What is scikit-learn?

scikit-learn is a Python module integrating classic machine learning algorithms in the tightly-knit scientific Python world (numpy, scipy, matplotlib). It aims to provide simple and efficient solutions to learning problems, accessible to everybody and reusable in various contexts: machine-learning as a versatile tool for science and engineering.

- Scikit-learn provides a range of supervised and unsupervised learning algorithms via a consistent interface in Python.
- Scikit-learn is licensed under a permissive simplified BSD license and is distributed under many Linux distributions, encouraging academic and commercial use.

- The library is built upon the **SciPy** (Scientific Python) that must be installed before you can use scikit-learn.
- This stack that includes:

  NumPy: Base n-dimensional array package

  SciPy: Fundamental library for scientific computing

  Matplotlib: Comprehensive 2D/3D plotting

  IPython: Enhanced interactive console

  Sympy: Symbolic mathematics

  Pandas: Data structures and analysis

- Extensions or modules for SciPy are conventionally named SciKits. As such, the module provides learning algorithms is named scikit-learn.
- The vision for the library is a level of robustness and support required for use in production systems. This means a deep focus on concerns such as easy of use, code quality, collaboration, documentation and and performance.
- Although the interface is Python, c-libraries are leverage for performance such as numpy for arrays and matrix operations, LAPACK, LibSVM and the careful use of cython.

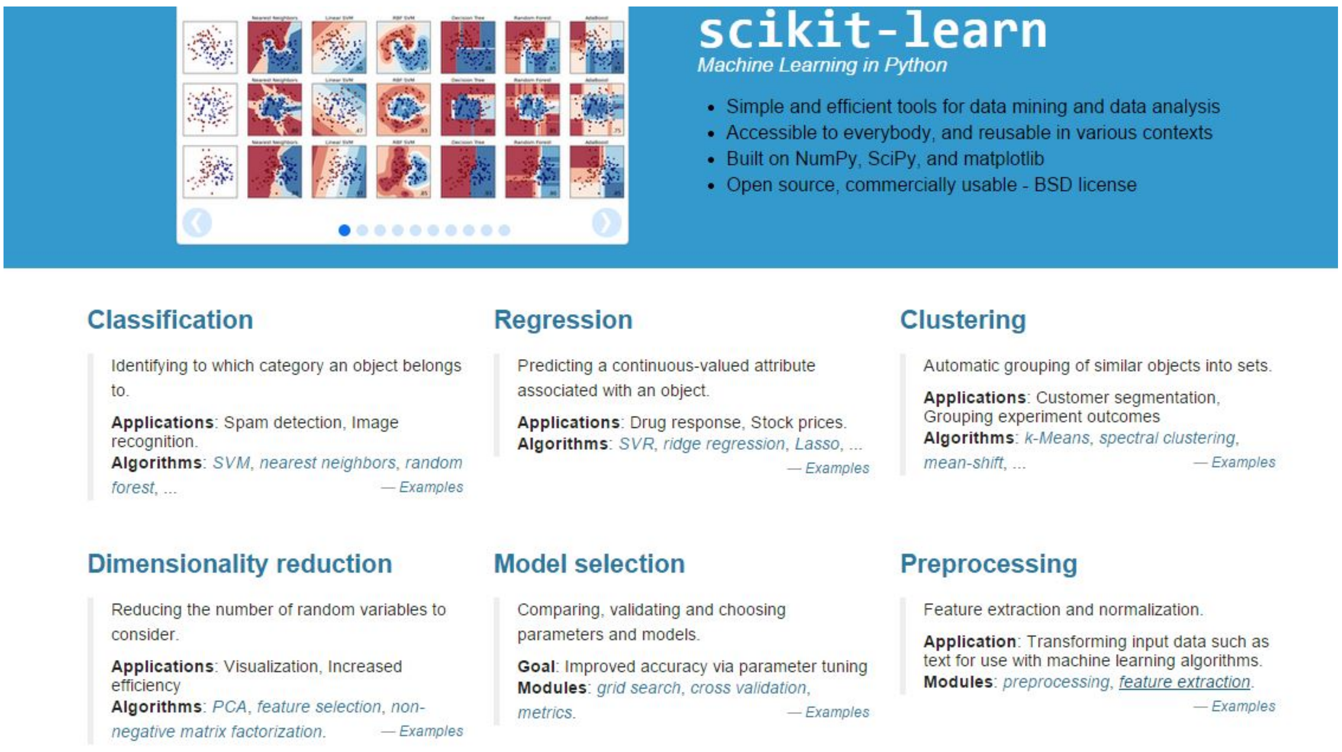
Underlying Technologies

Numpy: the base data structure used for data and model parameters. Input data is presented as numpy arrays, thus integrating seamlessly with other scientific Python libraries.
Numpy's viewbased memory model limits copies, even when binding with compiled code (*Van der Walt et al., 2011*). It also provides basic arithmetic operations.

Scipy: efficient algorithms for linear algebra, sparse matrix representation, special functions and basic statistical functions. Scipy has bindings for many Fortran-based standard numerical packages, such as LAPACK.
This is important for ease of installation and portability, as providing libraries around Fortran code can prove challenging on various platforms.

Cython: a language for combining C in Python. Cython makes it easy to reach the performance of compiled languages with Python-like syntax and high-level operations. It is also used to bind compiled libraries, eliminating the boilerplate code of Python/C extensions.

What does Scikit Learn do?



scikit-learn

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying to which category an object belongs to.

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

Regression

Predicting a continuous-valued attribute associated with an object.

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

Clustering

Automatic grouping of similar objects into sets.

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

Dimensionality reduction

Reducing the number of random variables to consider.

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

Model selection

Comparing, validating and choosing parameters and models.

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

Preprocessing

Feature extraction and normalization.

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

1. **Classification**
    - **Description:** Identifying to which category an object belongs to.
    - **Applications:** Spam detection, Image recognition.
    - **Algorithms:** SVM, nearest neighbors, random forest,
2. **Regression**
    - **Description:** Predicting a continuous-valued attribute associated with an object.
    - **Applications:** Drug response, Stock prices.
    - **Algorithms:** SVR, ridge regression, Lasso,
3. **Clustering**
    - **Description:** Automatic grouping of similar objects into sets.
    - **Applications:** Customer segmentation, Grouping experiment outcomes
    - **Algorithms:** k-Means, spectral clustering, mean-shift, ...

4. Dimensionality Reduction

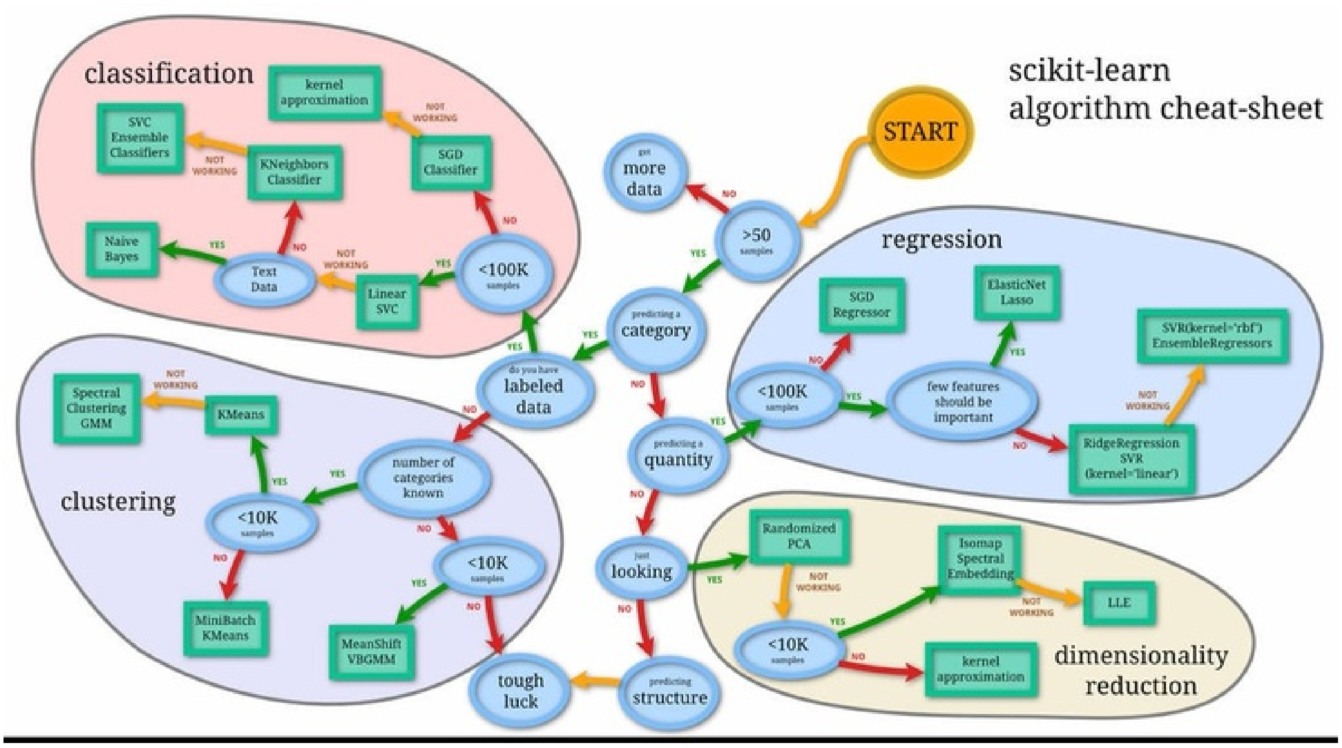
- **Description:** Reducing the number of random variables to consider.
- **Applications:** Visualization, Increased efficiency
- **Algorithms:** PCA, feature selection, non-negative matrix factorization.

