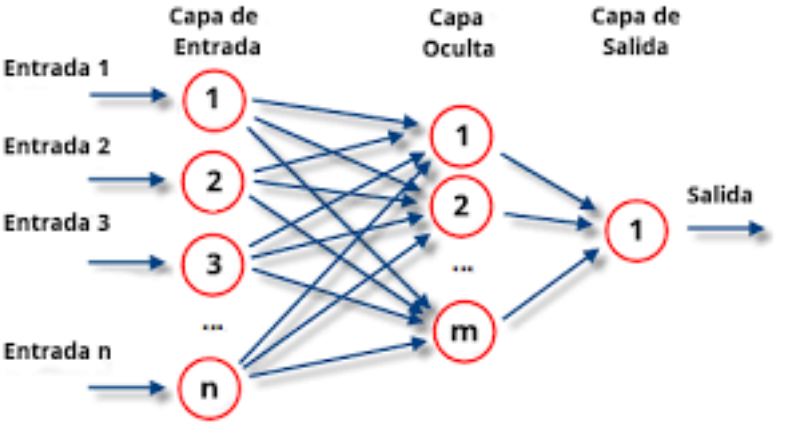
Regresión logística

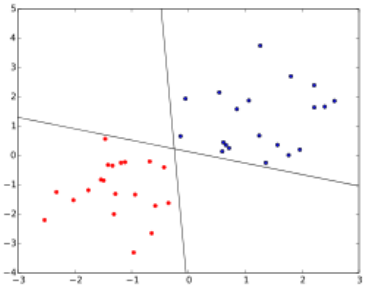


Classification
(LR, SVM…)

La regresión logística se utiliza para entrenar perceptrones de redes neuronales, entrenandolas para ajustar sus pesos y clasificar







Multi-layer Perceptron

scikit-learn.org/stable/modules/neural_networks_supervised.html

scikit learn Install User Guide API Examples More

Prev Up Next

scikit-learn 0.24.2
Other versions

Please **cite us** if you use the software.

1.17. Neural network models (supervised)

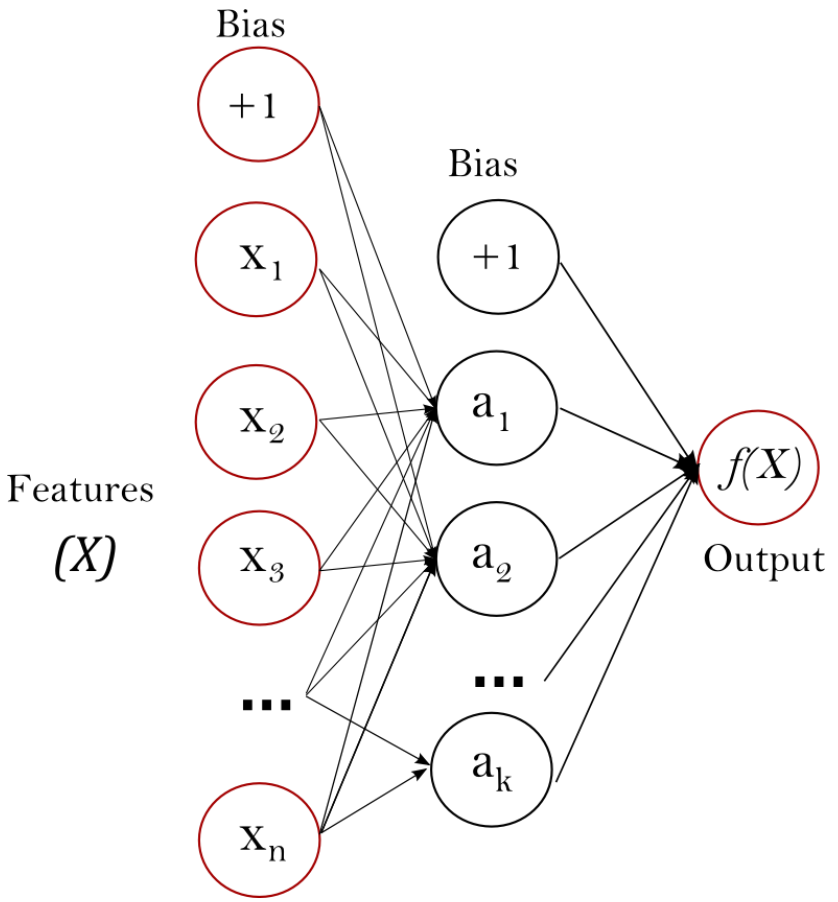
Warning: This implementation is not intended for large-scale applications. In particular, scikit-learn offers no GPU support. For much faster, GPU-based implementations, as well as frameworks offering much more flexibility to build deep learning architectures, see Related Projects.

