Theory

Bayes classifier

g is the *classifier.*

$$\begin{aligned} g : \mathcal{X} &\to \mathcal{Y} \\ \mathbb{R}^d &\to \{0,1\} \end{aligned}$$

To model the learning problem, we use the pair (X, Y) described by (μ, η) where μ is the probability measure:

$$\mu(A) = \mathbb{P}(X \in A)$$

And η is the regression of Y on X:

$$\eta(X) = \mathbb{P}(Y = 1|X = x) = \mathbb{E}[Y|X = x]$$

η is also called the *a posteriori probability.*
The Bayes classifier is:

$$\begin{cases} 1 & \text{if } \eta(x) > 1/2 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Or, if $\mathcal{Y}$ is $\{-1, 1\}$, we write the classifier as such: $g(x) = 2\mathbb{1}\{\eta(x) > 1/2\} - 1$.

Theorem:

For any classifier g: $\mathbb{R}^d \to \{0,1\}$,

$$\mathbb{P}(g^*(X) \neq Y) \leq \mathbb{P}(g(X) \neq Y)$$

In other words, the Bayes classifier is theorically **the best classifier**.

Proof: express $\mathbb{P}(g(X) \neq Y) - \mathbb{P}(g^*(X) \neq Y)$ in terms of dummies (use complementaries) and show that it is superior to 0.

Gradient descent

ℓ the loss function to minimize:
$\theta_{t+1} = \theta_t - \alpha \nabla \ell$

Algorithm 1 Global gradient descent

```
Loss function $\widehat{L}_n(\hat{f}_\omega(x)) = \Sigma_{i=1}^n \ell(\hat{f}_\omega(x), y)$
$E = 1000$
$\epsilon$ = small value
$\omega_0$ = intial value in $t_0$
while $E > \epsilon$ do
    $\omega_{t+1} = \omega_t - \epsilon \Sigma_{i=1}^n \nabla_\omega \ell(\hat{f}_\omega(x), y)$
    Compute $E = L_n(\omega_{t+1})$
end while
```

Algorithm 2 Stochastic gradient descent

```
Loss function $\widehat{L}_n(\hat{f}_\omega(x)) = \Sigma_{i=1}^n \ell(\hat{f}_\omega(x), y)$
$E = 1000$
$\epsilon$ = small value
$\omega_0$ = intial value in $t_0$
while $E > \epsilon$ do
    for i = 1,...,n do
        $\omega_{t+1} = \omega_t - \epsilon \nabla_\omega \ell(\hat{f}_\omega(x_i), y_i)$
    end for
    Compute $E = L_n(\omega_{t+1})$
end while
```

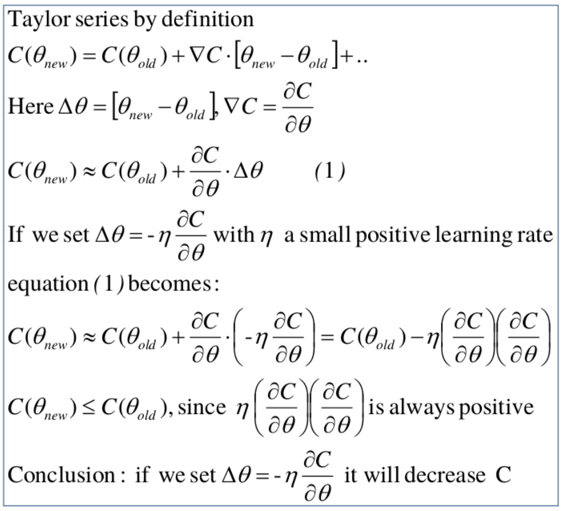
Algorithm 3 Stochastic and random gradient descent

```
Loss function $\widehat{L}_n(\hat{f}_\omega(x)) = \Sigma_{i=1}^n \ell(\hat{f}_\omega(x), y)$
$E = 1000$
$\epsilon$ = small value
$\omega_0$ = intial value in $t_0$
while $E > \epsilon$ do
    for i = 1,...,n do
        Random draw of $i \in \{1, ..., n\}$
        $\omega_{t+1} = \omega_t - \epsilon \nabla_\omega \ell(\hat{f}_\omega(x_i), y_i)$
    end for
    Compute $E = L_n(\omega_{t+1})$
end while
```

The main advantage of stochastic gradient is that it avoid computing the gradient descent on all the observations (greedy). However, in doing so, the gradient descent is subject to noise and can take longer to reach the optimum.