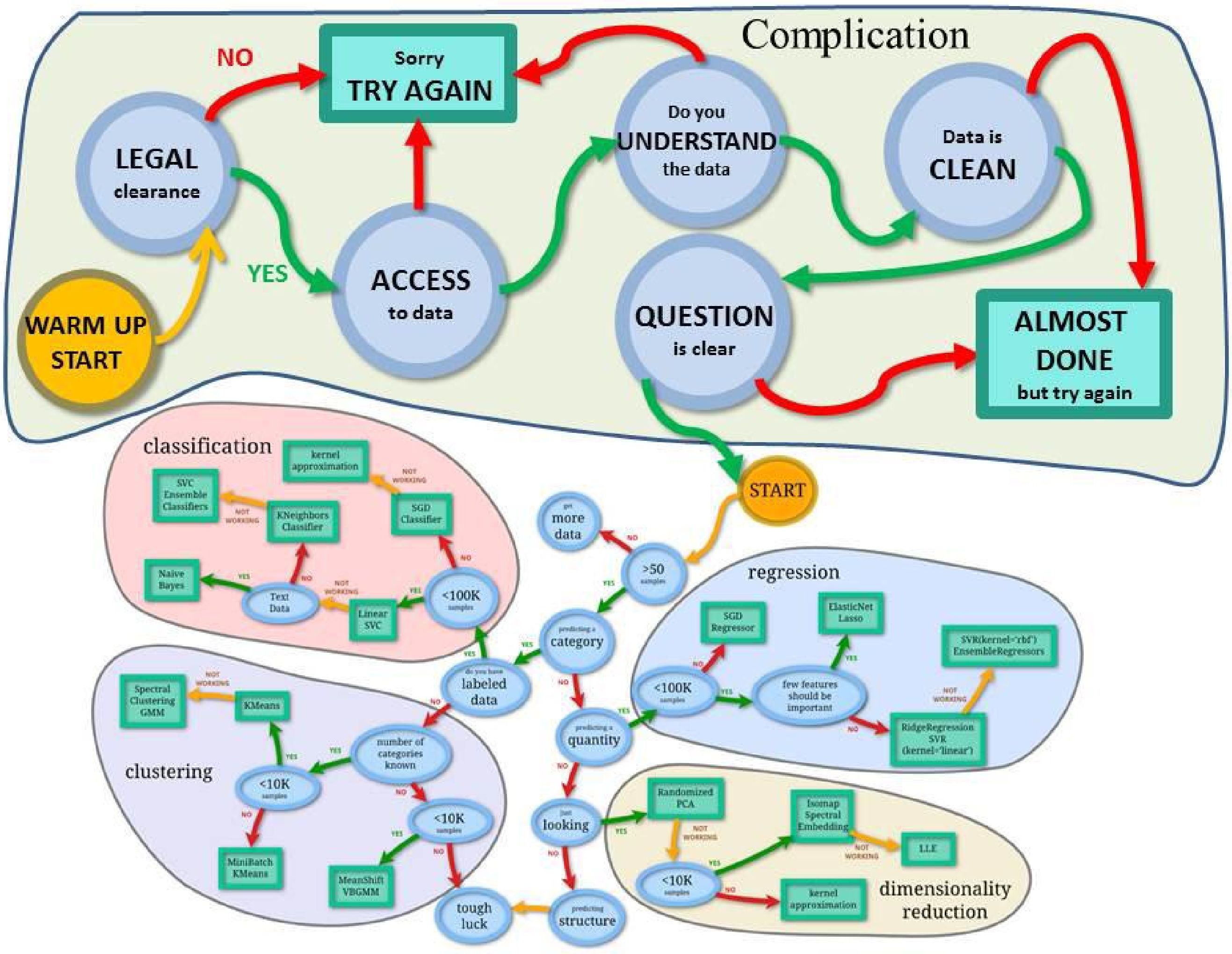
5. Model selection

- **Description:** Comparing, validating and choosing parameters and models.
- **Goal:** Improved accuracy via parameter tuning
- **Modules:** grid search, cross validation, metrics

6. Preprocessing

- **Description:** Feature extraction and normalization.
- **Application:** Transforming input data such as text for use with machine learning algorithms.
- **Modules:** preprocessing, feature extraction.


scikit-learn
algorithm cheat-sheet
START
>50 samples
get more data
predicting a category
do you have labeled data
<100K samples
SGD Classifier
kernel approximation
Linear SVC
Text Data
Naive Bayes
KNeighbors Classifier
SVC Ensemble Classifiers
classification
number of categories known
<10K samples
KMeans
Spectral Clustering GMM
MiniBatch KMeans
MeanShift VBGMM
clustering
tough luck
predicting a quantity
few features should be important
SGD Regressor
ElasticNet Lasso
RidgeRegression SVR (kernel='linear')
SVR(kernel='rbf') EnsembleRegressors
regression
just looking
predicting structure
Randomized PCA
Isomap Spectral Embedding
LLE
kernel approximation
dimensionality reduction
YES
NO
NOT WORKING


Complication
WARM UP START
LEGAL clearance
NO
Sorry TRY AGAIN
YES
ACCESS to data
Do you UNDERSTAND the data
Data is CLEAN
QUESTION is clear
ALMOST DONE but try again
START
>50 samples
get more data
predicting a category
do you have labeled data
classification
<100K samples
SGD Classifier
kernel approximation
Linear SVC
Text Data
Naive Bayes
KNeighbors Classifier
SVC Ensemble Classifiers
NOT WORKING
clustering
number of categories known
<10K samples
KMeans
Spectral Clustering GMM
MiniBatch KMeans
MeanShift VBGMM
tough luck
predicting a quantity
regression
SGD Regressor
few features should be important
ElasticNet Lasso
RidgeRegression SVR (kernel='linear')
SVR(kernel='rbf') EnsembleRegressors
just looking
predicting structure
dimensionality reduction
Randomized PCA
Isomap Spectral Embedding
LLE
kernel approximation

Machine learning: the problem setting

In general, a learning problem considers a set of n samples of data and try to predict properties of unknown data. If each sample is more than a single number, and for instance a multi-dimensional entry (aka multivariate data), is it said to have several attributes, or features.

We can separate learning problems in a few large categories:

- **Supervised learning**, in which the data comes with additional attributes that we want to predict.
  This problem can be either:

  Classification: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data.
  An example of classification problem would be the digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories.

  Regression: if the desired output consists of one or more continuous variables, then the task is called regression.
  An example of a regression problem would be the prediction of the weight of a pony as a function of its age and height.

- **Unsupervised learning**, in which the training data consists of a set of input vectors x without any corresponding target values.
  The goal in such problems may be
  - to discover groups of similar examples within the data, where it is called ***clustering***,
  - to determine the distribution of data within the input space, known as ***density estimation***,
  - to project the data from a high-dimensional space down to two or thee dimensions for the purpose of visualization

Estimators objects: Fitting data:
The core object of scikit-learn is the estimator object. All estimator objects expose a `fit` method, that takes as input a dataset (2D array):

```
>>> estimator.fit(data)
```

Suppose `LogReg` and `KNN` are (shorthand names for) scikit-learn estimators.

```
>>> # Supervised Learning Problem
>>> LogReg.fit(SAheartFeat, SAheartTarget)
>>>
>>> # Unsupervised Learning Problem
>>> KNN.fit(IrisFeat)
```

Estimator parameters:
All the parameters of an estimator can be set when it is instanciated, or by modifying the corresponding attribute:

```
>>> estimator = Estimator(param1=1, param2=2)
>>> estimator.param1
```

Retrieving Estimator parameters:

- When data is fitted with an estimator, parameters are estimated from the data at hand.
- All the estimated parameters are attributes of the estimator object ending by an underscore:

```
>>> estimator.estimated_param_
```

Layout of Datasets

The scikit-learn deals with learning information from one or more datasets that are represented as 2D arrays. They can be understood as a list of multi-dimensional observations. We say that the first axis of these arrays is the samples axis, while the second is the features axis.
When the data is not intially in the (`n_samples, n_features`) shape, it needs to be preprocessed to be used by the scikit.

Packaged Datasets

The scikit-learn library is packaged with datasets. These datasets are useful for getting a handle on a given machine learning algorithm or library feature before using it in your own work.

A simple example shipped with the scikit: iris dataset

```
>>> from scikits.learn import datasets
>>> iris = datasets.load_iris()
>>> data = iris.data
>>> data.shape
(150, 4)
```

Iris is made of 150 observations of irises, each described by 4 features: their sepal and petal length and width, as detailed in `iris.DESCR`.

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

## -  Load the packaged iris flowers dataset
## - Iris flower dataset
## - (4x150, reals, multi-label classification)

iris = load_iris()
print(iris)
iris.keys()
```

Classifying irises

The iris dataset is a classification task consisting in identifying 3 different types of irises (Setosa, Versicolour, and Virginica) from their petal and sepal length and width:

```
>>> import numpy as np
>>> from sklearn import datasets
>>> iris = load_iris()
>>>
>>> iris_X = iris.data
>>> iris_y = iris.target
>>>
>>> np.unique(iris_y)

array([0, 1, 2])
>>>
>>> # Three Classes (Species)
```

Split iris data in train and test data A random permutation, to split the data randomly

```
>>> np.random.seed(0)
>>> indices = np.random.permutation(len(iris_X))
>>> iris_X_train = iris_X[indices[:-10]]
>>> iris_y_train = iris_y[indices[:-10]]
>>> iris_X_test  = iris_X[indices[-10:]]
>>> iris_y_test  = iris_y[indices[-10:]]
>>> # Create and fit a nearest-neighbor classifier
>>> from scikits.learn.neighbors import NeighborsClassif
>>> knn = NeighborsClassifier()
>>> knn.fit(iris_X_train, iris_y_train)
NeighborsClassifier(n_neighbors=5, leaf_size=20, algorit
>>> knn.predict(iris_X_test)
array([1, 2, 1, 0, 0, 0, 2, 1, 2, 0])
>>> iris_y_test
array([1, 1, 1, 0, 0, 0, 2, 1, 2, 0])
```

0.0.1 k-Nearest neigbhors classifier

The simplest possible classifier is the nearest neighbor: given a new observation `x_test`, find in the training set (i.e. the data used to train the estimator) the observation with the closest feature vector.

Load from CSV

- In most of the Scikit-learn algorithms, the data must be loaded as a `Bunch` Object.
- However there are many example in the tutorial where `load_files()` or other functions are used to populate the bunch object.
- Function like `load_files()` expect data to be present in certain format. Suppose we have a different format in which data is stored.
- It is very common to have a dataset as a CSV file on the local workstation or on a remote server.
- You load a CSV file from a URL, in this case the Pima Indians diabetes classification dataset from the UCI Machine Learning Repository.
- From the prepared `X` and `y` variables, you can train a machine learning model.