1.17.1. Multi-layer Perceptron

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^o$ by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output. Given a set of features $X = x_1, x_2, \ldots, x_m$ and a target y, it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. Figure 1 shows a one hidden layer MLP with scalar output.

https://scikit-learn.org/stable/modules/neural_networks_supervised.html

Multi-layer Perceptron

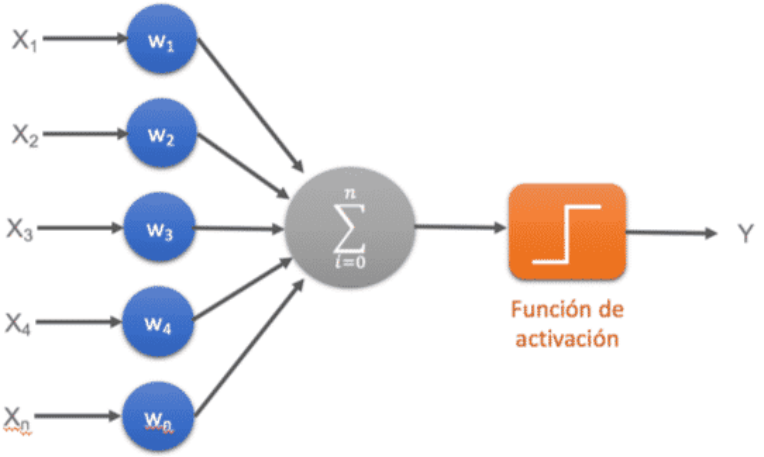


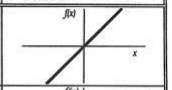
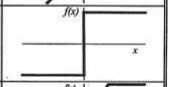
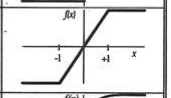
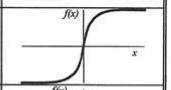
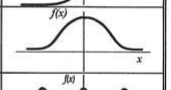
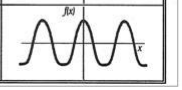
https://scikit-learn.org/stable/modules/neural_networks_supervised.html