Note: the gradient is the derivation w.r.t ω
Note (stochastic and random gradient descent): the draw can be done with or without replacement.

Proof of the gradient descent formula:



Taylor series by definition

$$C(\theta_{new}) = C(\theta_{old}) + \nabla C \cdot [\theta_{new} - \theta_{old}] + ..$$

Here $\Delta\theta = [\theta_{new} - \theta_{old}]$, $\nabla C = \frac{\partial C}{\partial \theta}$

$$C(\theta_{new}) \approx C(\theta_{old}) + \frac{\partial C}{\partial \theta} \cdot \Delta\theta \qquad (1)$$

If we set $\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$ with η a small positive learning rate

equation (1) becomes:

$$C(\theta_{new}) \approx C(\theta_{old}) + \frac{\partial C}{\partial \theta} \cdot \left(-\eta \frac{\partial C}{\partial \theta}\right) = C(\theta_{old}) - \eta \left(\frac{\partial C}{\partial \theta}\right)\left(\frac{\partial C}{\partial \theta}\right)$$

$C(\theta_{new}) \leq C(\theta_{old})$, since $\eta \left(\frac{\partial C}{\partial \theta}\right)\left(\frac{\partial C}{\partial \theta}\right)$ is always positive

Conclusion: if we set $\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$ it will decrease C

(Idemia courses "DeepLearningTPT2018S1S2.pdf")

Learning rate optimization

Note:
- one epoch = one forward pass and one backward pass of all the training examples.
- batch size = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space needed.
- number of iterations = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass.

Learning rate decay
Simple idea: reduce the learning rate progressively.
E.g. $1/t$ decay:

$$\alpha_t = \frac{1}{(t+1)}$$

Momentum
Momentum is a method that helps accelerate SGD in the relevant direction and reduce oscillations.
General idea:
$0 \leq \gamma \leq 1$

$M_{t_0} = x_0$
$M_{t_1} = \gamma M_{t_0} + x_1$
$M_{t_2} = \gamma M_{t_1} + x_2$
Let's develop M_{t_2} to have a better view of momentum effect:
$M_{t_2} = \gamma(\gamma M_{t_0} + x_1) + x_2 = \gamma^2 x_0 + \gamma x_1 + x_2$
We can see that more importance is given to the most recent value (x_2) and the least to the past values.

In practice, $\gamma = 0.9$ gives good results.
Advantage: less dependent on noise

Adagrad - Adaptive Gradient Algorithm (2011)
Divide the learning rate by "average" gradient:
$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\Sigma_{i=0}^{t}(\nabla f_i^2)}}\nabla f$

RMSProp - Root Mean Squared Propagation
Same as AdaGrad, but with an exponentially decaying average of past squared gradients.
$\theta_t = \theta_{t-1} - \frac{\alpha}{\sigma_{t-1}}\nabla f$
Where:
$\sigma_t = \sqrt{\alpha(\sigma_{t-1})^2 + (1-\alpha)(\nabla f_t)^2}$

Adam - Adaptive moment estimation
Adam = RMS + momentum => use of a exponential decaying average of past <u>squared</u> gradients and past gradients

Bias-Complexity trade-off

$\mathcal{H}$ = hypothesis class = all the classifiers that are considered
The size of $\mathcal{H}$ can be seen as a measure of complexity.

We can decompose the error of an $ERM_{\mathcal{H}}$ (Empiric Risk Minimization algorithm):

$$L_{\mathcal{D}}(h_s) = \epsilon_{app} + \epsilon_{est}$$

- *Approximation error*: $\epsilon_{app} = min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$. This is the error done by the best predictor among those considered. It is the *bias* we have in choosing a specific class $\mathcal{H}$.