```
# Pima Indians diabetes
# Load the  dataset from CSV URL

import numpy as np
import urllib

# URL for the Pima Indians Diabetes dataset
# (UCI Machine Learning Repository)

url = "http://goo.gl/j0Rvxq"

# download the file
raw_data = urllib.urlopen(url)

# load the CSV file as a numpy matrix

dataset = np.loadtxt(raw_data, delimiter=",")
print(dataset.shape)

# separate the data from the target attributes
X = dataset[:,0:7]
y = dataset[:,8]
```

Logistic Regression

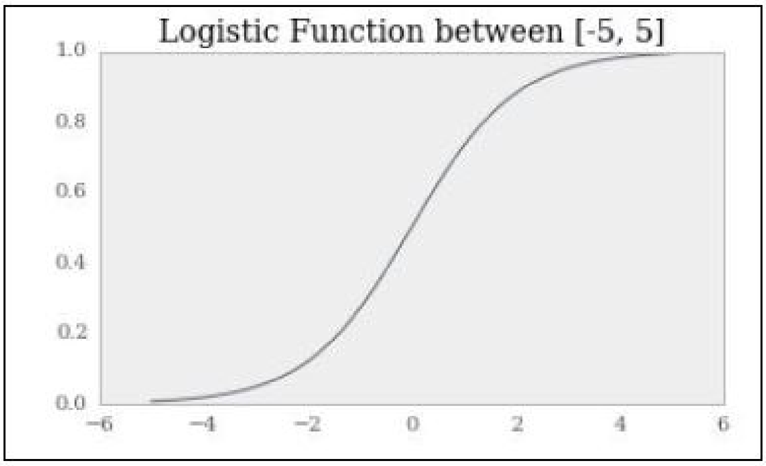
Linear models can actually be used for classification tasks. This involves fitting a linear model to the probability of a certain class, and then using a function to create a threshold at which we specify the outcome of one of the classes.

The Logistic Function

The function used here is typically the logistic function. The logistic function $f(t)$ is defined as follows:

$$f(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}},$$

Visually, it looks like the following:



Worked Example

Let's use the `make_classification` method, create a dataset, and get to classifying:

```
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4)
```

Implementation

The `LogisticRegression` object works in the same way as the other linear models:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression()
```

- We will use the last 200 samples to test the trained model on.
- Since this is a random dataset, it's fine to hold out the last 200.
- If you're dealing with structured data, don't do this (for example, if you deal with time series data):

```
>>> X_train = X[:-200]
>>> X_test = X[-200:]
>>> y_train = y[:-200]
>>> y_test = y[-200:]
```

Now, we need to fit the model with logistic regression. We'll keep around the predictions on the train set, just like the test set. It's a good idea to see how often you are correct on both sets.

```
>>> lr.fit(X_train, y_train)
>>> y_train_pred = lr.predict(X_train)
>>> y_test_pred = lr.predict(X_test)
```

Model Evaluation

- Now that we have the predictions, let's take a look at how good our predictions were.
- Here, we'll simply look at the number of times we were correct.
- The calculation is simple; it's the number of times we were correct over the total sample:

```
>>> (y_train_pred == y_train).sum().astype(float) / y_train.shape[0]

0.8662499
```

Similarly for the test sample:

```
>>> (y_test_pred == y_test).sum().astype(float) / y_test.shape[0]

0.900000
```

Skip This

- The question then changes to how to move on from the logistic function to a method by which we can classify groups.
- First, recall the linear regression hopes offending the linear function that fits the expected value of Y, given the values of X; this is $E(Y|X) = X\beta$. Here, the Y values are the probabilities of the classes.
- Therefore, the problem we're trying to solve is $E(p|X) = X\beta$. Then, once the threshold is applied, this becomes $\text{Logit}(p) = X\beta$.
- The idea expanded is how other forms of regression work, for example, Poisson.

Class Imbalance

- There will be a situation where one class is weighted differently from the other classes; for example, one class may be 99 percent of cases.
- This situation will occur regularly in real world data science.
- The canonical example is fraud detection, where most transactions aren't fraud, but the cost associated with misclassification is asymmetric between classes.

Let's create a classification problem with 95 percent imbalance and see how the basic stock logistic regression handles this case:

```
>>> X, y = make_classification(n_samples=5000,
n_features=4,
weights=[.95])
>>>
>>>  #to confirm the class imbalance
>>> sum(y) / (len(y)*1.)
0.0555
```

Create the train and test sets, and then fit logistic regression:

```
>>> X_train = X[:-500]
>>> X_test = X[-500:]
>>> y_train = y[:-500]
>>> y_test = y[-500:]
>>>
>>> lr.fit(X_train, y_train)
>>> y_train_pred = lr.predict(X_train)
>>> y_test_pred = lr.predict(X_test)
```

Now, to see how well our model fits the data, do the following:

```
>>> (y_train_pred == y_train).sum().astype(float) / y_train.shape[0]

0.96977
>>> (y_test_pred== y_test).sum().astype(float) / y_test.shape[0]

0.97999
```

- At first, it looks like we did well, but it turns out that when we always guessed that a transaction was not fraud (or class 0 in general) we were right around 95 percent of the time.
- If we look at how well we did in classifying the 1 class, it's not nearly as good:

```
>>> (y_test[y_test==1] == y_test_pred[y_test==1])
.sum().astype(float) / y_test[y_test==1].shape[0]

0.583333
```

- Hypothetically, we might care more about identifying fraud cases than non-fraud cases; this could be due to a business rule, so we might alter how we weigh the correct and incorrect values.
- By default, the classes are weighted (and thus resampled) in accordance with the inverse of the class weights of the training set. However, because we care more about fraud cases, let's oversample the fraud relative to non-fraud cases.
- We know that our relative weighting right now is 95 percent nonfraud; let's change this to overweight fraud cases:

```
>>> lr = LogisticRegression(class_weight={0:.15, 1:.85})
>>> lr.fit(X_train, y_train)
```

Let's predict the outputs again:

```
>>> y_train_pred = lr.predict(X_train)
>>> y_test_pred = lr.predict(X_test)
```

We can see that we did a much better job on classifying the fraud cases:

```
>>> (y_test[y_test==1] == y_test_pred[y_test==1]).sum().
astype(float) / y_test[y_test==1].shape[0]
0.875
```

At what expense do we do this? To find out, use the following command:

```
>>> (y_test_pred == y_test).sum().astype(float) / y_test.shape[0]
0.967999
```

- Here, there's only about 1 percent less accuracy. Whether thatis acceptable depends on your problem.
- Put in the context of the problem, if the estimated cost associated with fraud is sufficiently large, it can eclipse the cost associated with tracking fraud.

Evaluating Models with ROC Curves

- Receiving Operating Characteristic, or ROC, is a visual way for inspecting the performance of a binary classifier .
- In particular, it's comparing the rate at which your classifier is making correct predictions (True Positives or TP) and the rate at which your classifier is making false alarms (False Positives or FP).
- When talking about True Positive Rate (TPR) or False Positive Rate (FPR) we're referring to the definitions below:

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

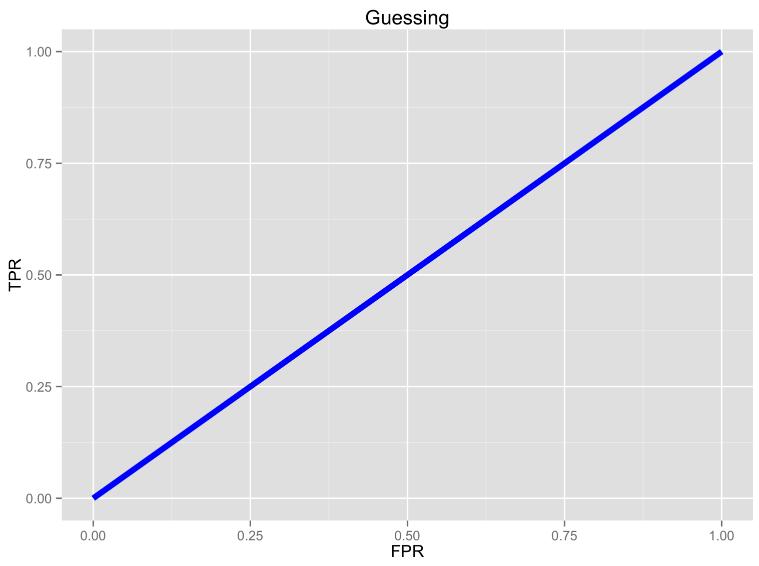
$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

Background

- ROC curves were first used during WWII to analyze radar effectiveness.
- In the early days of radar, it was sometimes hard to tell a bird from a plane.
- The British pioneered using ROC curves to optimize the way that they relied on radar for detecting incoming German planes.

Guessing at Random

The first example is the simplest: a diagonal line. A diagonal line indicates that the classifier is just making completely random guesses. Since your classifier is only going to be correct 50% of the time, it stands to reason that your TPR and FPR will also be equal.

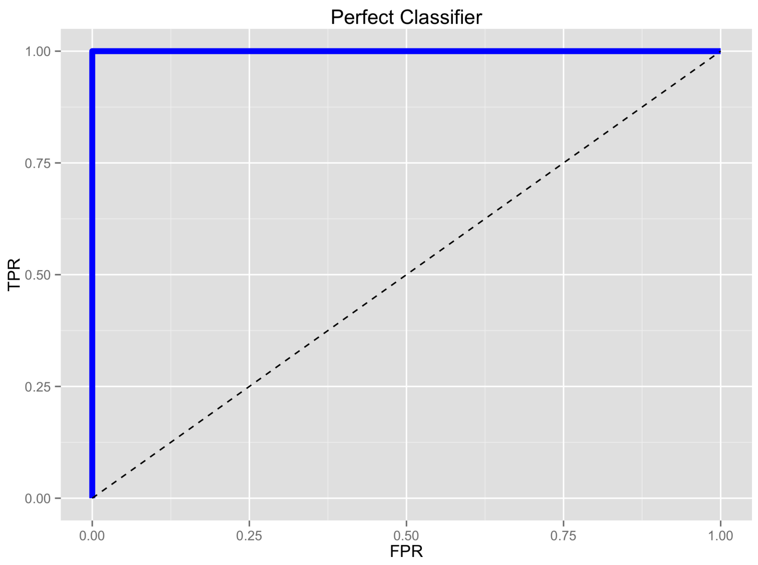


Often, ROC charts will include the random ROC curve to provide the user with a benchmark for what a naive classifier would do. Any curves above the line are better than guessing, while those below the line, you would be better off guessing.

The Area Under the Curve (AUC) is 0.500.