Multi-layer Perceptron

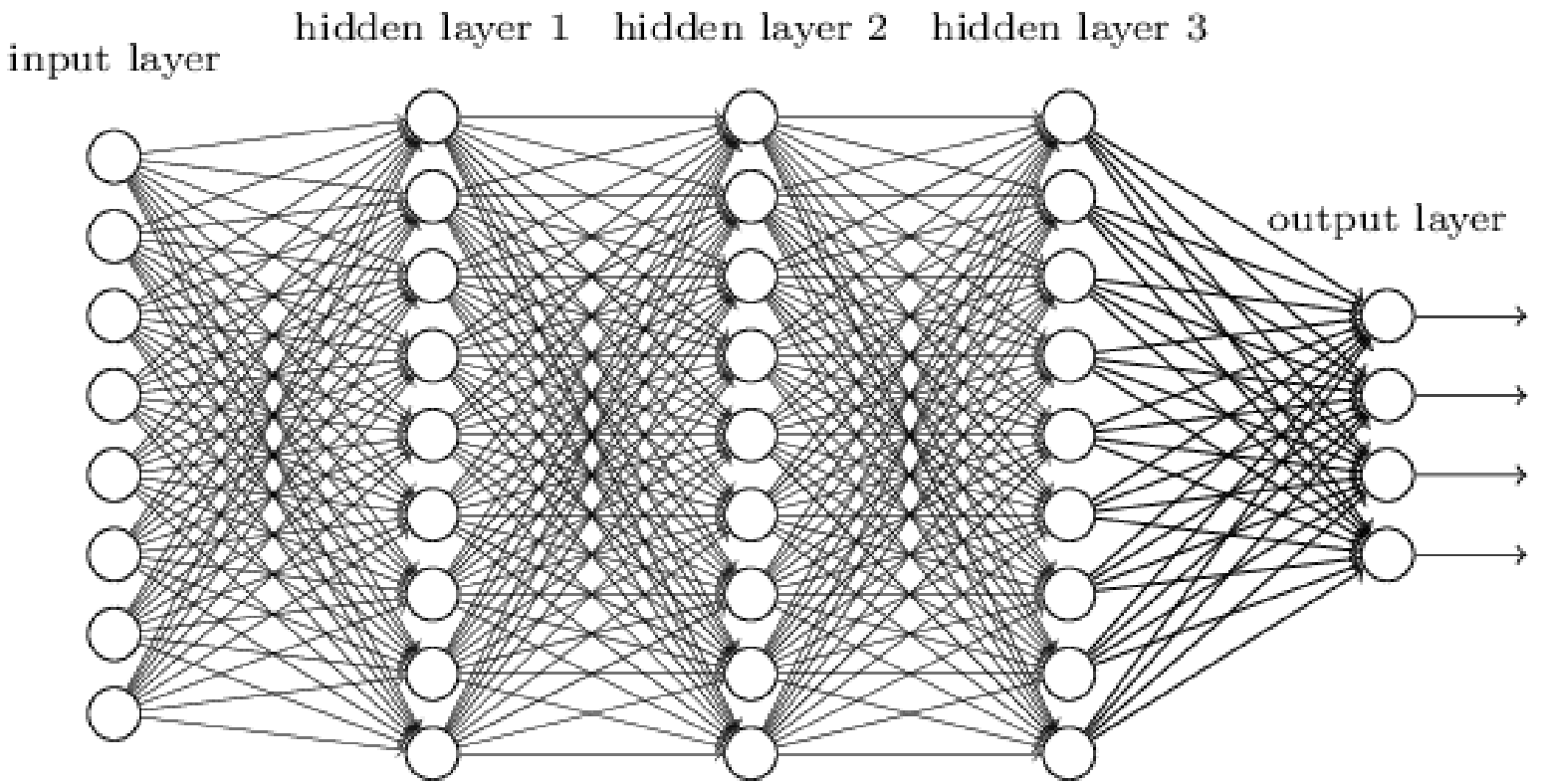
The leftmost layer, known as the input layer, consists of a set of neurons $\{x_i|x_1, x_2, \ldots, x_m\}$ representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $w_1x_1 + w_2x_2 + \ldots + w_mx_m$, followed by a non-linear activation function $g(\cdot) : R \rightarrow R$ - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values.

The module contains the public attributes `coefs_` and `intercepts_`. `coefs_` is a list of weight matrices, where weight matrix at index i represents the weights between layer i and layer $i + 1$. `intercepts_` is a list of bias vectors, where the vector at index i represents the bias values added to layer $i + 1$.



	Función	Rango	Gráfica
Identidad	$y = x$	$[-\infty, +\infty]$	
Escalón	$y = sign(x)$ $y = H(x)$	$\{-1, +1\}$ $\{0, +1\}$	
Lineal a tramos	$y = \begin{cases} -1, & si\ x < -l \\ x, & si\ +l \leq x \leq -l \\ +1, & si\ x > +l \end{cases}$	$[-1, +1]$	
Sigmoidea	$y = \frac{1}{1+e^{-x}}$ $y = tgh(x)$	$[0, +1]$ $[-1, +1]$	
Gaussiana	$y = Ae^{-Bx^2}$	$[0,+1]$	
Sinusoidal	$y = A\,\mathrm{sen}(\omega x + \varphi)$	$[-1,+1]$	

