The lesson we've learned is that a machine is never efficient or trendy without a concrete possibility to use it with pragmatism. Automatic machines, built (nowadays we'd say programmed) to accomplish specific goals by transforming energy into work.

Learning:

What does learning exactly mean? Simply, we can say that learning is the ability to change according to external stimuli and remembering most of all previous experiences. Therefore, the main goal of machine learning is to study, engineer, and improve mathematical models which can be trained (once or continuously) with context-related data (provided by a generic environment), to infer the future and to make decisions without complete knowledge of all influencing elements (external factors). In other words, an agent (which is a software entity that receives information from an environment, picks the best action to reach a specific goal, and observes the results of it) adopts a statistical learning approach, trying to determine the right probability distributions and use them to compute the action (value or decision) that is most likely to be successful (with the least error).

Supervised Learning:

A supervised scenario is characterized by the concept of a teacher or supervisor, whose main task is to provide the agent with a precise measure of its error (directly comparable with output values).

Formally, the example is called regression if it's based on continuous output values. Instead, if there is only a discrete number of possible outcomes (called categories), the process becomes a classification.

For classification ,the majority of algorithms try to find the best separating hyperplane (in this case, it's a linear problem) by imposing different conditions. However, the goal is always the same: reducing the number of misclassifications and increasing the noise-robustness.

Common supervised learning applications include:

- Predictive analysis based on regression or categorical classification
- Spam detection
- Pattern detection
- Natural Language Processing
- Sentiment analysis
- Automatic image classification
- Automatic sequence processing (for example, music or speech)

UnSupervised learning:

This approach is based on the absence of any supervisor and therefore of absolute error measures; it's useful when it's necessary to learn how a set of elements can be grouped (clustered) according to their similarity (or distance measure).

There are also boundary points (such as the triangles overlapping the circle area) that need a specific criterion (normally a trade-off distance measure) to determine the corresponding cluster.

Commons unsupervised applications include:

- Object segmentation (for example, users, products, movies, songs, and so on)
- Similarity detection
- Automatic labeling

Semi-Supervised Learning:

Other important techniques involve the usage of both labeled and unlabeled data. This approach is therefore called semi-supervised and can be adopted when it's necessary to categorize a large amount of data with a few complete (labeled) examples or when there's the need to impose some constraints to a clustering algorithm (for example, assigning some elements to a specific cluster or excluding others).

Parameter learning:

A generic parametric training process must find the best parameter vector which minimizes the regression/classification error given a specific training dataset and it should also generate a predictor that can correctly generalize when unknown samples are provided.

To get model coefficients in sci-kit learn
model = LinearRegression()
model.fit(X, Y)
model.coef_
Output: array([9.10210898])

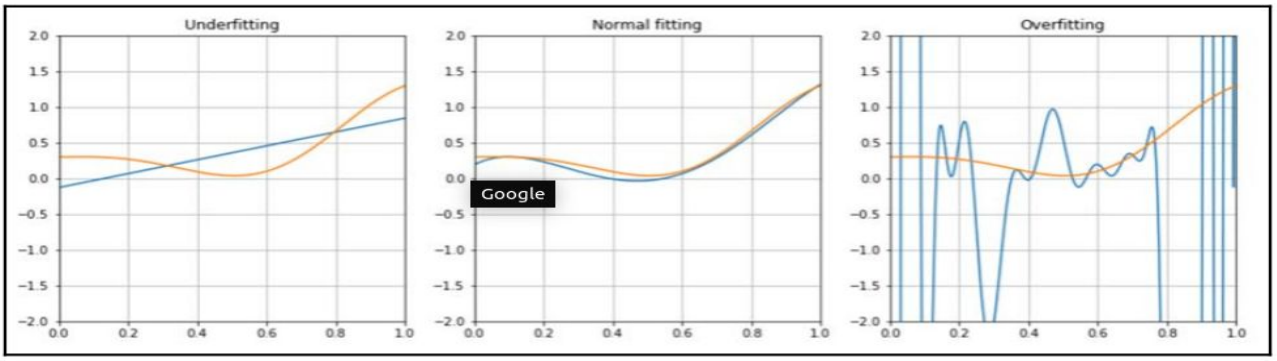
The goal of a parametric learning process is to find the best hypothesis whose corresponding prediction error is minimum and the residual generalization ability is enough to avoid overfitting.

Linear separability:

We say that the dataset X is linearly separable (without transformations) if there exists a hyperplane which divides the space into two subspaces containing only elements belonging to the same class.

Underfitting: It means that the model isn't able to capture the dynamics shown by the same training set (probably because its capacity is too limited).

Overfitting: the model has excess capacity and it's not more able to generalize considering the original dynamics provided by the training set. It can associate almost perfectly all the known samples to the corresponding output values, but when an unknown input is presented, the corresponding prediction the error can be very high.



Error measures(em):

$$Error_H = \sum_{i=1}^{n} e_m(\tilde{y}_i, y_i) \; where \; e_m \geq 0 \; \forall \tilde{y}_i, y_i$$

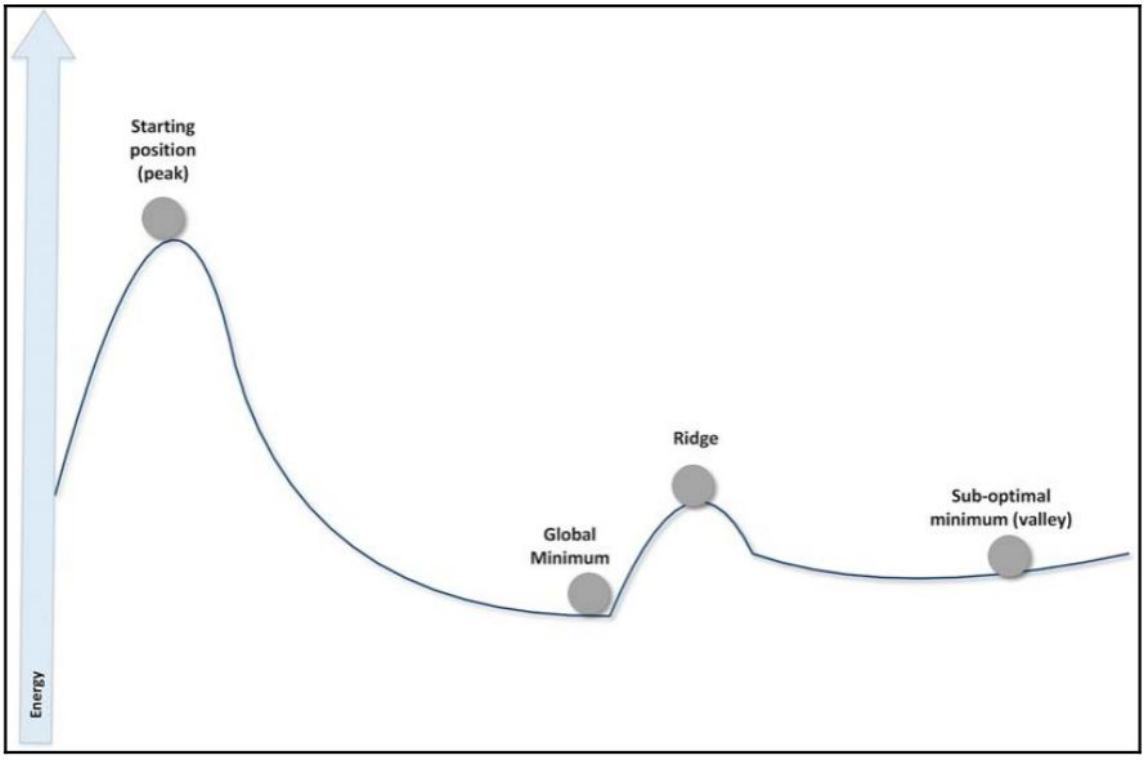
Mean Square error:

$$Error_H = \frac{1}{n}\sum_{i=1}^{n} (\tilde{y}_i - y_i)^2$$

Loss function:

Its initial value represents a starting point over the surface of an n-variables function. A generic training algorithm has to find the global minimum or a point quite close to it (there's always a tolerance to avoid an excessive number of iterations and a consequent risk of overfitting). This measure is also called loss function because its value must be minimized through an optimization problem. When it's easy to determine an element which must be maximized, the corresponding loss function will be it's reciprocal.

A helpful interpretation of a generic (and continuous) loss function can be expressed in terms of potential energy: The predictor is like a ball upon a rough surface: starting from a random point where energy (=error) is usually rather high, it must move until it reaches a stable equilibrium point where its energy (relative to the global minimum) is null.