A Perfect Classifier

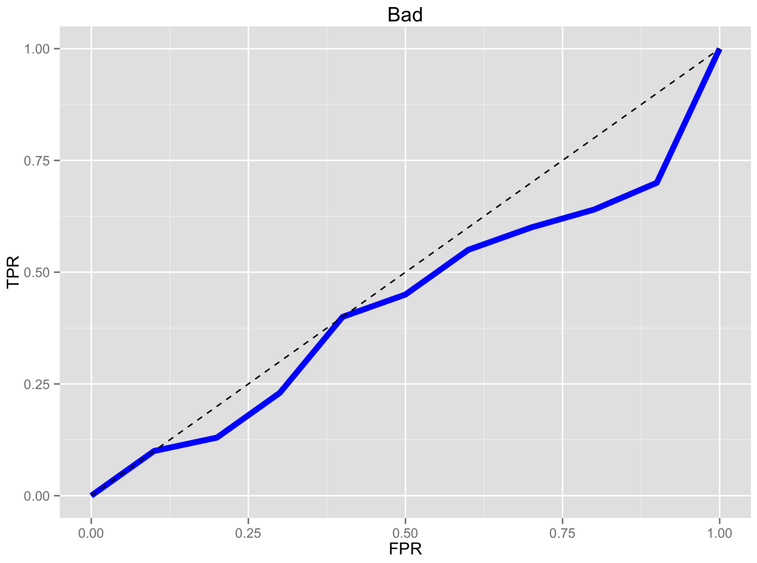
A perfect classifier will yield a perfect trade-off between TPR and FPR (meaning you'll have a TPR of 1 and an FPR of 0). In that case, your ROC curve looks something like this.



Note the "random curve" is included as a benchmark as a dotted line. The Area Under the Curve (AUC) is 1.

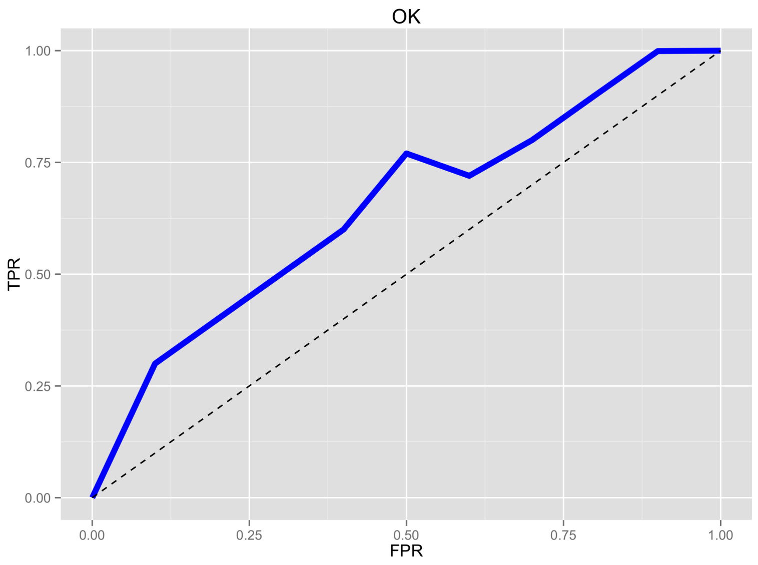
Worse than guessing

A bad classifier (i.e. something that's worse than guessing) will appear below the random line.

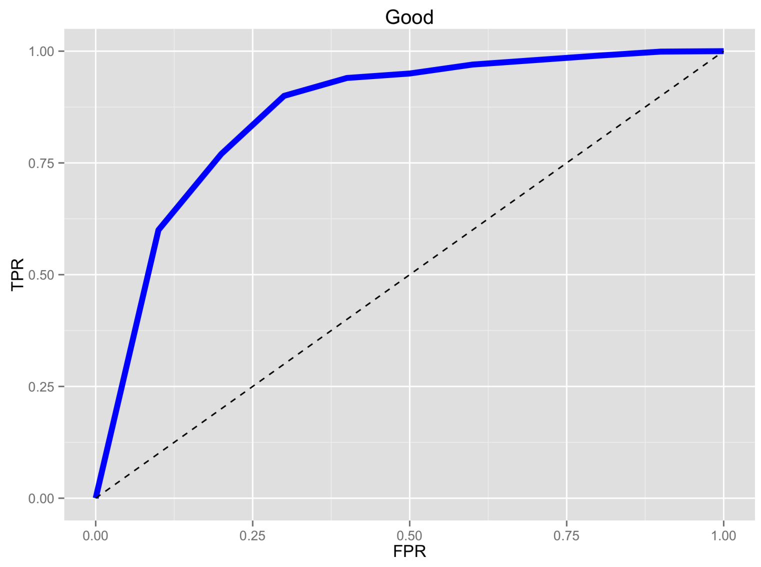


Better than guessing

A much more interesting activity is attempting to decipher the difference between an ”OK” and a ”Good” classifier. The chart below shows an example of a very mediocre classifier.



Pretty good



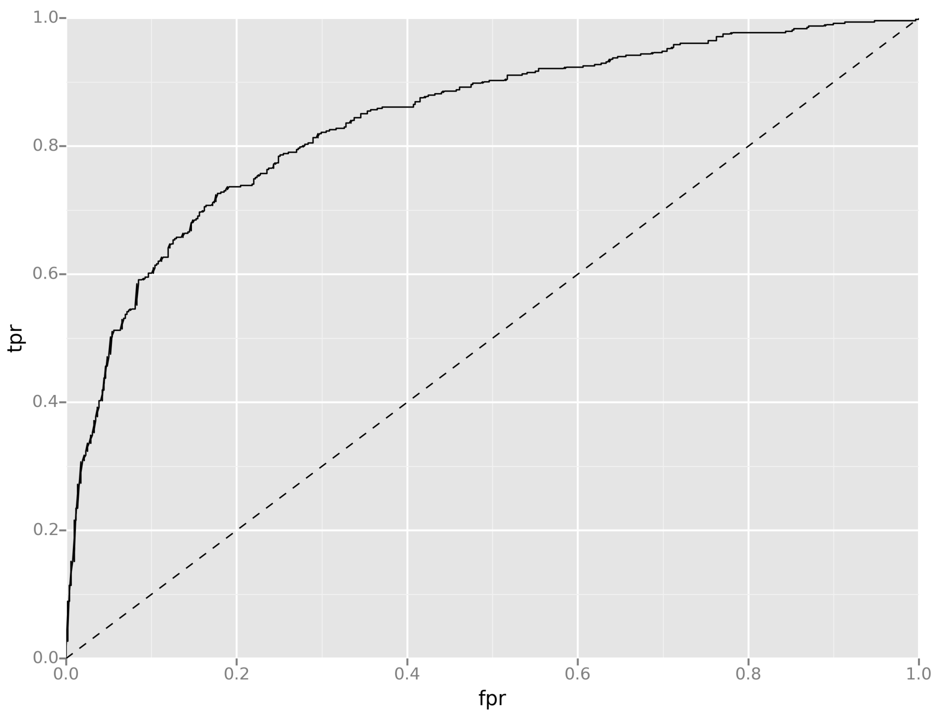
Calculating an ROC Curve in Python

scikit-learn makes it easy to calculate ROC Curves. Firstly, we need a classification model to evaluate. For this example, we will make a synthetic dataset and then build a logistic regression model using scikit-learn.

```
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression

X, y = make_classification(n_samples=10000,
         n_features=10, n_classes=2,
         n_informative=5)

Xtrain = X[:9000]
Xtest = X[9000:]
ytrain = y[:9000]
ytest = y[9000:]
```

```
clf = LogisticRegression()
clf.fit(Xtrain, ytrain)
```

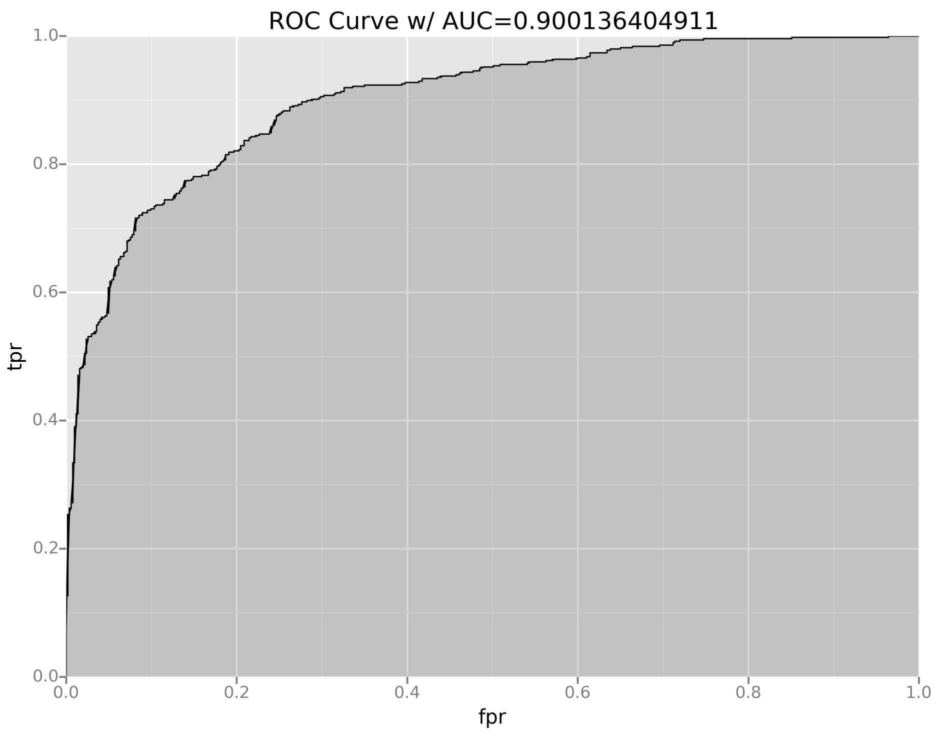
```
from sklearn import metrics
import pandas as pd
from ggplot import *

preds = clf.predict_proba(Xtest)[:,1]

fpr, tpr, _ = metrics.roc_curve(ytest, preds)

df = pd.DataFrame(dict(fpr=fpr, tpr=tpr))

ggplot(df, aes(x='fpr', y='tpr')) +
    geom_line() + geom_abline(linetype='dashed')
```



Finally to calculate the AUC:

```
auc = metrics.auc(fpr,tpr)
ggplot(df, aes(x='fpr', ymin=0, ymax='tpr')) +
    geom_area(alpha=0.2) +
    geom_line(aes(y='tpr')) +
    ggtitle("ROC Curve w/ AUC=%s" % str(auc))
```

The AUC is 0.900.