Deep Learning



Neuronal Networks

Class MLPClassifier implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation

MLP trains on two arrays: array X of size (n_samples, n_features), which holds the training samples represented as floating point feature vectors; and array y of size (n_samples,), which holds the target values (class labels) for the training samples:

```
from sklearn.neural_network import MLPClassifier
X = [[0., 0.], [1., 1.]]
y = [0, 1]
clf = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(5, 2),
random_state=1)
clf.fit(X, y)
```

https://scikit-learn.org/stable/modules/neural_networks_supervised.html

Neuronal Networks

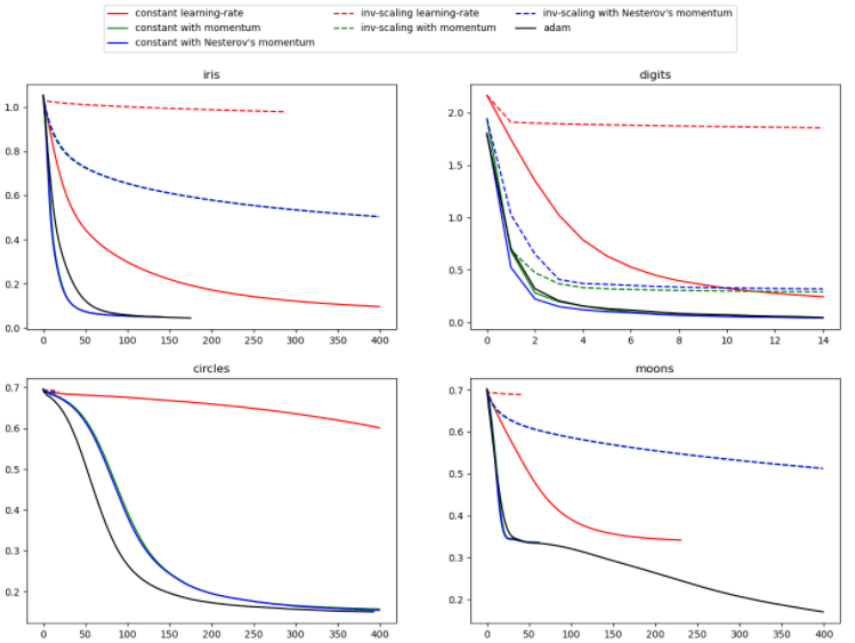
After fitting (training), the model can predict labels for new samples

clf.predict([[2., 2.], [-1., -2.]])

```
In [513]: from sklearn.neural_network import MLPClassifier
          X = [[0., 0.], [1., 1.]]
          y = [0, 1]
          clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
                              hidden_layer_sizes=(5, 2), random_state=1)

          clf.fit(X, y)

Out[513]: MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto', beta_1=0.9,
                        beta_2=0.999, early_stopping=False, epsilon=1e-08,
                        hidden_layer_sizes=(5, 2), learning_rate='constant',
                        learning_rate_init=0.001, max_fun=15000, max_iter=200,
                        momentum=0.9, n_iter_no_change=10, nesterovs_momentum=True,
                        power_t=0.5, random_state=1, shuffle=True, solver='lbfgs',
                        tol=0.0001, validation_fraction=0.1, verbose=False,
                        warm_start=False)

In [514]: clf.predict([[2., 2.], [-1., -2.]])

Out[514]: array([1, 0])
```