PAC learning: probably approximately correct
A mathematical approach to determine whether a problem is learnable by a computer.

we can think about a classification problem where an algorithm A has to learn a set of concepts. In particular, a concept is a subset of input patterns X which determine the same output element. Therefore, learning a concept (parametrically) means minimizing the corresponding loss function restricted to a specific class, while learning all possible concepts (belonging to the same universe), means finding the minimum of a global loss function.

However, given a problem, we have many possible (sometimes, theoretically infinite) hypotheses and a probabilistic trade-off are often necessary. For this reason, we accept good approximations with high probability based on a limited number of input elements and produced in polynomial time.

Therefore, an algorithm A can learn the class C of all concepts (making them PAC learnable) if it's able to find a hypothesis H with a procedure O(n k) so that A, with a probability p, can classify all patterns correctly with a maximum allowed error m e . This must be valid for all statistical distributions on X and for a number of training samples which must be greater than or equal to a minimum value depending only on p and m e .

Curse of Dimensionality:
An effect that often happens in some models where training or prediction time is proportional (not always linearly) to the dimensions, so when the number of features increases, the performance of the models (that can be reasonable when the input dimensionality is small) gets dramatically reduced.

Hughes phenomenon: Without enough training data, the approximation can become problematic.

Elements of information theory:
Machine learning problem can also be analyzed in terms of information transfer or exchange. Our dataset is composed of n features, which are considered independent (for simplicity, even if it's often a realistic assumption) drawn from n different statistical distributions. Therefore, there are n probability density functions p i (x) which must be approximated through other n q i (x) functions. In any machine learning task, it's very important to understand how two corresponding distributions diverge and what is the amount of information we lose when approximating the original dataset.

The most useful measure is called **entropy**:

$$H(X) = -\sum_{x \in X} p(x) log_2 p(x)$$

For many purposes, high entropy is preferable, because it means that a certain feature contains more information.
The entropy is proportional to the variance, which is a measure of the amount of information carried by a single feature.

If we have a target probability distribution p(x), which is approximated by another distribution q(x), a useful measure is cross-entropy between p and q

$$H(P, Q) = -\sum_{x \in X} p(x) log_2 q(x)$$

Mutual information:
It is the amount of information shared by both variables and therefore, the reduction of uncertainty about X provided by the knowledge of Y:

$$I(X;Y) = H(X) - H(X|Y)$$

Intuitively, when X and Y are independent, they don't share any information. However, in machine learning tasks, there's a very tight dependence between an original feature and its prediction, so we want to maximize the information shared by both distributions. If the conditional entropy is small enough (so Y is able to describe X quite well), the mutual information gets close to the marginal entropy H(X), which measures the amount of information we want to learn.

Feature selection and Feature engineering

Scikit learn datasets

```
from sklearn.datasets import load_boston
boston = load_boston()
X = boston.data
Y = boston.target
```

Creating training and test sets

There are two main rules in performing such an operation:

- Both datasets must reflect the original distribution
- The original dataset must be randomly shuffled before the split phase in order to avoid a correlation between consequent elements

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.25, random_state=1000)
Or
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
train_size=0.75, random_state=1000)
```

Managing Categorical data:

```
import numpy as np
X = np.random.uniform(0.0, 1.0, size=(10, 2))
Y = np.random.choice(('Male','Female'), size=(10))

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
yt = le.fit_transform(Y)

from sklearn.preprocessing import LabelBinarizer
```

```
lb = LabelBinarizer()
Yb = lb.fit_transform(Y)
from sklearn.preprocessing import OneHotEncoder
data = [ [0, 10], [1, 11], [1, 8], [0, 12], [0, 15] ]
oh = OneHotEncoder(categorical_features=[0])
Y_oh = oh.fit_transform(data1)
```

Managing the missing features:
Sometimes a dataset can contain missing features, so there are a few options that can be taken into account:

- Removing the whole line
- Creating sub-model to predict those features
- Using an automatic strategy to input them according to the other known values

```
from sklearn.preprocessing import Imputer
imp = Imputer(strategy='mean') | imp = Imputer(strategy='median') |
 imp = Imputer(strategy='most_frequent')
imp.fit_transform(data)
#repalces missing values with mean or median or most_frequent
```

Data scaling and normalization
It's always preferable to standardize datasets before processing them.

```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
scaled_data = ss.fit_transform(data)

from sklearn.preprocessing import RubustScaler
rb1 = RobustScaler(quantile_range=(15, 85))
scaled_data1 = rb1.fit_transform(data)
rb1 = RobustScaler(quantile_range=(25, 75))
scaled_data1 = rb1.fit_transform(data)
rb2 = RobustScaler(quantile_range=(30, 60))
scaled_data2 = rb2.fit_transform(data)


from sklearn.preprocessing import Normalizer
>>> data = np.array([1.0, 2.0])
>>> n_max = Normalizer(norm='max')
>>> n_max.fit_transform(data.reshape(1, -1))
[[ 0.5, 1. ]]
>>> n_l1 = Normalizer(norm='l1')
>>> n_l1.fit_transform(data.reshape(1, -1))
[[ 0.33333333, 0.66666667]]
>>> n_l2 = Normalizer(norm='l2')
```

```
>>> n_l2.fit_transform(data.reshape(1, -1))
[[ 0.4472136 , 0.89442719]]
```

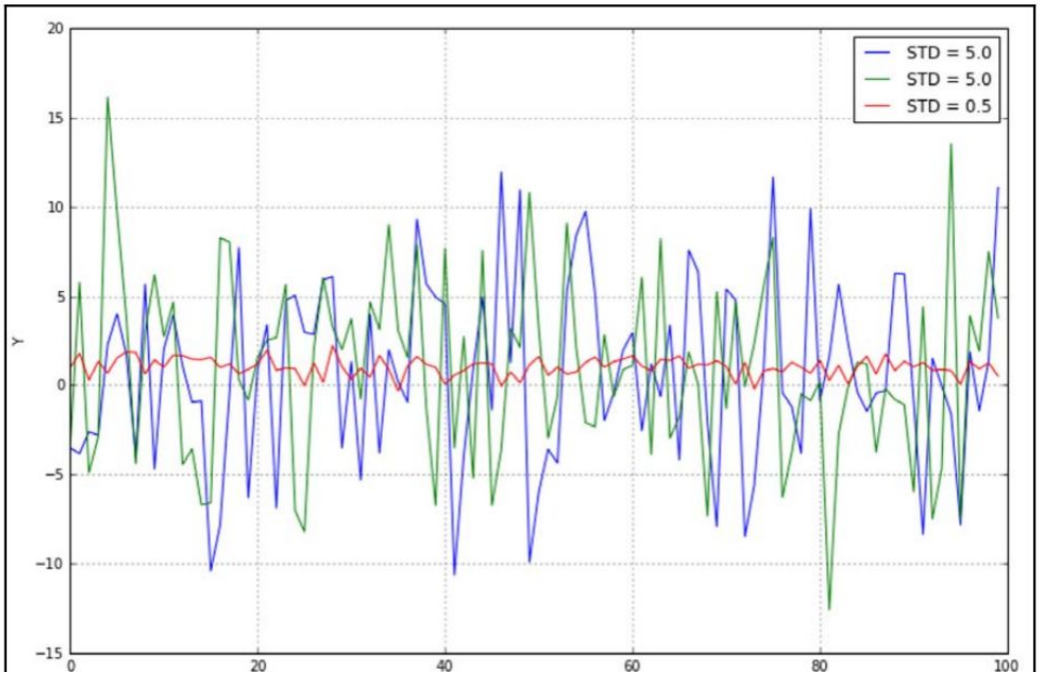
$$Max\ norm: \quad \|X\|_{max} = \frac{X}{|max_i\{X\}|}$$

$$L1\ norm: \quad \|X\|_{L1} = \frac{X}{\sum_i |x_i|}$$

$$L2\ norm: \quad \|X\|_{L2} = \frac{X}{\sqrt{\sum_i |x_i|^2}}$$

Feature Selection

Even without further analysis, it's obvious that the central line (with the lowest variance) is almost constant and doesn't provide any useful information. If you remember the previous chapter, the entropy H(X) is quite small, while the other two variables carry more information. A variance threshold is, therefore, a useful approach to remove all those elements whose contribution (in terms of variability and so, information) is under a predefined level.