Classification with scikit-learn

- Here we will look into the problem of classification, a situation in which a response is a **categorical variable**.
- We will introduces a number of classification techniques, and convey their corresponding strengths and weaknesses by visually inspecting the decision boundaries for each model.
    1. Logistic Regression
    2. Linear Discriminant Analysis
    3. Knearest Neighbour
- Here we will use **scikit-learn**, an easy-to-use, general-purpose toolbox for machine learning in Python

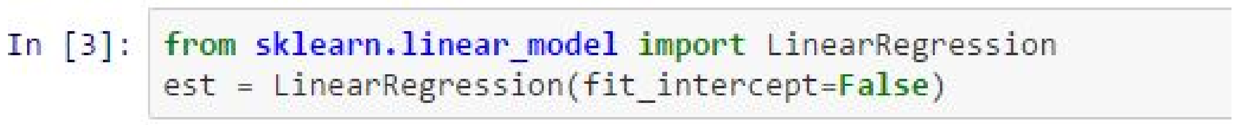
Scikit-learn

- Scikit-learn is a library that provides a variety of both supervised and unsupervised machine learning techniques.
- **Supervised machine learning** refers to the problem of inferring a function from labeled training data, and it comprises both regression and classification.
- **Unsupervised machine learning**, on the other hand, refers to the problem of finding interesting patterns or structure in the data; it comprises techniques such as clustering and dimensionality reduction.
- In addition to statistical learning techniques, scikit-learn provides utilities for common tasks such as model selection, feature extraction, and feature selection.

Estimators

- Scikit-learn provides an object-oriented interface centered around the concept of an **Estimator**.
- According to the scikit-learn tutorial :
  "*An estimator is any object that learns from data; it may be a classification, regression or clustering algorithm or a transformer that extracts/filters useful features from raw data.*"
- Usually, the data is comprised of a two-dimensional numpy array `X` of shape `(n_samples, n_predictors)` , (*in other words number of rows and columns*) that holds the so-called **feature matrix** and a one-dimensional numpy array `y` that holds the responses.
- The `Estimator.fit` method sets the state of the estimator based on the training data.
- Some estimators allow the user to control the fitting behavior.

- For example, the `sklearn.linear_model.LinearRegression` estimator allows the user to specify whether or not to fit an intercept term.
- This is done by setting the corresponding constructor arguments of the estimator object:



```
In [3]: from sklearn.linear_model import LinearRegression
        est = LinearRegression(fit_intercept=False)
```

```
from sklearn.linear_model import LinearRegression
est = LinearRegression(fit_intercept=False)
```

- During the fitting process, the state of the estimator is stored in instance attributes that have a trailing underscore ('_').
- For example, the coefficients of a `LinearRegression` estimator are stored in the attribute `coef_`:

```
## Using Previous Code to define "est"
import numpy as np

# random training data
X = np.random.rand(10, 2)
y = np.random.randint(2, size=10)
est.fit(X, y)
est.coef_   # access coefficients

# Output : array([ 0.33176871,  0.34910639])
```

Making Prediction with Estimators

- Estimators that can generate predictions provide a `Estimator.predict` method.
- In the case of regression, Estimator.predict will return the predicted regression values; it will return the corresponding class labels in the case of classification.
- Classifiers that can predict the probability of class membership have a method `Estimator.predict_probability` that returns a two-dimensional numpy array of shape `(n_samples, n_classes)` where the classes are lexicographically ordered.

Understanding Classification

Although regression and classification appear to be very different they are in fact similar problems.

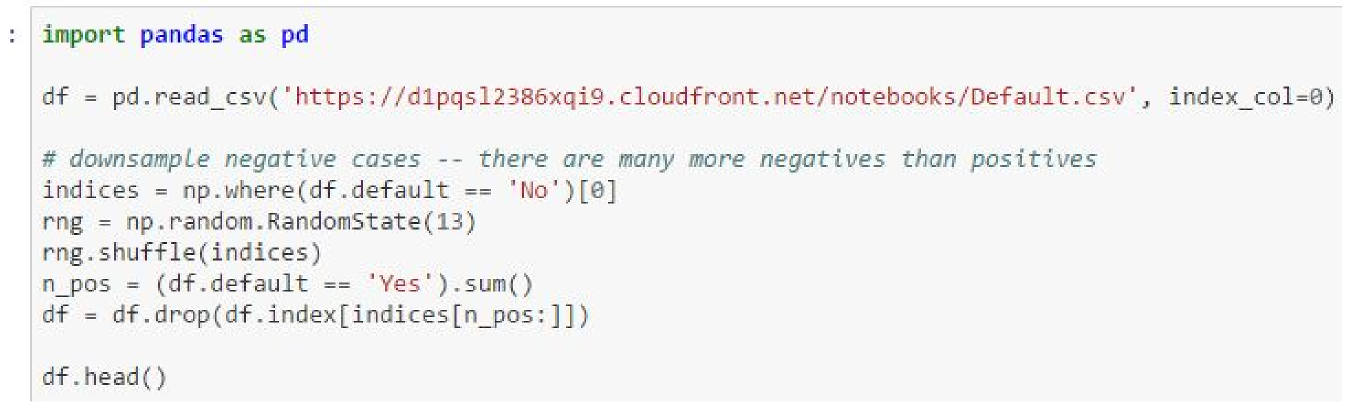
- In regression our predictions for the response are real-valued numbers
- on the other hand, in classification the response is a mutually exclusive class label
- Example "*Is the email spam?*" or "*Is the credit card transaction fraudulent?*".

Binary Classsification Problems

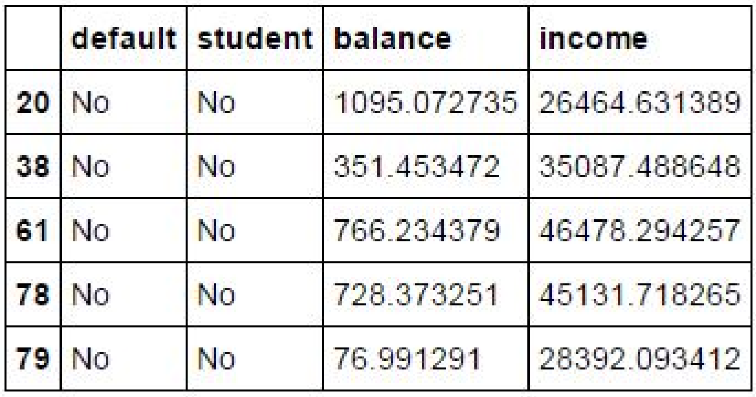
- If the number of classes is equal to two, then we call it a binary classification problem; if there are more than two classes, then we call it a multiclass classification problem.
- In the following we will assume binary classification because it's the more general case, and — we can always represent a multiclass problem as a sequence of binary classification problems.

Credit Card Fraud

- We can also think of classification as a function estimation problem where the function that we want to estimate separates the two classes.
- This is illustrated in the example below where our goal is to predict whether or not a credit card transaction is fraudulent
- *The dataset is provided by James et al.,* ***Introduction to Statistical Learning***.