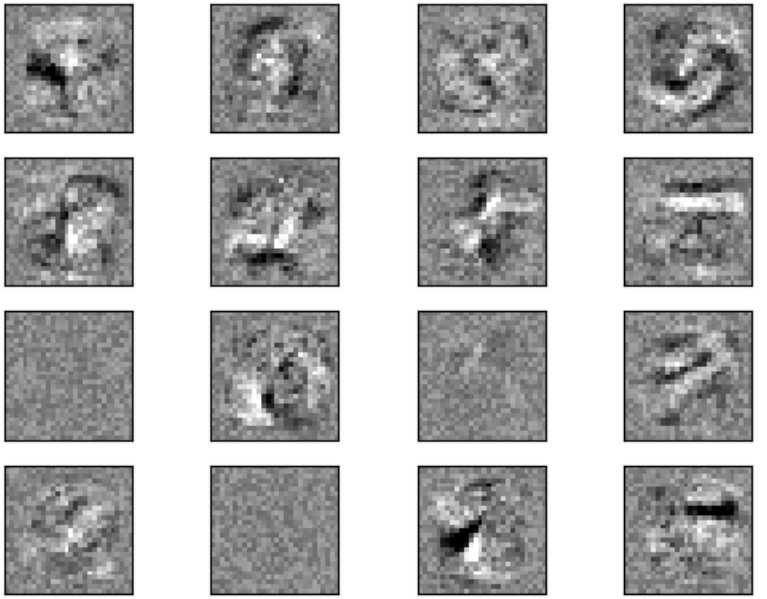
https://scikit-learn.org/stable/modules/neural_networks_supervised.html

Compare Stochastic learning strategies for MLPClassifier



https://scikit-learn.org/stable/auto_examples/neural_networks/plot_mlp_training_curves.html#sphx-glr-auto-examples-neural-networks-plot-mlp-training-curves-py

Visualization of MLP weights on MNIST



https://scikit-learn.org/stable/auto_examples/neural_networks/plot_mnist_filters.html#sphx-glr-auto-examples-neural-networks-plot-mnist-filters-py

MNIST

No es seguro | yann.lecun.com/exdb/mnist/

THE MNIST DATABASE

of handwritten digits

Yann LeCun, Courant Institute, NYU
Corinna Cortes, Google Labs, New York
Christopher J.C. Burges, Microsoft Research, Redmond

Please refrain from accessing these files from automated scripts with high frequency. Make copies!`

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normaliz centered in a fixed-size image.

It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

Four files are available on this site:

```
train-images-idx3-ubyte.gz:  training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:  training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:   test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:   test set labels (4542 bytes)
```

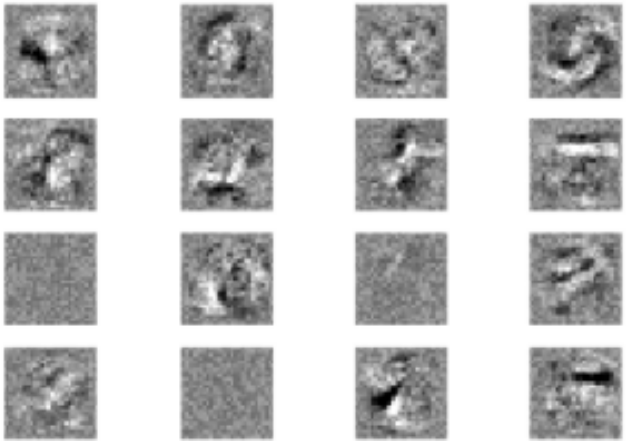
http://yann.lecun.com/exdb/mnist/

Visualization of MLP weights on MNIST

```
import warnings
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.exceptions import ConvergenceWarning
from sklearn.neural_network import MLPClassifier
print(__doc__)
# Load data from https://www.openml.org/d/554
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
X = X / 255.
# rescale the data, use the traditional train/test split
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]

mlp = MLPClassifier(hidden_layer_sizes=(50,), max_iter=10, alpha=1e-4,
                    solver='sgd', verbose=10, random_state=1,
                    learning_rate_init=.1)
# this example won't converge because of CI's time constraints, so we catch the
# warning and are ignore it here
with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=ConvergenceWarning,
                            module="sklearn")
    mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
fig, axes = plt.subplots(4, 4)
# use global min / max to ensure all weights are shown on the same scale
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
               vmax=.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())
plt.show()
```

Visualization of MLP weights on MNIST

```
Automatically created module for IPython interactive environment
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Training set score: 0.986800
Test set score: 0.970000
```



Neuronal Networks

The advantages of Multi-layer Perceptron are:

- Capability to learn non-linear models.
- Capability to learn models in real-time (on-line learning) using partial_fit.