```
from sklearn.feature_selection import VarianceThreshold
vt = VarianceThreshold(threshold=1.5)
X_t = vt.fit_transform(X)
```

Two examples of feature selection that use the classes **SelectKBest** (which selects the best K high-score features) and **SelectPercentile** (which selects only a subset of features belonging to a certain percentile)

```
from sklearn.datasets import load_boston, load_iris
from sklearn.feature_selection import SelectKBest, SelectPercentile,
chi2, f_regression
regr_data = load_boston()
regr_data.data.shape
(506L, 13L)
kb_regr = SelectKBest(f_regression)
X_b = kb_regr.fit_transform(regr_data.data, regr_data.target)
X_b.shape
(506L, 10L)
class_data = load_iris()
class_data.data.shape
(150L, 4L)
perc_class = SelectPercentile(chi2, percentile=15)
X_p = perc_class.fit_transform(class_data.data, class_data.target)
X_p.shape
(150L, 1L)
perc_class.scores_
array([ 10.81782088, 3.59449902, 116.16984746, 67.24482759])
```

Principal component analysis:
In general, if we consider a Euclidean space, we have:

$$X = \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\} \text{ where } \bar{x}_i \in \mathbb{R}^m \text{ and } \bar{x}_i = x_{i1}\overrightarrow{e_1} + x_{i2}\overrightarrow{e_2} + \cdots + x_{im}\overrightarrow{e_m}$$

So each point is expressed using an orthonormal basis made of m linearly independent vectors. Now, considering a dataset X, a natural question arises: is it possible to reduce m without a drastic loss of information?

In order to assess how much information is brought by each component, and the correlation among them, a useful tool is the **covariance** matrix

$$\boldsymbol{C} = \begin{pmatrix} {\sigma_1}^2 & \cdots & \sigma_{1m} \\ \vdots & \ddots & \vdots \\ \sigma_{m1} & \cdots & {\sigma_m}^2 \end{pmatrix}$$

$$where\ \sigma_{ij} = \frac{1}{m}\sum_{k}(x_{ki} - E[X_i])\,(x_{kj} - E[X_j])$$

C is symmetric and positive semidefinite, so all the eigenvalues are non-negative.

In real-life examples, there could be features which present a residual cross-correlation. In terms of information theory, it means that knowing Y gives us some information about X (which we already know), so they share information which is indeed doubled. So our goal is also to decorrelate X while trying to reduce its dimensionality. This can be achieved considering the sorted eigenvalues of C and selecting g < m values:

$$\text{Let be } \Lambda = \{\lambda_1 \geq \lambda_2 \geq \cdots \lambda_m\}\ and\ \Lambda_g \subseteq \Lambda \text{ with } \dim(\Lambda_g) \leq \dim(\Lambda)$$

$$\boldsymbol{W} = (\vec{w}_{\lambda 1}, \vec{w}_{\lambda 2}, \ldots, \vec{w}_{\lambda g})\ so\ that\ \bar{y}_R = \boldsymbol{W}\bar{y}\ where\ \bar{y}_R \in \mathbb{R}^g$$

So, it's possible to project the original feature vectors into this new (sub-)space, where each component carries a portion of total variance and where the new covariance matrix is decorrelated to reduce useless information sharing (in terms of correlation) among different features.

```
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
digits = load_digits()

pca = PCA(n_components=36, whiten=True)
X_pca = pca.fit_transform(digits.data / 255)

pca.explained_variance_ratio_

X_rebuilt = pca.inverse_transform(X_pca)
```

Non-negative matrix factorization

When the dataset is made up of non-negative elements, it's possible to use non-negative matrix factorization (NNMF).The algorithm optimizes a loss function (alternatively on W and H) based on the Frobenius norm:

$$L = \frac{1}{2}\|X - WH\|_{Frob}^2 \ where \ \|A\|_{Frob}^2 = \sqrt{\sum_i \sum_j |a_{ij}|^2}$$

```
from sklearn.datasets import load_iris
from sklearn.decomposition import NMF
iris = load_iris()
iris.data.shape
(150L, 4L)
nmf = NMF(n_components=3, init='random', l1_ratio=0.1)
Xt = nmf.fit_transform(iris.data)
nmf.reconstruct
```

SparsePCA:
Allows exploiting the natural sparsity of data while extracting principal components.

We can always use a limited number of components, but without the limitation given by a dense projection matrix. This can be achieved by using sparse matrices (or vectors), where the number of non-zero elements is quite low. In this way, each element can be rebuilt using its specific components (in most cases, they will be always the most important), which can include elements normally discarded by a dense PCA.

For PCA the equation is

$$y_R = c_1 y_{R1} + c_2 y_{R2} + \cdots + c_g y_{Rg}$$

For sparse PCA the equation is

$$y_R = (c_1 y_{R1} + c_2 y_{R2} + \cdots + c_g y_{Rg)} + (0 \cdot y_{Rg+1} + 0 \cdot y_{Rg+2} + \cdots + 0 \cdot y_{Rm})$$

```
from sklearn.decomposition import SparsePCA
spca = SparsePCA(n_components=60, alpha=0.1)
X_spca = spca.fit_transform(digits.data / 255)
spca.components_.shape
```

Kernel PCA:
Performs a PCA with non-linearly separable data sets.
It's useful to consider a projection of each sample into a particular space where the dataset becomes linearly separable. The components of this space correspond to the first, second, ... principal components and a kernel PCA algorithm.

```
from sklearn.datasets import make_circles
Xb, Yb = make_circles(n_samples=500, factor=0.1, noise=0.05)

from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=2, kernel='rbf',
fit_inverse_transform=True, gamma=1.0)
X_kpca = kpca.fit_transform(Xb)
```

Kernel PCA is a powerful instrument when we think of our dataset as made up of elements that can be a function of components (in particular, radial-basis or polynomials) but we aren't able to determine a linear relationship among them.

Linear Regression

Linear models
Consider a dataset of real-values vectors:

$$X = \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\} \; where \; \bar{x}_i \in \mathbb{R}^m$$

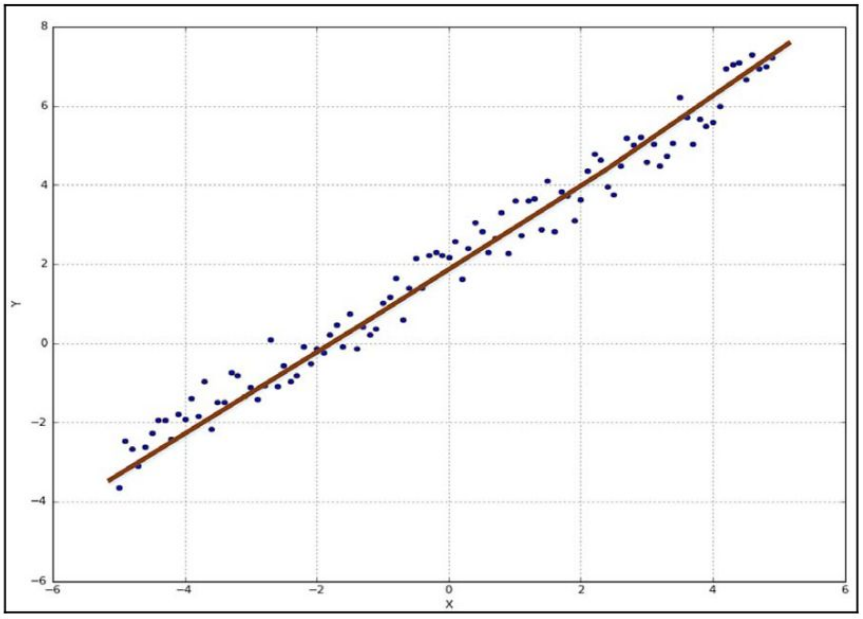
Each input vector is associated with a real value y i :

$$Y = \{y_1, y_2, \dots, y_n\} \; where \; y_n \in \mathbb{R}$$

A linear model is based on the assumption that it's possible to approximate the output values through a regression process based on the rule:

$$\tilde{y} = \alpha_0 + \sum_{i=1}^{m} \alpha_i x_i \;\; where \; \mathrm{A} = \{\alpha_0, \alpha_1, \dots, \alpha_m\}$$

Consider the following plot



As we're working on a plane, the regressor we're looking for is a function of only two parameters:

Y=a+b*X;

In order to fit our model, we must find the best parameters and to do that we choose an ordinary least squares approach. The loss function to minimize is:

$$L = \frac{1}{2}\sum_{i=1}^{n} \|\tilde{y}_i - y_i\|_2^2 \quad which\ becomes \quad L = \frac{1}{2}\sum_{i=1}^{n} (\alpha + \beta x_i - y_i)^2$$

In order to find the global minimum, we must impose:

$$\begin{cases} \dfrac{\partial L}{\partial \alpha} = \displaystyle\sum_{i=1}^{n} (\alpha + \beta x_i - y_i) = 0 \\ \dfrac{\partial L}{\partial \beta} = \displaystyle\sum_{i=1}^{n} (\alpha + \beta x_i - y_i) x_i = 0 \end{cases}$$