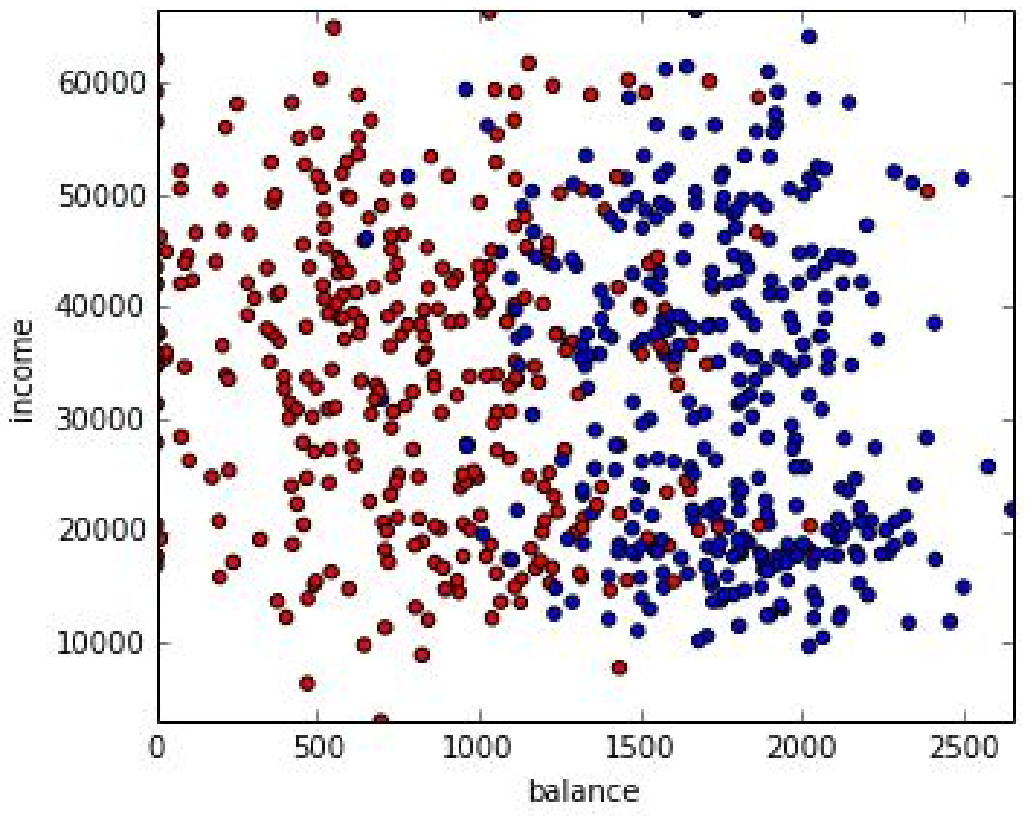


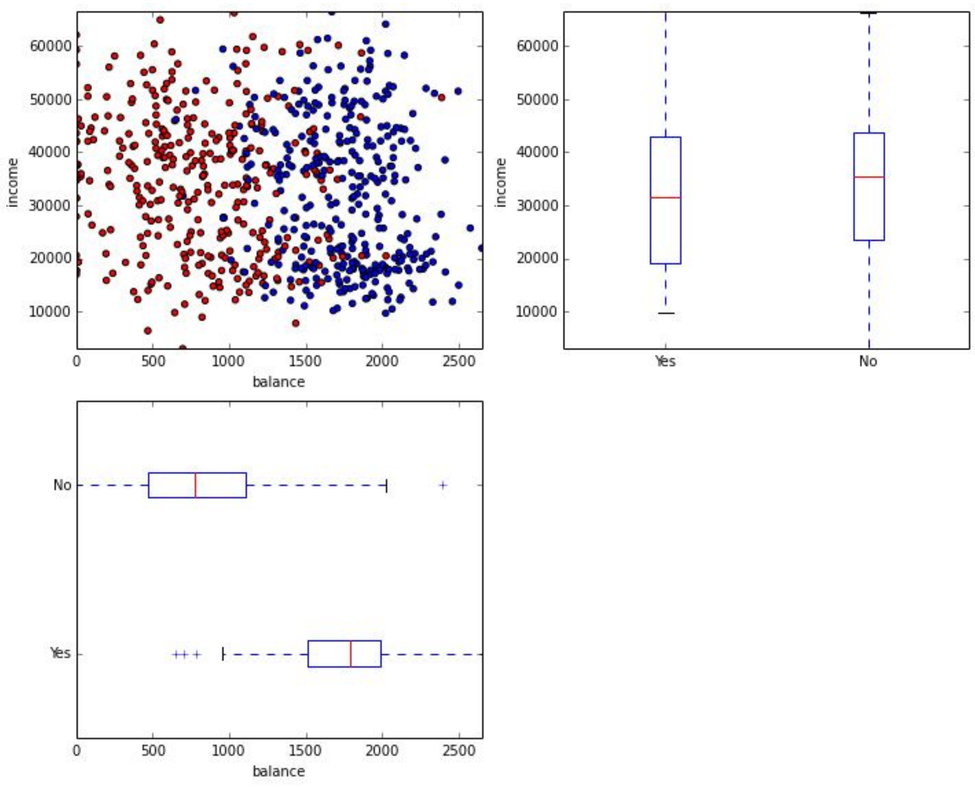
```
import pandas as pd

df = pd.read_csv('https://d1pqsl2386xqi9.cloudfront.net/notebooks/Default.csv', index_col=0)

# downsample negative cases -- there are many more negatives than positives
indices = np.where(df.default == 'No')[0]
rng = np.random.RandomState(13)
rng.shuffle(indices)
n_pos = (df.default == 'Yes').sum()
df = df.drop(df.index[indices[n_pos:]])

df.head()
```



	default	student	balance	income
20	No	No	1095.072735	26464.631389
38	No	No	351.453472	35087.488648
61	No	No	766.234379	46478.294257
78	No	No	728.373251	45131.718265
79	No	No	76.991291	28392.093412

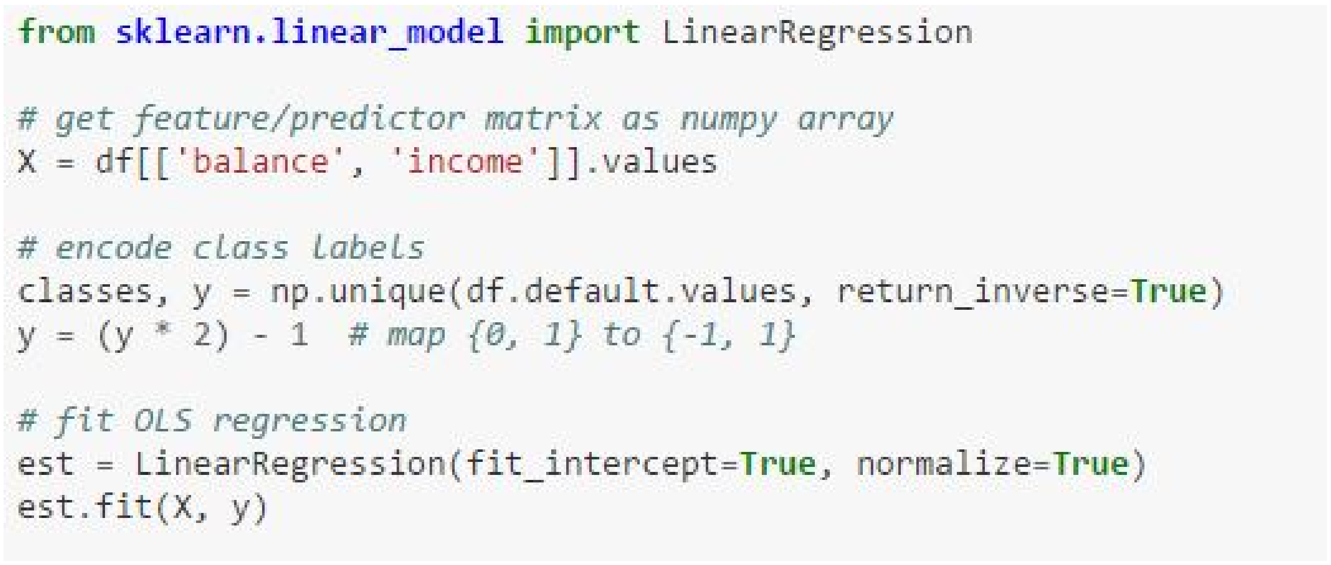
- On the left you can see a scatter plot where fraudulent cases are red dots and non-fraudulent cases are blue dots.
- A good separation seems to be a vertical line at around a balance of 1400 as indicated by the boxplots on the next slide.



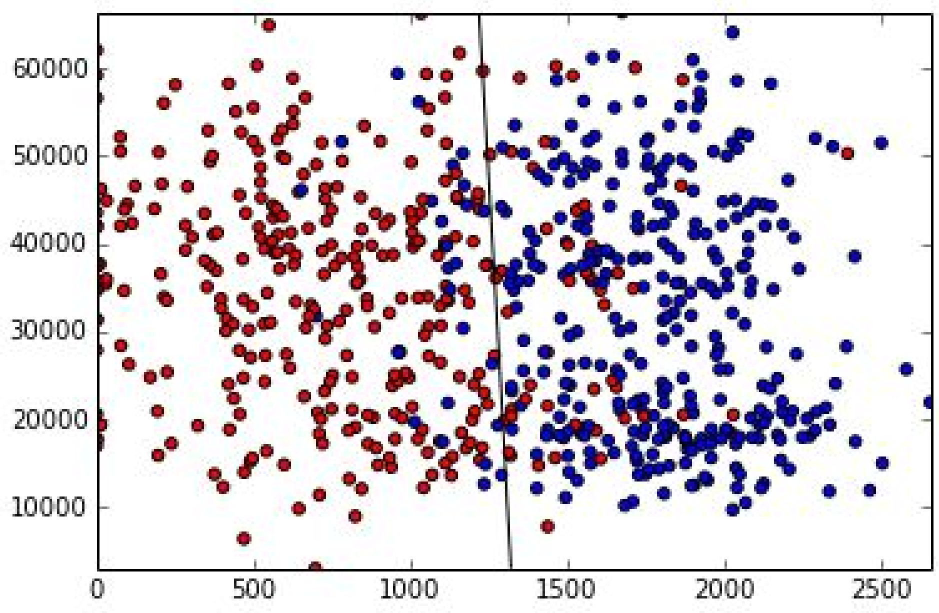

income
balance
Yes
No
income
No
Yes
balance

Simple Approach - Linear Regression

- A simple (or perhaps simplistic) approach to binary classification is to simply encode default as a numeric variable with `'Yes' == 1` and `'No' == -1`.
- Then we fit an Ordinary Least Squares regression model and use this model to predict the response as 'Yes' if the regressed value is higher than 0.0 and 'No' otherwise.
- The points for which the regression model predicts 0.0 lie on the so-called decision surface — since we are using a linear regression model, the decision surface is linear as well.



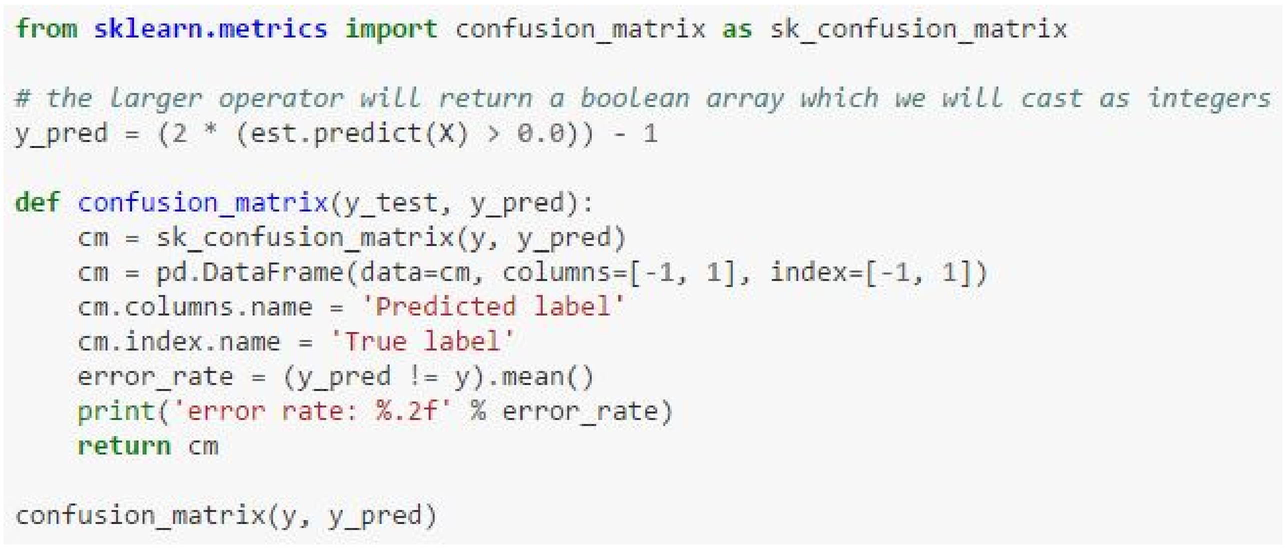
```python
from sklearn.linear_model import LinearRegression

# get feature/predictor matrix as numpy array
X = df[['balance', 'income']].values

# encode class labels
classes, y = np.unique(df.default.values, return_inverse=True)
y = (y * 2) - 1  # map {0, 1} to {-1, 1}

# fit OLS regression
est = LinearRegression(fit_intercept=True, normalize=True)
est.fit(X, y)
```