The disadvantages of Multi-layer Perceptron (MLP) include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.
- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.
- Black Box!

https://scikit-learn.org/stable/modules/neural_networks_supervised.html

References

https://ipython.org/

https://numpy.org/

https://jupyter.org/try

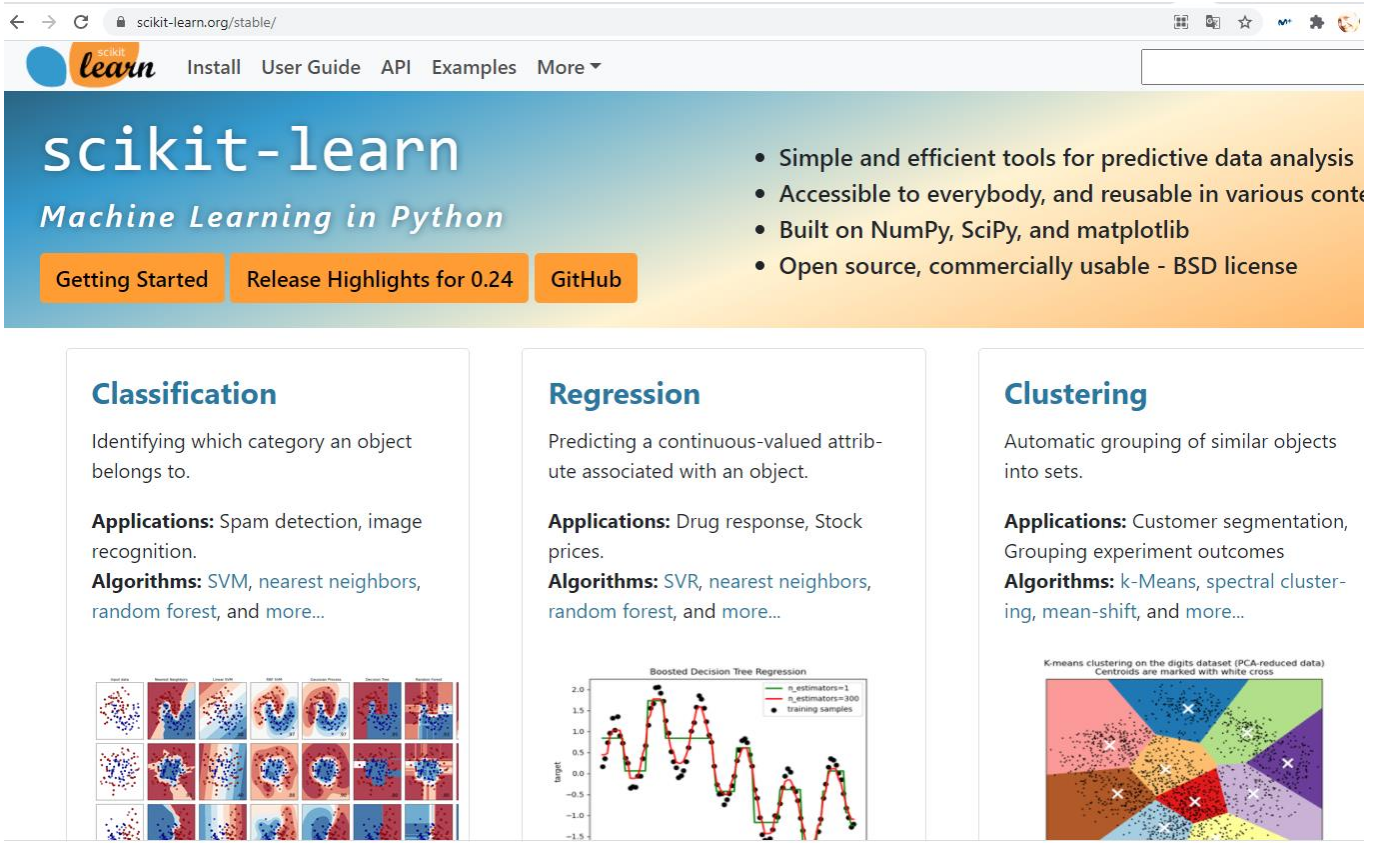
https://pandas.pydata.org/