```
import numpy as np
def loss(v):
      e = 0.0
      for i in range(nb_samples):
            e += np.square(v[0] + v[1]*X[i] - Y[i])
```

```
    return 0.5 * e
```

And the gradient can be defined as:

```
def gradient(v):
    g = np.zeros(shape=2)
    for i in range(nb_samples):
        g[0] += (v[0] + v[1]*X[i] - Y[i])
        g[1] += ((v[0] + v[1]*X[i] - Y[i]) * X[i])
    return g

from scipy.optimize import minimize
minimize(fun=loss, x0=[0.0, 0.0], jac=gradient, method='L-BFGS-B')
```

Linear regression with scikit-learn and higher dimensionality

```
from sklearn.datasets import load_boston
boston = load_boston()
boston.data.shape
(506L, 13L)
boston.target.shape
(506L,)
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(boston.data,
boston.target, test_size=0.1)
lr = LinearRegression(normalize=True)
lr.fit(X_train, Y_train)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=True)
lr.score(X_test, Y_test)
0.77371996006718879
```

Another very important metric used in regressions is called the coefficient of determination or R squared.
It measures the amount of variance on the prediction which is explained by the dataset. We define residuals, the following quantity:

$$\forall\, i \in (0,n) \quad r_i = x_i - \tilde{x}_i$$

So the R 2 is defined as follows:

$$R^2 = 1 - \frac{\sum_i r_i^2}{\sum_i (x_i - E[X])^2}$$

R squared values close to 1 mean an almost perfect regression, while values close to 0 (or negative) imply a bad model.
cross_val_score(lr, X, Y, cv=10, scoring='r2')
0.75

Ridge regression imposes an additional shrinkage penalty to the ordinary least squares loss function to limit its squared L2 norm:

$$L(\bar{w}) = \|\boldsymbol{X}\bar{w} - \bar{y}\|_2^2 + \alpha\|\bar{w}\|_2^2$$

The additional term (through the coefficient alpha—if large it implies a stronger regularization and smaller values) forces the loss function to disallow an infinite growth of w, which can be caused by multicollinearity or ill-conditioning.

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression, Ridge
diabetes = load_diabetes()
lr = LinearRegression(normalize=True)
rg = Ridge(0.001, normalize=True)
lr_scores = cross_val_score(lr, diabetes.data, diabetes.target,
cv=10)
lr_scores.mean()
0.46196236195833718
rg_scores = cross_val_score(rg, diabetes.data, diabetes.target,
cv=10)
rg_scores.mean()
0.46227174692391299
```

For finding the right value of alpha we have **RidgeCV** which will perform **automatic grid search**

```
from sklearn.linear_model import RidgeCV
rg = RidgeCV(alphas=(1.0, 0.1, 0.01, 0.005, 0.0025, 0.001, 0.00025),
normalize=True)
rg.fit(diabetes.data, diabetes.target)
rg.alpha_
0.0050000000000000001
```

A **Lasso regression** imposes a penalty on the L1 norm of w to determine a potentially higher number of null coefficients:

$$L(\bar{w}) = \frac{1}{2n}\|X\bar{w} - \bar{y}\|_2^2 + \alpha\|\bar{w}\|_1$$

```
from sklearn.linear_model import Lasso
ls = Lasso(alpha=0.001, normalize=True)
ls_scores = cross_val_score(ls, diabetes.data, diabetes.target,
cv=10)
ls_scores.mean()
0.46215747851504058
```

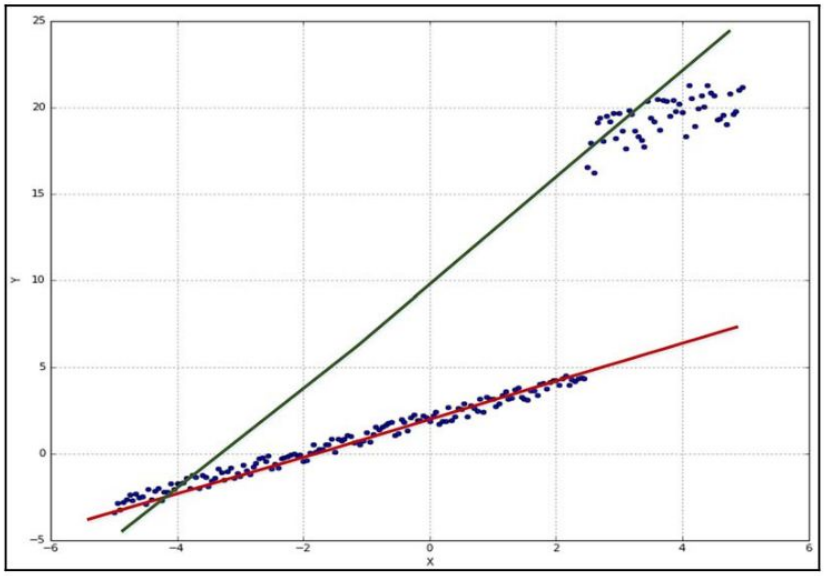
ElasticNet, which combines both Lasso and Ridge into a single model with two penalty factors: one proportional to L1 norm and the other to L2 norm. In this way, the resulting model will be sparse like a pure Lasso, but with the same regularizationability as provided by Ridge. The resulting loss function is:

$$L(\bar{w}) = \frac{1}{2n}\|X\bar{w} - \bar{y}\|_2^2 + \alpha\beta\|\bar{w}\|_1 + \frac{\alpha(1-\beta)}{2}\|\bar{w}\|_2^2$$

```
from sklearn.linear_model import ElasticNet, ElasticNetCV
>>> en = ElasticNet(alpha=0.001, l1_ratio=0.8, normalize=True)
en_scores = cross_val_score(en, diabetes.data, diabetes.target,
cv=10)
>>> en_scores.mean()
0.46358858847836454
>>> encv = ElasticNetCV(alphas=(0.1, 0.01, 0.005, 0.0025, 0.001),
l1_ratio=(0.1, 0.25, 0.5, 0.75, 0.8), normalize=True)
>>> encv.fit(dia.data, dia.target)
ElasticNetCV(alphas=(0.1, 0.01, 0.005, 0.0025, 0.001), copy_X=True,
cv=None,
eps=0.001, fit_intercept=True, l1_ratio=(0.1, 0.25, 0.5, 0.75, 0.8),
max_iter=1000, n_alphas=100, n_jobs=1, normalize=True,
positive=False, precompute='auto', random_state=None,
selection='cyclic', tol=0.0001, verbose=0)
>>> encv.alpha_
0.001
>>> encv.l1_ratio_
0.75
```

Robust regression with random sample consensus

A common problem with linear regressions is caused by the presence of outliers. An ordinary least square approach will take them into account and the result (in terms of coefficients) will be therefore biased



An interesting approach to avoid this problem is offered by random sample consensus (RANSAC), which works with every regressor by subsequent iterations, after splitting the dataset into inliers and outliers. The model is trained only with valid samples (evaluated internally or through the callable is_data_valid()) and all samples are re-evaluated to verify if they're still inliers or they have become outliers. The process ends after a fixed number of iterations or when the desired score is achieved.

```
from sklearn.linear_model import RANSACRegressor
>>> rs = RANSACRegressor(lr)
>>> rs.fit(X.reshape((-1, 1)), Y.reshape((-1, 1)))
>>> rs.estimator_.intercept_
array([ 2.03602026])
>>> es.estimator_.coef_
array([[ 0.99545348]])
```

Polynomial regression

The equation is

$$\bar{y} = \alpha_0 + \sum_{i=1}^{m} \alpha_i x_i + \sum_{j=m+1}^{k} \alpha_j f_{Pj}(x_1, x_2, \ldots, x_m) \text{ where } f_{Pj} \text{ is a polynomial function}$$

$$\bar{x} = (x_1, x_2) \Rightarrow \bar{x}_t = (x_1, x_2, {x_1}^2, {x_2}^2, x_1 x_2)$$

```
from sklearn.preprocessing import PolynomialFeatures
>>> pf = PolynomialFeatures(degree=2)
>>> Xp = pf.fit_transform(X.reshape(-1, 1))
>>> Xp.shape
(100L, 3L)
>>> lr = LinearRegression(normalize=True)
>>> lr.fit(Xp, Y.reshape((-1, 1)))
>>> lr.score(Xp, Y.reshape((-1, 1)))
0.99692778265941961
```