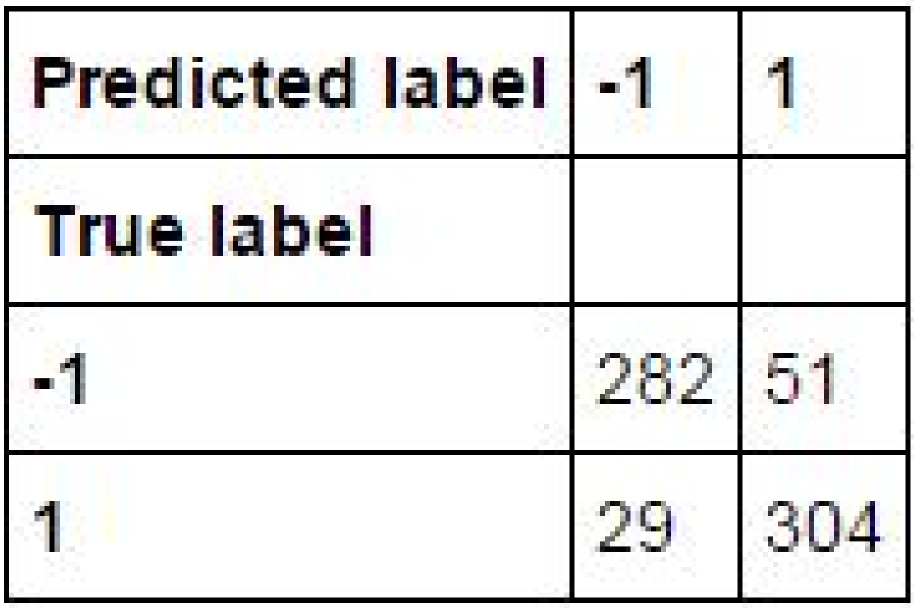
- Points that lie on the left side of the decision boundary will be classified as negative;
- Points that lie on the right side, positive.

Confusion Matrix

- We can assess the performance of the model by looking at the confusion matrix — a cross tabulation of the actual and the predicted class labels.
- The correct classifications are shown in the diagonal of the confusion matrix. The off-diagonal terms show you the **classification errors**.
- A condensed summary of the model performance is given by the **misclassification rate** determined simply by dividing the number of errors by the total number of cases.



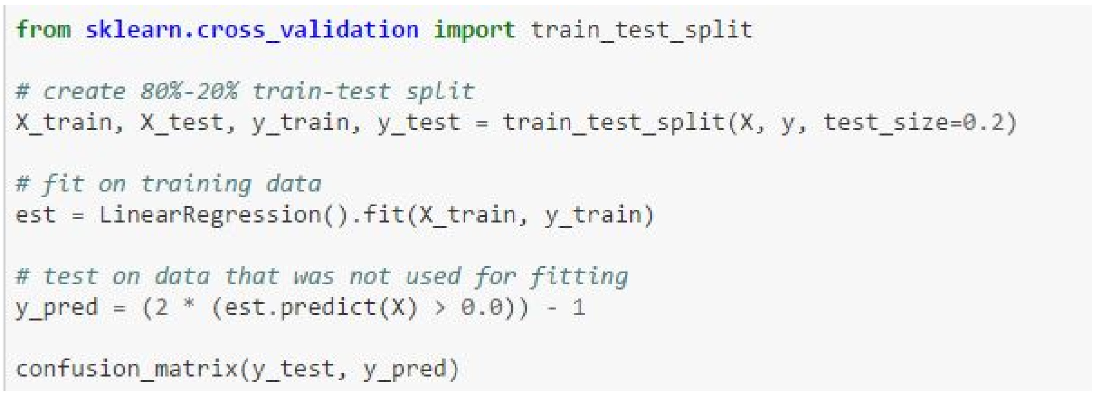
```
from sklearn.metrics import confusion_matrix as sk_confusion_matrix

# the larger operator will return a boolean array which we will cast as integers
y_pred = (2 * (est.predict(X) > 0.0)) - 1

def confusion_matrix(y_test, y_pred):
    cm = sk_confusion_matrix(y, y_pred)
    cm = pd.DataFrame(data=cm, columns=[-1, 1], index=[-1, 1])
    cm.columns.name = 'Predicted label'
    cm.index.name = 'True label'
    error_rate = (y_pred != y).mean()
    print('error rate: %.2f' % error_rate)
    return cm

confusion_matrix(y, y_pred)
```



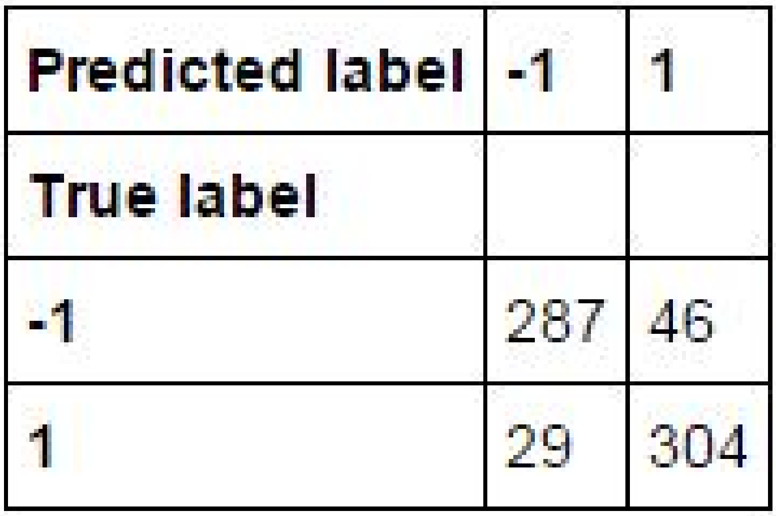
Predicted label	-1	1
True label		
-1	282	51
1	29	304

Cross Validation

- In this example we are assessing the model performance on the same data that we used to fit the model.
- This might be a biased estimate of the models performance, for a classifier that simply memorizes the training data has zero training error but would be totally useless to make predictions.
- It is much better to assess the model performance on a separate dataset called the test data.
- Scikit-learn provides a number of ways to compute such held-out estimates of the model performance.
- One way is to simply split the data into a **training set** and **testing set**.



```
from sklearn.cross_validation import train_test_split

# create 80%-20% train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# fit on training data
est = LinearRegression().fit(X_train, y_train)

# test on data that was not used for fitting
y_pred = (2 * (est.predict(X) > 0.0)) - 1

confusion_matrix(y_test, y_pred)
```



Predicted label	-1	1
True label		
-1	287	46
1	29	304

Classification Techniques

- Different classification techniques can often be compared using the type of decision surface they can learn.
- The decision surfaces describe for what values of the predictors the model changes its predictions and it can take several different shapes: piece-wise constant, linear, quadratic, vornoi tessellation, ...

This next part will introduce three popular classification techniques:

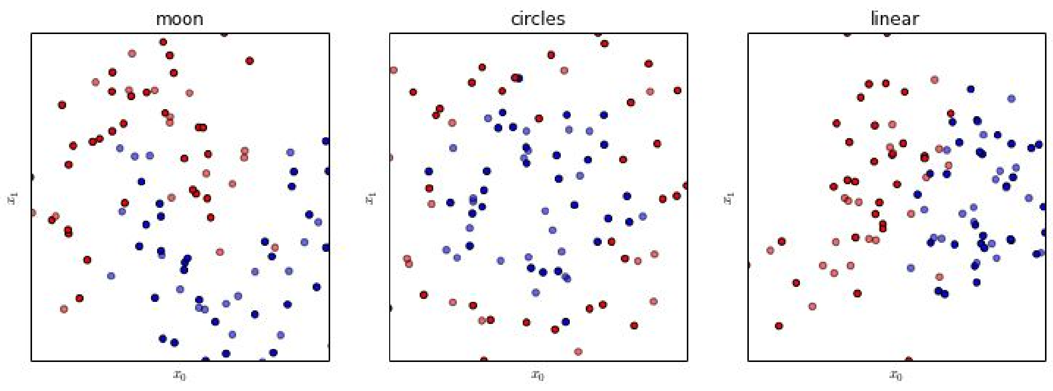
1 Logistic Regression,

2 Discriminant Analysis,

3 Nearest Neighbor.

We will investigate what their strengths and weaknesses are by looking at the decision boundaries they can model. In the following we will use three synthetic datasets that we adopted from this scikit-learn example.

Synthetic Data Sets

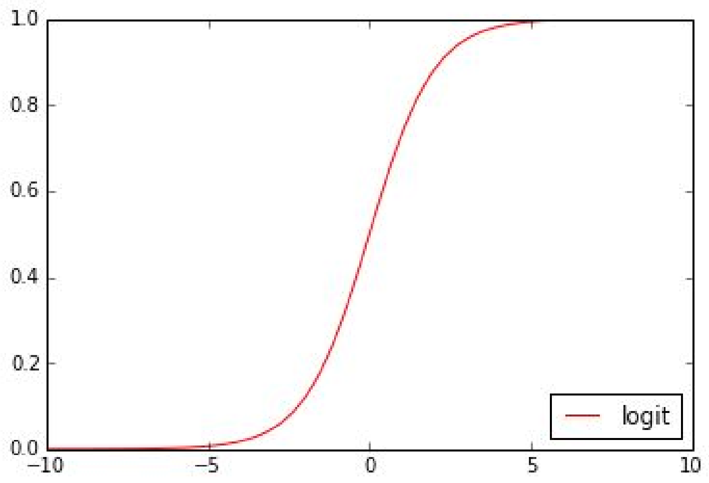


- The task in each of the above examples is to separate the red from the blue points.
- Testing data points are plotted in lighter color.
- The left example contains two intertwined moon sickles; the middle example is a circle of blues framed by a ring of reds; and the right example shows two linearly separable gaussian blobs.

Method 1: Logistic Regression

- Logistic regression can be viewed as an extension of linear regression to classification problems.
- One of the limitations of linear regression is that it cannot provide class probability estimates.
- This is often useful, for example, when we want to inspect manually the most fraudulent cases.
- Basically, we would like to constrain the predictions of the model to the range $[0, 1]$ so that we can interpret them as probability estimates.
- In Logistic Regression, we use the logit function to clamp predictions from the range $[-\infty, \infty]$ to $[0, 1]$.