http://seaborn.pydata.org/

https://scikit-learn.org/stable/

References

https://scikit-learn.org/stable/



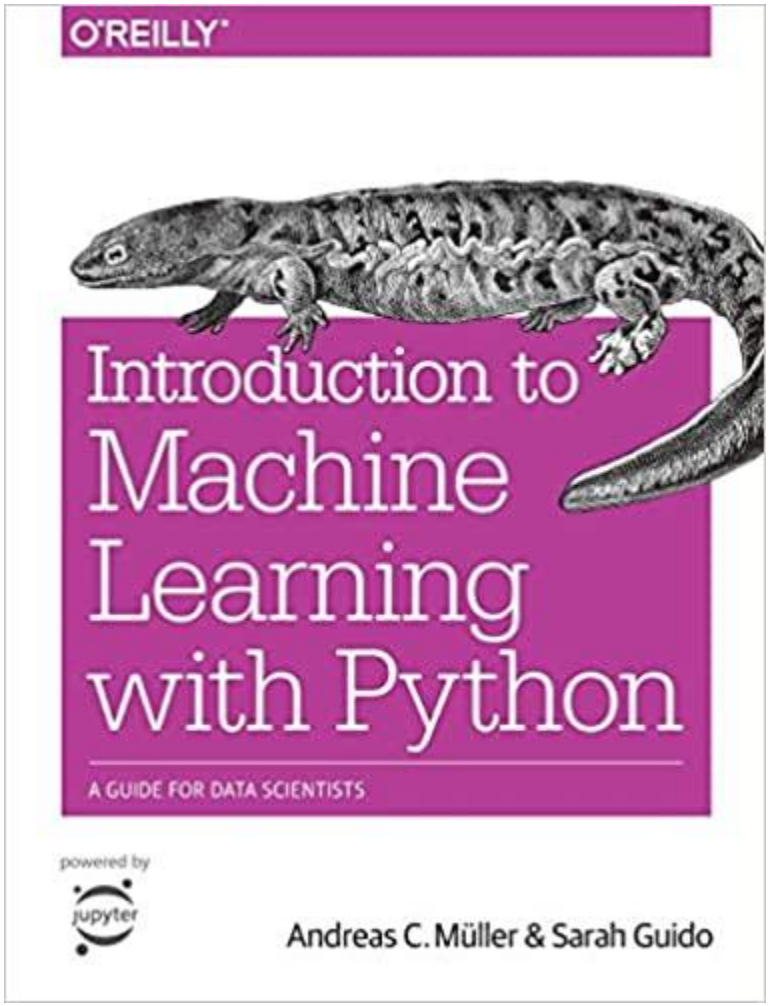
References

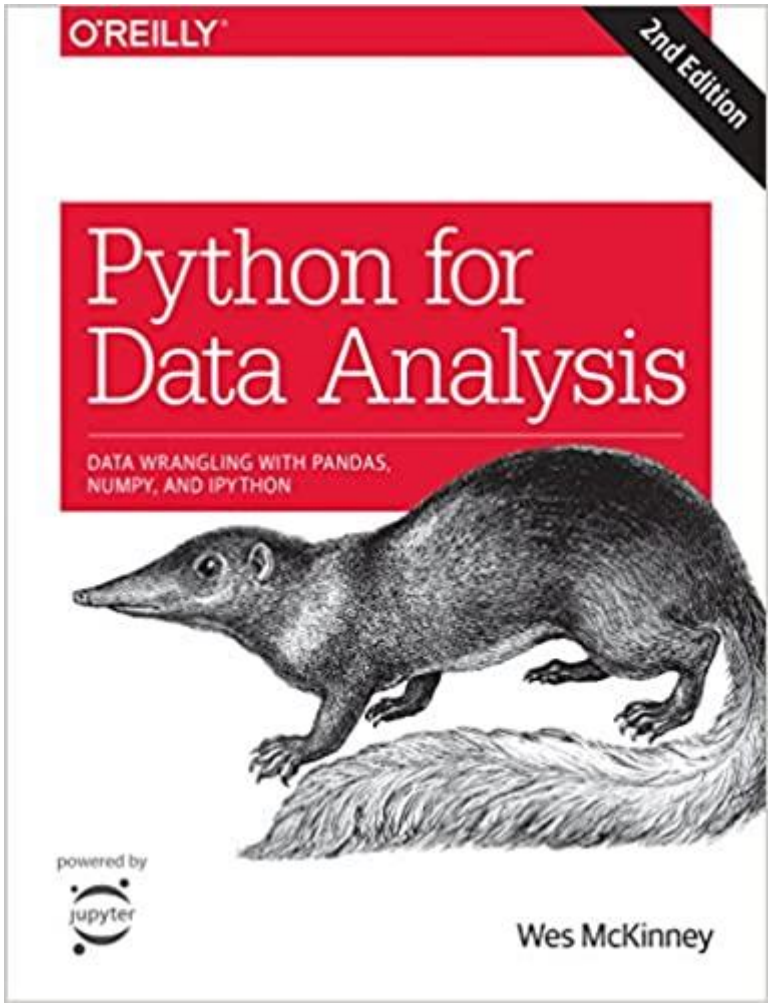
Face recognition example:

https://scikit-learn.org/stable/auto_examples/applications/plot_face_recognition.html#sphx-glr-auto-examples-applications-plot-face-recognition-py

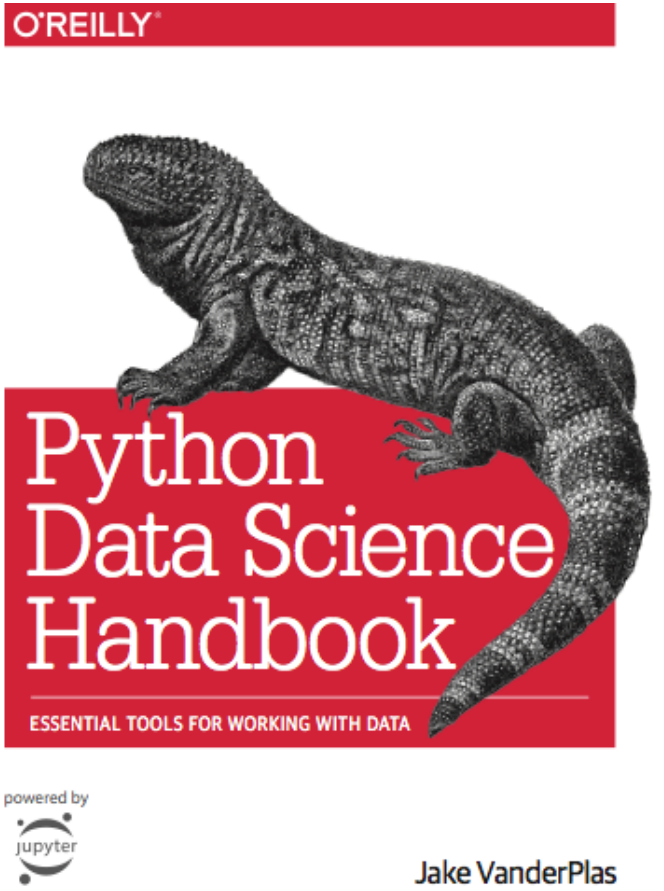
https://jefflirion.github.io/udacity/Intro_to_Machine_Learning/Lesson12.html

References


O'REILLY
Introduction to
Machine
Learning
with Python
A GUIDE FOR DATA SCIENTISTS
powered by
jupyter
Andreas C. Müller & Sarah Guido


O'REILLY
2nd Edition
Python for
Data Analysis
DATA WRANGLING WITH PANDAS,
NUMPY, AND IPYTHON
powered by
jupyter
Wes McKinney

References



https://jakevdp.github.io/PythonDataScienceHandbook/