Isotonic regression

Here are situations when we need to find a regressor for a dataset of non-decreasing points which can present low-level oscillations (such as noise)

For these situations, scikit-learn offers the class **Isotonic Regression** , which produces a piecewise interpolating function minimizing the functional:

$$L = \sum_i w_i (y_i - \bar{y}_i)^2 \text{ where } y_0 \leq y_1 \leq \cdots \leq y_n$$

The class Isotonic Regression needs to know y min and y max (which correspond to the variables y 0 and y n in the loss function)

```
from sklearn.isotonic import IsotonicRegression
>>> ir = IsotonicRegression(-6, 10)
>>> Yi = ir.fit_transform(X, Y)
```

Logistic Regression

philosophical principle: **Occam's razor** it states that the first choice must always be the simplest and only if it doesn't fit, it's necessary to move on to more complex models

In linear classification problems, our goal is to find an optimal hyperplane, which separates the two classes. Even if called regression(**Logistic regression**), this is a classification method which is based on the probability for a sample to belong to a class.

$$X = \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\} \; where \; \bar{x}_i \in \mathbb{R}^m$$

$$Y = \{y_1, y_2, \dots, y_n\} \; where \; y_n \in \{0, 1\}$$

$$W = \{w_1, w_2, \dots, w_m\} \; where \; w_i \in \mathbb{R}$$

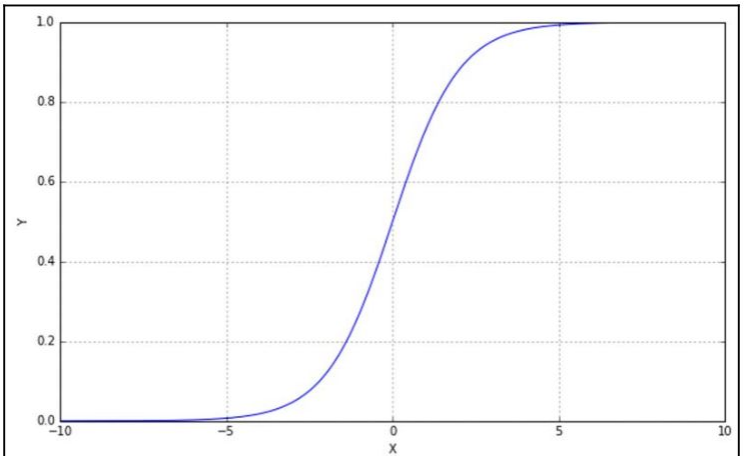
$$\forall \bar{x} \in \mathbb{R}^m \; z = \bar{x} \cdot \bar{w} = \sum_i x_i w_i$$

$$sign(z) = \begin{cases} +1 \; if \; x \in Class \; 1 \\ -1 \; if \; x \in Class \; 2 \end{cases}$$

The name **logistic** comes from the decision to use the **sigmoid** (or logistic) function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \; which \; becomes \; \sigma(\bar{x}; \bar{w}) = \frac{1}{1 + e^{-\bar{x} \cdot \bar{w}}}$$

Graphically it looks like this



$$P(y|\bar{x}) = \sigma(\bar{x}; \bar{w})$$

At this point, finding the optimal parameters is equivalent to maximizing the log-likelihood given the output class:

$$L(\bar{w}; y) = \log P(y|\bar{w}) = \sum_i \log P(y_i|\bar{x}_i, \bar{w})$$

Therefore, the optimization problem can be expressed, using the indicator notation, as the minimization of the loss function:

$$J(\bar{w}) = -\sum_i \log P(y_i|\bar{x}_i, \bar{w}) = -\sum_i (y_i \log \sigma(z_i) + (1 - y_i) \log(1 - \sigma(z_i)))$$

In terms of information theory, it means minimizing the cross-entropy between a target distribution and an approximated one:

$$H(X) = -\sum_{x \in X} p(x) log_2 q(x)$$

```
from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression()
>>> lr.fit(X_train, Y_train)
LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True, intercept_scaling=1, max_iter=100,
multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
>>> lr.score(X_test, Y_test)
0.95199999999999996

from sklearn.model_selection import cross_val_score
>>> cross_val_score(lr, X, Y, scoring='accuracy', cv=10)
lr.intercept_
array([-0.64154943])
>>> lr.coef_
array([[ 0.34417875,3.89362924]])
```

It's possible to impose norm conditions on the weights. In particular, the actual functional becomes:

$$\tilde{J}(\bar{w}) = \begin{cases} J(\bar{w}) + \alpha\|\bar{w}\|_1 \\ J(\bar{w}) + \alpha\|\bar{w}\|_2 \end{cases}$$

Stochastic gradient descent
The idea behind stochastic gradient descent is iterating a weight update based on the gradient of loss function:

$$\bar{w}(k+1) = \bar{w}(k) - \gamma \nabla L(\bar{w})$$

SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing.
The advantages of Stochastic Gradient Descent are:

- Efficiency.
- Ease of implementation (lots of opportunities for code tuning).

The disadvantages of Stochastic Gradient Descent include:

- SGD requires a number of hyperparameters such as the regularization parameter and the number of iterations.
- SGD is sensitive to feature scaling.

```
from sklearn.linear_model import SGDClassifier
X = [[0., 0.], [1., 1.]]
y = [0, 1]
clf = SGDClassifier(loss="hinge", penalty="l2", max_iter=5)
clf.fit(X, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=5,
n_iter=None, n_iter_no_change=5, n_jobs=None, penalty='l2', power_t=0.5,
random_state=None, shuffle=True, tol=None, validation_fraction=0.1,
verbose=0, warm_start=False)
>>> clf.predict([[2., 2.]])
```

SGD classifier supports the following loss functions:

- loss="hinge": (soft-margin) linear Support Vector Machine,
- loss="modified_huber": smoothed hinge loss,
- loss="log": logistic regression,

- loss='perceptron' logistic regression

Classification metrics
Jaccard similarity coefficient

$$A = \{ y_i \text{ where } y_i \text{ is a true label} \}$$

$$B = \{ \tilde{y}_i \text{ where } \tilde{y}_i \text{ is predicted label} \}$$

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

```
from sklearn.metrics import jaccard_similarity_score
>>> jaccard_similarity_score(Y_test, lr.predict(X_test))
```

- True positive: A positive sample correctly classified
- False positive: A negative sample classified as positive
- True negative: A negative sample correctly classified
- False negative: A positive sample classified as negative

```
from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_true=Y_test, y_pred=lr.predict(X_test))
```

Precision:

$$Precision = \frac{True\ positives}{True\ positives + False\ positives}$$

```
from sklearn.metrics import precision_score
>>> precision_score(Y_test, lr.predict(X_test))
```

Recall :

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives}$$

```
from sklearn.metrics import recall_score
>>> recall_score(Y_test, lr.predict(X_test))
```

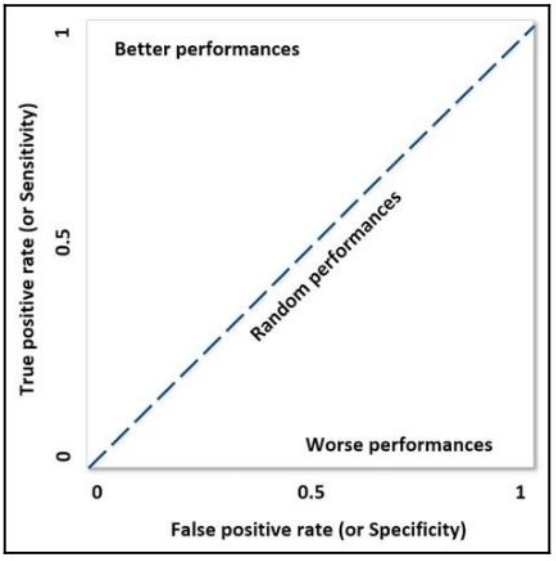
F Beta

$$F_{Beta} = (\beta^2 + 1)\frac{Precision \cdot Recall}{(\beta^2 Precision) + Recall}$$

```
from sklearn.metrics import fbeta_score
>>> fbeta_score(Y_test, lr.predict(X_test), beta=1)
0.93457943925233655
>>> fbeta_score(Y_test, lr.predict(X_test), beta=0.75)
0.94197437829691033
>>> fbeta_score(Y_test, lr.predict(X_test), beta=1.25)
0.92886270956048933
```

ROC curve

The ROC curve (or receiver operating characteristics) is a valuable tool to compare different classifiers that can assign a score to their predictions.