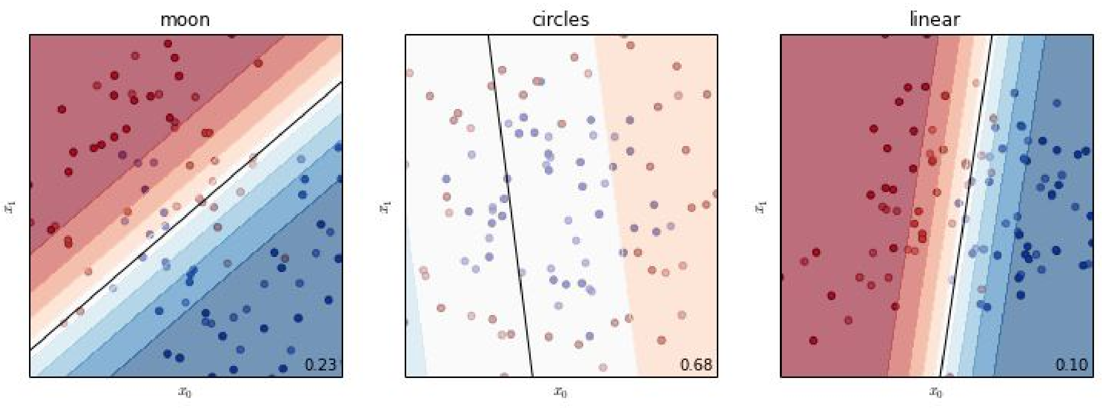
Logistic Transformation



- Logistic regression is available in scikit-learn via the class `sklearn.linear_model.LogisticRegression`.
- Lets see how Logistic Regression does on our three toy datasets.

```
from sklearn.linear_model import LogisticRegression

est = LogisticRegression()
plot_datasets(est)
```

Model Appraisal



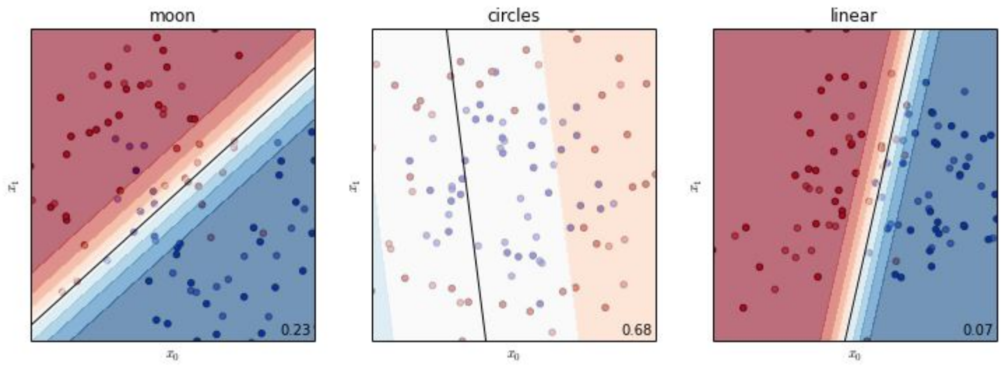
- As we can see, a linear decision boundary is not a poor approximation for the moon datasets, although we fail to separate the two tips of the sickles in the center.
- The **circles** dataset, on the other hand, is not well suited for a linear decision boundary.
- The error rate of 0.68 is in fact worse than random guessing.
- For the linear dataset we picked in fact the correct model class — the error rate of 10% is due to the noise component in our data.
- The gradient shows you the probability of class membership — white shows you that the model is very uncertain about its prediction.

Method 2: Linear Discriminant Analysis

- Linear discriminant Analysis (LDA) is another popular technique which shares some similarities with Logistic Regression.
- LDA too finds linear boundary between the two classes where points on side are classified as one class and those on the other as classified as the other class.

```
from sklearn.lda import LDA

est = LDA()
plot_datasets(est)
```

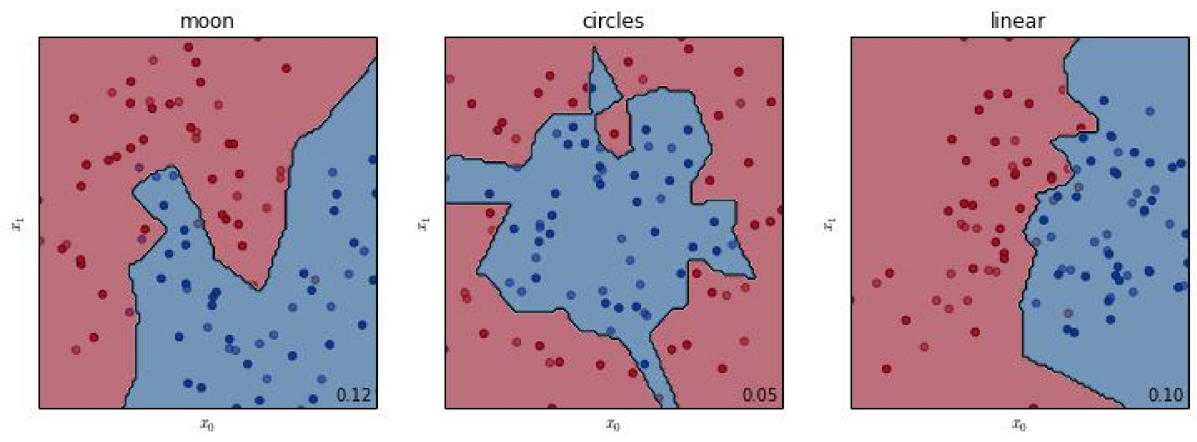
Linear Discriminant Analysis : Model Appraisal
(Remark - almost same as logistic regression)



- The major difference between LDA and Logistic Regression is the way both techniques picks the linear decision boundary.
- Linear Discriminant Analysis models the decision boundary by making distributional assumptions about the data generating process
- Logistic Regression models the probability of a sample being member of a class given its feature values.

Method 3: Nearest Neighbor

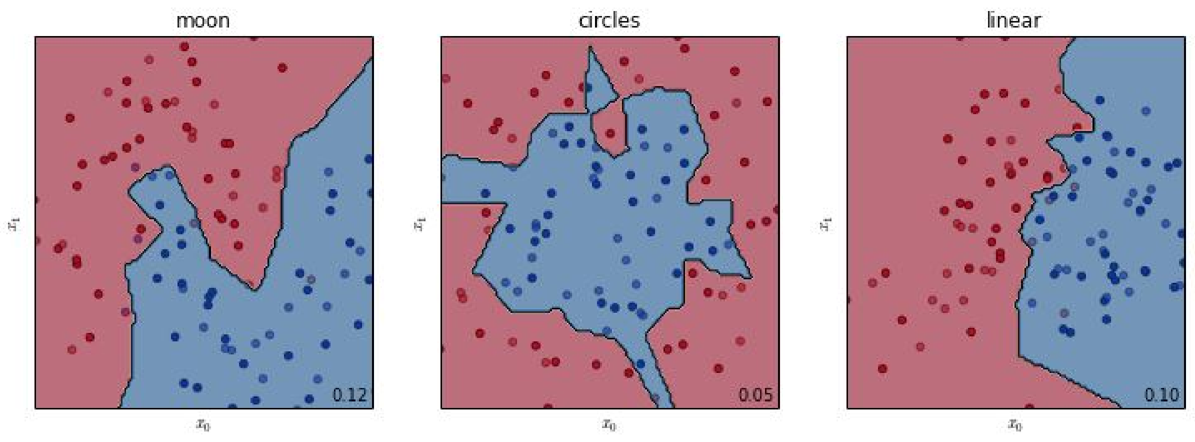
- Nearest Neighbor uses the notion of similarity to assign class labels; it is based on the smoothness assumption that points which are nearby in input space should have similar outputs.
- It does this by specifying a similarity (or distance) metric, and at prediction time it simply searches for the k most similar among the training examples to a given test example.
- The prediction is then either a majority vote of those k training examples or a vote weighted by similarity.
- The parameter k specifies the smoothness of the decision surface.
- The decision surface of a k-nearest neighbor classifier can be illustrated by the **Voronoi tesselation** of the training data, that show you the regions of constant respones.



- Yet Nearest Neighbor differs fundamentally from the above models in that it is a so-called non-parametric technique: the number of parameters of the model can grow infinitely as the size of the training data grows.
- Furthermore, it can model non-linear decision boundaries, something that is important for the first two datasets: moons and circles.

Method 3: Nearest Neighbor

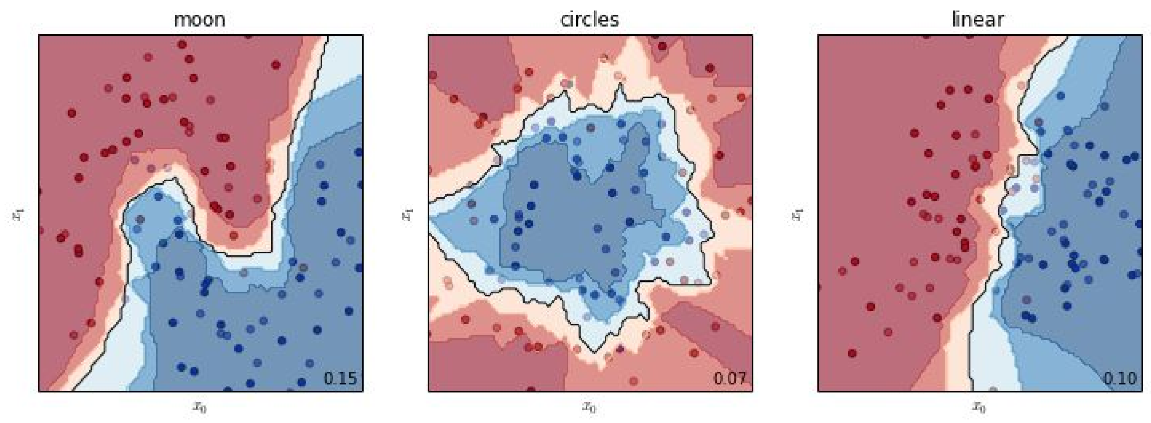
```
from sklearn.neighbors import KNeighborsClassifier

est = KNeighborsClassifier(n_neighbors=1)
plot_datasets(est)
```



Adjusting Smoothness Parameter

- If we increase k we enforce the smoothness assumption.
- This can be seen by comparing the decision boundaries in the plots below where k=5 to those above where k=1.

```
est = KNeighborsClassifier(n_neighbors=5)
plot_datasets(est)
```


moon
circles
linear
x_1
x_0