The x axis represents the increasing false positive rate (also known as specificity), while the y axis represents the true positive rate (also known as sensitivity). The dashed oblique line represents a perfectly random classifier, so all the curves below this threshold perform worse than a random choice, while the ones above it show better performances.

```
>>> Y_scores = lr.decision_function(X_test)
from sklearn.metrics import roc_curve
>>> fpr, tpr, thresholds = roc_curve(Y_test, Y_scores)
```

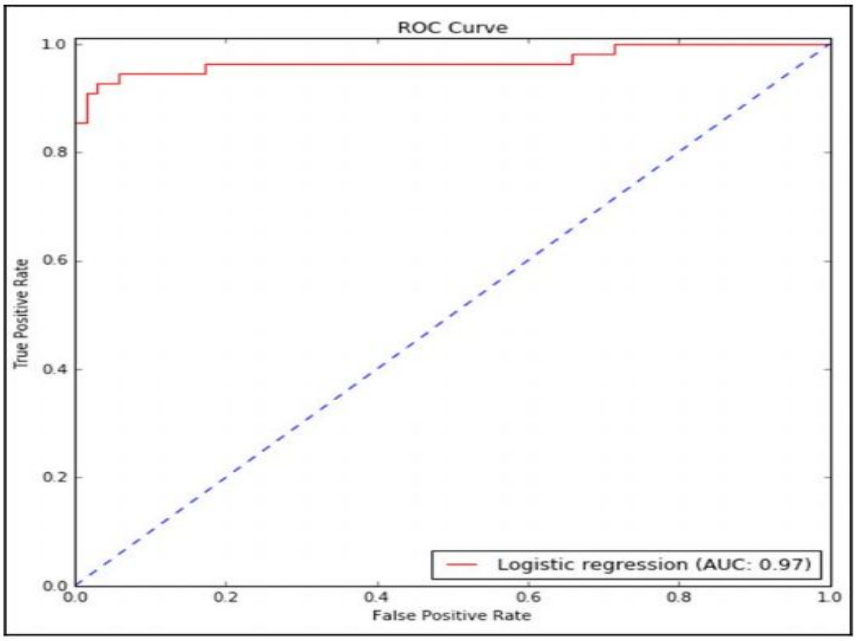
Before proceeding, it's also useful to compute the area under the curve (AUC), whose value is bounded between 0 (worst performances) and 1 (best performances), with a perfectly random value corresponding to 0.5:

```
from sklearn.metrics import auc
```

```
>>> auc(fpr, tpr)
0.96961038961038959
```

We already know that our performances are rather good because the AUC is close to 1.
Plotting with matplotlib

```
import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 8))
>>> plt.plot(fpr, tpr, color='red', label='Logistic regression (AUC:
%.2f)'
% auc(fpr, tpr))
>>> plt.plot([0, 1], [0, 1], color='blue', linestyle='--')
>>> plt.xlim([0.0, 1.0])
>>> plt.ylim([0.0, 1.01])
>>> plt.title('ROC Curve')
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.legend(loc="lower right")
>>> plt.show()
```



Naive Bayes

Naive Bayes are a family of powerful and easy-to-train classifiers that determine the probability of an outcome given a set of conditions using Bayes' theorem

Bayes' theorem

Let's consider two probabilistic events A and B. We can correlate the marginal probabilities

P(A) and P(B) with the conditional probabilities P(A|B) and P(B|A) using the product rule:

$$\begin{cases} P(A \cap B) = P(A|B)P(B) \\ P(B \cap A) = P(B|A)P(A) \end{cases}$$

From the above 2 equations

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

First of all, let's consider the marginal probability P(A); this is normally a value that determines how probable a target event is, such as P(Spam) or P(Rain). As there are no other elements, this kind of probability is called priori, because it's often determined by mathematical considerations or simply by a frequency count.

The first term is similar to P(Spam) because it's the probability of spam given a certain condition. For this reason, it's called a posteriori (in other words, it's a probability that we can estimate after knowing some additional elements).

$$P_{A-Posteriori} \propto Likelihood \cdot P_{A-Priori}$$

The formula becomes

$$P(A|B) = \alpha P(B|A)P(A)$$

The last step is considering the case when there are more concurrent conditions

$$P(A|C_1 \cap C_2 \cap \dots \cap C_n)$$

A common assumption is called conditional independence

$$P(A|C_1 \cap C_2 \cap \dots \cap C_n) = \alpha P(C_1|A)P(C_2|A) \dots P(C_n|A)P(A)$$

Naive Bayes classifiers

A naive Bayes classifier is called so because it's based on a naive condition, which implies the conditional independence of causes

Given the dataset

$$X = \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\} \; where \; \bar{x}_i \in \mathbb{R}^m$$

Every feature vector is represented as

$$\bar{x}_i = [x_1, x_2, \dots, x_m]$$

Considering Bayes' theorem under conditional independence, we can write:

$$P(y|x_1, x_2, \ldots, x_m) = \alpha P(y) \prod_i P(x_i|y)$$

The values of the marginal Apriori probability P(y) and of the conditional probabilities P(x i |y) is obtained through a frequency count; therefore, given an input vector x, the predicted class is the one for which the a posteriori probability is maximum.

scikit-learn implements three naive Bayes variants based on the same number of different probabilistic distributions: **Bernoulli**, **multinomial**, and **Gaussian**. The first one is a **binary distribution**, useful when a feature can be present or absent. The second one is a **discrete distribution** and is used whenever a feature must be represented by a whole number (for example, in natural language processing, it can be the frequency of a term), while the third is a **continuous distribution** characterized by its mean and variance.

Bernoulli naive Bayes

If X is random variable and is Bernoulli-distributed, it can assume only two values (for simplicity, let's call them 0 and 1) and their probability is

$$P(X) = \begin{cases} p & if\ X = 1 \\ q & if\ X = 0 \end{cases}$$

$$where\ q = 1 - p\ and\ 0 < p < 1$$

```
from sklearn.naive_bayes import BernoulliNB
from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.25)
>>> bnb = BernoulliNB(binarize=0.0)
>>> bnb.fit(X_train, Y_train)
>>> bnb.score(X_test, Y_test)
0.85333333333333339
```

Multinomial naive Bayes

A multinomial distribution is useful to model feature vectors where each value represents, for example, the number of occurrences of a term or its relative frequency.

$$P(X_1 = x_1 \cap X_2 = x_2 \cap \ldots \cap X_k = x_k) = \frac{n!}{\prod_i x_i!} \prod_i {p_i}^{x_i}$$

```
from sklearn.naive_bayes import MultinomialNB
>>> mnb = MultinomialNB()
>>> mnb.fit(X, Y)
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
>>> mnb.predict(dv.fit_transform(test_data))
```

Gaussian naive Bayes

Gaussian naive Bayes is useful when working with continuous values whose probabilities can be modeled using a Gaussian distribution:

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The conditional probabilities P(x i |y) are also Gaussian distributed;

$$L(\mu; \sigma^2; x_i|y) = log \prod_k P(x_i^{(k)}|y) = \sum_k log\ P(x_i^{(k)}|y)$$

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
>>> gnb.fit(X_train, Y_train)
>>> Y_gnb_score = gnb.predict_proba(X_test)
```

Support Vector Machines

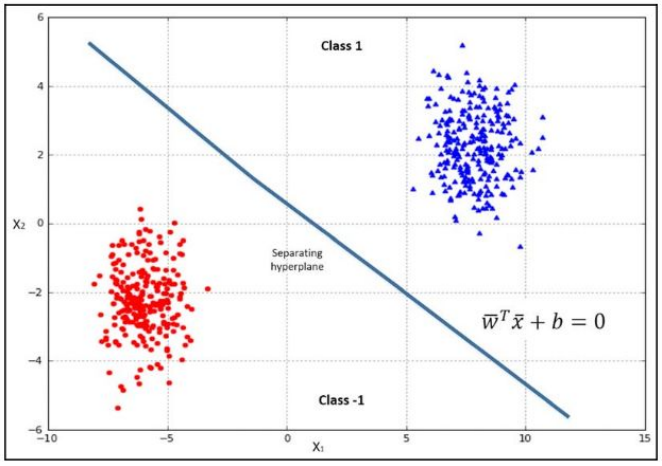
Linear support vector machines

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}\ where\ \bar{x}_i \in \mathbb{R}^m$$

$$Y = \{y_1, y_2, \ldots, y_n\}\ where\ y_n \in \{-1, 1\}$$

Our goal is to find the best separating hyperplane, for which the equation is:

$$\bar{w}^T\bar{x} + b = 0 \quad where\ \bar{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix}\ and\ \bar{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}$$

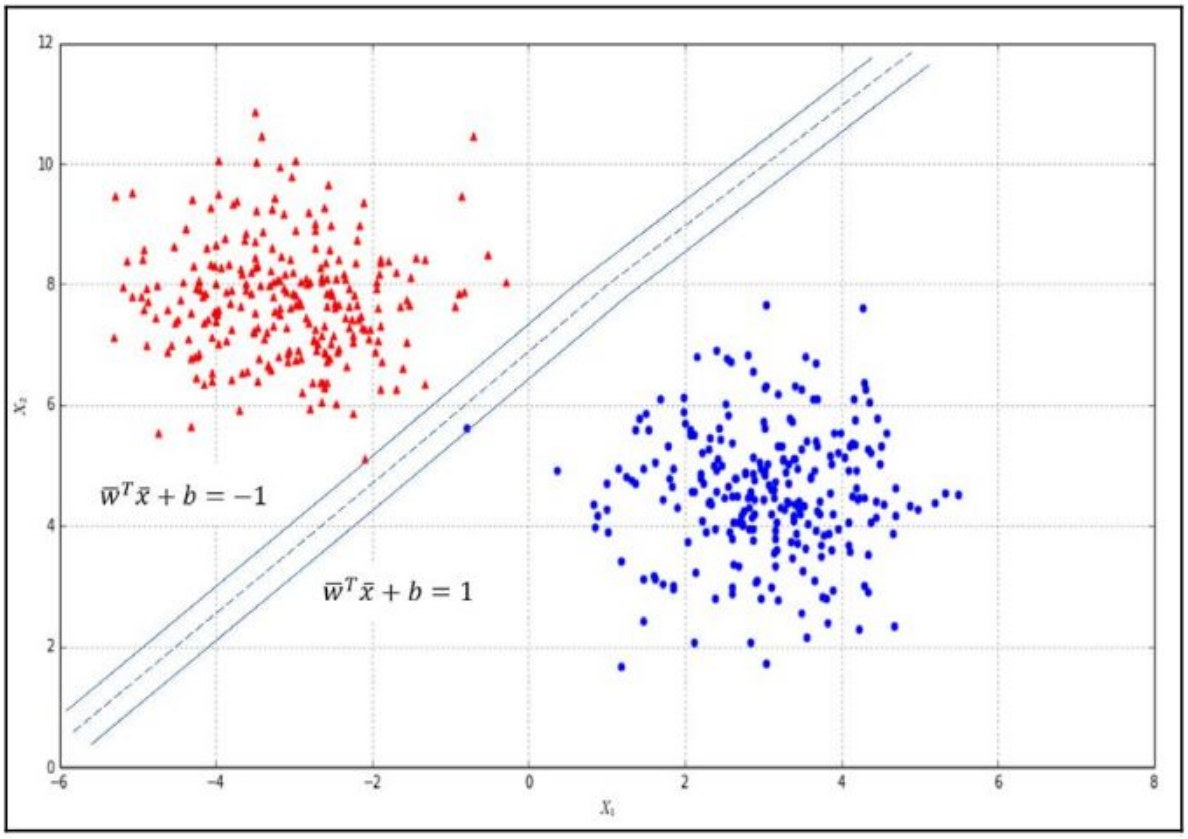


our classifier can be written as:

$$\hat{y} = f(\bar{x}) = sgn(\bar{w}^T \bar{x} + b)$$

In a realistic scenario, the two classes are normally separated by a margin with two boundaries where a few elements lie. Those elements are called support vectors. For a more
generic mathematical expression, it's preferable to renormalize our dataset so that the support vectors will lie on two hyperplanes with equations:

$$\begin{cases} \bar{w}^T \bar{x} + b = -1 \\ \bar{w}^T \bar{x} + b = \quad 1 \end{cases}$$



Our goal is to maximize the distance between these two boundary hyperplanes so as to reduce the probability of misclassification

The function to minimize in order to train a support vector machine:

$$\begin{cases} min \dfrac{1}{2} \|\bar{w}\| \\ y_i(\bar{w}^T \bar{x}_i + b) \geq 1 \end{cases}$$

This can be further simplified

$$\begin{cases} min \frac{1}{2} \overline{w}^T \overline{w} \\ y_i(\overline{w}^T \bar{x}_i + b) \geq 1 \end{cases}$$

(CS229 Lecture notes Andrew Ng)
We see that if a point is far from the separating hyperplane, then we may be significantly more confident in our prediction3

Functional and geometric margins

Let's formalize the notions of the functional and geometric margins. Given a training example (x (i) , y(i)), we define the functional margin of (w, b) with respect to the training example

γˆ (i) = y (i) (w T x + b)

large functional margin represents a confident and a correct prediction.

```
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
>>> svc = SVC(kernel='linear')
>>> cross_val_score(svc, X, Y, scoring='accuracy', cv=10).mean()
0.93191356542617032
```

Kernel-based classification

(Read in my coursera notes good clarity)

When working with non-linear problems, it's useful to transform the original vectors by projecting them into a higher dimensional space where they can be linearly separated

Our mathematical formulation becomes:

$$\begin{cases} min \frac{1}{2} \overline{w}^T \overline{w} + C \sum_i \zeta_i \\ y_i(\overline{w}^T \phi(\bar{x}_i) + b) \geq 1 - \zeta_i \end{cases}$$

Every feature vector is now filtered by a non-linear function that can completely reshape the Scenario. The final formulation is

$$\begin{cases} max \left(\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \phi(\bar{x}_i)^T \phi(\bar{x}_j) \right) \\ \sum_i \alpha_i y_i = 0 \end{cases}$$

Therefore it's necessary to compute the following for every couple of vectors:

$$\phi(\bar{x}_i)^T \phi(\bar{x}_j)$$

It's now that the so-called **kernel** trick takes place. There are particular functions (called kernels) that have the following property:

$$K(\bar{x}_i, \bar{x}_j) = \phi(\bar{x}_i)^T \phi(\bar{x}_j)$$

In other words, the value of the kernel for two feature vectors is the product of the two projected vectors.

Types of kernels
Radial Basis Function
It's function is

$$K(\bar{x}_i, \bar{x}_j) = e^{-\gamma |\bar{x}_i - \bar{x}_j|^2}$$

The gamma parameter determines the amplitude of the function.

Polynomial kernel

$$K(\bar{x}_i, \bar{x}_j) = (\gamma \bar{x}_i{}^T \cdot \bar{x}_j + r)^c$$

The exponent c is specified through the parameter degree

Sigmoid kernel

$$K(\bar{x}_i, \bar{x}_j) = \frac{1 - e^{-2(\gamma \bar{x}_i{}^T \cdot \bar{x}_j + r)}}{1 + e^{-2(\gamma \bar{x}_i{}^T \cdot \bar{x}_j + r)}}$$

As per my coursera course
Each point in our dataset is labelled as a landmark, the similarity of the input point and individual landmarks is calculated lets call it f(i) for ith landmark. Then we predict y=1 if (theta)T*f>=0 and 0 otherwise. In the cost function itself (theta)T*X is replaced with (theta)T*f. This is what I understood from kernel trick.

```
from sklearn.svm import SVR
>>> svr = SVR(kernel='poly', degree=2, C=1.5, epsilon=0.5)
>>> cross_val_score(svr, X.reshape((nb_samples*2, 1)), Y,
scoring='neg_mean_squared_error', cv=10).mean()
-1.4641683636397234
```

Support vector machines for regression

```
from sklearn.svm import SVR
>>> svr = SVR(kernel='poly', degree=2, C=1.5, epsilon=0.5)
```

```
>>> cross_val_score(svr, X.reshape((nb_samples*2, 1)), Y,
scoring='neg_mean_squared_error', cv=10).mean()
-1.4641683636397234
```

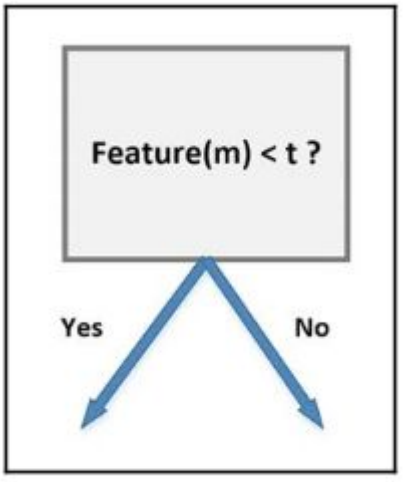
Decision Trees and Ensemble Learning

Binary decision trees

A binary decision tree is a structure based on a sequential decision process. Starting from the root, a feature is evaluated and one of the two branches is selected. This procedure is repeated until a final leaf is reached,

Binary decisions

Every vector is made up of m features, so each of them can be a good candidate to create a node based on the (feature, threshold) tuple:



Suppose we define the selection tuple as:

$$\sigma = \langle i, t_k \rangle$$

Here, the first element is the index of the feature we want to use to split our dataset at a certain node (it will be the entire dataset only at the beginning; after each step, the number of samples decreases), while the second is the threshold that determines left and right branches. The choice of the best threshold is a fundamental element because it determines the structure of the tree and, therefore, its performance. The goal is to reduce the residual impurity in the least number of splits so as to have a very short decision path between the sample data and the classification result.

We can also define a total impurity measure by considering the two branches:

$$I(D,\sigma) = \frac{N_{left}}{N_D} I(D_{left}) + \frac{N_{right}}{N_D} I(D_{right})$$

Here, D is the whole dataset at the selected node, D left and D right are the resulting subsets (by applying the selection tuple), and the I are impurity measures.

Impurity measures
According to a frequentist approach, this value is the ratio
between the number of samples belonging to class i and the total number of samples belonging to the selected node.

Gini impurity index

$$I_{Gini}(j) = \sum_i p(i|j)(1 - p(i|j))$$

Gini impurity measures the probability of a misclassification if a label is randomly chosen using the probability distribution of the branch.

Cross-entropy impurity index

$$I_{Cross-entropy}(j) = -\sum_i p(i|j)\log\, p(i|j)$$

the cross-entropy allows you to select the split that minimizes the uncertainty about the classification
we defined the concept of mutual information I(X; Y) = H(X) - H(X|Y) as the amount of information shared by both variables, thereby reducing the uncertainty about X provided by the knowledge of Y. We can use this to define the information gain provided by a split:

$$IG(\sigma) = H(Parent) - H(Parent|Children)$$

When growing a tree, we start by selecting the split that provides the highest information gain and proceed until one of the following conditions is verified:

- All nodes are pure
- The information gain is null
- The maximum depth has been reached

Feature importance using Decision trees:

Decision trees offer a different approach based on the impurity reduction determined by every single feature. In particular, considering a feature x i , its importance can be determined as:

$$Importance(x_i) = \sum_k \frac{N_k}{N} \Delta I_{x_i}$$

N k is the number of samples reaching the node k.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
>>> dt = DecisionTreeClassifier()
>>> print(cross_val_score(dt, X, Y, scoring='accuracy',
cv=10).mean())
0.970
from sklearn.tree import export_graphviz
>>> dt.fit(X, Y)
>>> with open('dt.dot', 'w') as df:
df = export_graphviz(dt, out_file=df,
feature_names=['A','B','C'],
class_names=['C1', 'C2', 'C3'])
```

A , B , and C as feature names and C1 , C2 , and C3 as class names. there are two kinds of nodes:

- Nonterminal, which contains the splitting tuple (as feature <= threshold) and a positive impurity measure
- Terminal, where the impurity measure is null and a final target class is present

Ensemble learning

This approach is based on so-called strong learners, or methods that are optimized to solve a specific problem by looking for the best possible solution. Another approach is based on a set of weak learners that can be trained in parallel or sequentially (with slight modifications on the parameters) and used as an ensemble based on a majority vote or the averaging of results. These methods can be classified into two main categories:

Bagged (or Bootstrap) trees: In this case, the ensemble is built completely. The training process is based on a random selection of the splits and the predictions are based on a majority vote. Random forests are an example of bagged tree Ensembles.

Boosted trees: The ensemble is built sequentially, focusing on the samples that have been previously misclassified. Examples of boosted trees are AdaBoost and gradient tree boosting.

Random forests

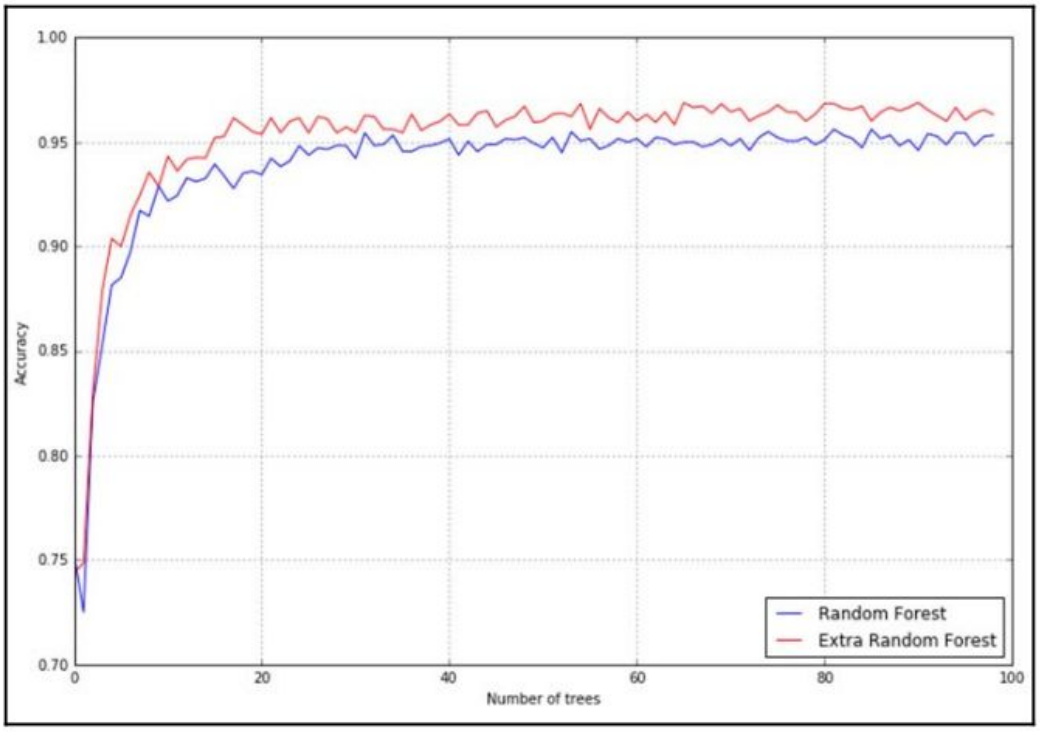
A random forest is a set of decision trees built on random samples with a different policy

for splitting a node: Instead of looking for the best choice, in such a model, a random subset of features (for each tree) is used, trying to find the threshold that best separates the data

```
from sklearn.ensemble import RandomForestClassifier
>>> nb_classifications = 100
>>> accuracy = []
>>> for i in range(1, nb_classifications):
a = cross_val_score(RandomForestClassifier(n_estimators=i),
digits.data, digits.target, scoring='accuracy', cv=10).mean()
rf_accuracy.append(a)
```

Using the **ExtraTreesClassifier** class, it's possible to implement a model that randomly computes thresholds and picks the best one.

```
from sklearn.ensemble import ExtraTreesClassifier
>>> nb_classifications = 100
>>> for i in range(1, nb_classifications):
a = cross_val_score(ExtraTreesClassifier(n_estimators=i),
digits.data, digits.target, scoring='accuracy', cv=10).mean()
et_accuracy.append(a)
```



Feature importance in random forests

$$Importance(x_i) = \frac{1}{N_{Trees}} \sum_t \sum_k \frac{N_k}{N} \Delta I_{x_i}$$

AdaBoost - Adaptive Boosting

The basic structure behind this can be a decision tree, but the dataset used for training is continuously adapted to force the model to focus on those samples that are misclassified. Moreover, the classifiers are added sequentially, so a new one boosts the previous one by improving the performance in those areas where it was not as accurate as expected.

At each iteration, a weight factor is applied to each sample so as to increase the importance of the samples that are wrongly predicted and decrease the importance of others. In other words, the model is repeatedly boosted, starting as a very weak learner until the maximum n_estimators number is reached. The predictions, in this case, are always obtained by majority vote.

```
from sklearn.ensemble import AdaBoostClassifier
>>> accuracy = []
>>> nb_classifications = 100
>>> for i in range(1, nb_classifications):
a = cross_val_score(AdaBoostClassifier(n_estimators=i,
learning_rate=0.1), digits.data, digits.target, scoring='accuracy',
cv=10).mean()
>>> ab_accuracy.append(a)
```

Unsupervised Learning:

Clustering:

Consider a dataset of points.We assume that it's possible to find a criterion (not unique) so that each sample can be associated with a specific group. Each group is called a cluster and the process of finding the function G is called clustering.