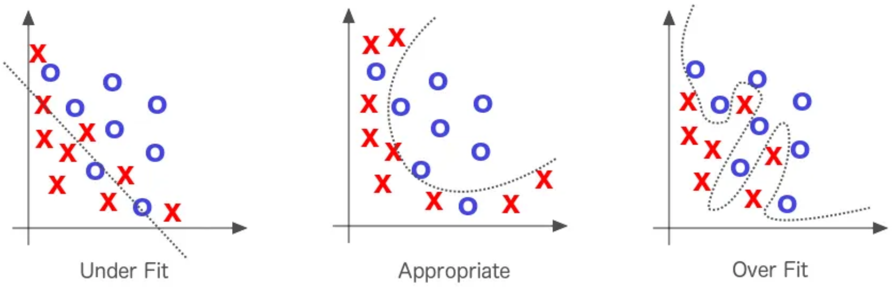
- *Estimation error*: $\epsilon_{est} = L_{\mathcal{D}}(h) - \epsilon_{app}$. This is the error difference from a used predictor and the best one. The larger $\mathcal{H}$ (complexity), the more predictors we consider and thus the larger ϵ_{est} is likely to be.

High complexity <=> ϵ_{app} (bias) low <=> ϵ_{est} high <=> overfitting
Low complexity <=> ϵ_{app} (bias) high <=> ϵ_{est} low <=> underfitting

We also call this trade-off the **bias-variance trade-off** since a high complexity leads to a high variance. For more details on the variance, see last section of the Trees chapter.

Overfitting

“Overfitting occurs when our hypothesis fits the training data "too well"”.
Understanding Machine Learning - From Theory To Algorithms.



Let S be a training set sampled according to the probability distribution $\mathcal{D}$. An algorithm A overfits if the difference between the *true risk* of its output $L_{\mathcal{D}}(A(S))$ and the *empirical risk* of its output $L_S(A(S))$ is large.

Note: Recall from the bias-complexity trade-off that overfitting is also when ϵ_{app} (error made by the best predictor) is low and thus ϵ_{est} (difference between the error of the best predictor and the error of the used predictor) is high.

Stability

An algorithm is stable if a small change in input gives a small change in output.

Let $S^{(i)}$ be a training set where we replace the *i-th* element by z': $S^{(i)} = (z_1, ..., z_{i-1}, z', z_{i+1}, ..., z_m)$. We measure an effect of a small change comparing the losses $\ell(A(S^{(i)}), z_i)$ and $\ell(A(S), z_i)$ (z_i is the value to predict).
We note that the algorithm trained on $S^{(i)}$ doesn't observe z_i while the algorithm trained on S does. Intuitively, we should have $\ell(A(S^{(i)}), z_i) - \ell(A(S), z_i) \geq 0$.

A is stable on average if $\mathbb{E}_{(S,z')\sim\mathcal{D}^{m+1}, i\sim U(m)}[\ell(A(S^{(i)}), z_i) - \ell(A(S), z_i)]$ is small.

Theorem:

$$\mathbb{E}_{S\sim\mathcal{D}}[L_{\mathcal{D}}(A(S)) - L_S(A(S))] = \mathbb{E}_{(S,z')\sim\mathcal{D}^{m+1}, i\sim U(m)}[\ell(A(S^{(i)}), z_i) - \ell(A(S), z_i)]$$

With $U(m)$ the uniform distribution over m.

Thanks to this theorem we have the following: A is a stable algorithm <=> second term is small <=> first term is small <=> A does not overfit

Regularization

In this part we show that using regularization leads to a stable algorithm.

Let A be a regularized algorithm:

$$A(S) = \underset{\omega}{\operatorname{argmin}}(L_S(\omega) + \lambda||\omega||^2)$$

$f_S : \omega \mapsto L_S(\omega) + \lambda||\omega||^2$

Lemma:
- f_S is 2λ-strongly convex
- For any v: $f_S(v) - f_S(A(S)) \geq \lambda||v - A(S)||^2$ since $A(S)$ minimizes f_S

We have:

$$\begin{aligned} f_S(v) - f_S(u) &= L_S(v) + \lambda||v||^2 - (L_S(u) + \lambda||u||^2) \\ &= L_{S^{(i)}}(v) + \lambda||v||^2 - (L_{S^{(i)}}(u) + \lambda||u||^2) \\ &\quad + \frac{1}{m}(\ell(v, z_i) - \ell(v, z')) - \frac{1}{m}(\ell(u, z_i) - \ell(u, z')) \end{aligned}$$

We removed the loss computed on the additional observation z' and we added the loss computed on observation z_i.

$$\begin{aligned} f_S(v) - f_S(u) &= L_{S^{(i)}}(v) + \lambda||v||^2 - (L_{S^{(i)}}(u) + \lambda||u||^2) \\ &\quad + \frac{\ell(v, z_i) - \ell(u, z_i)}{m} + \frac{\ell(u, z') - \ell(v, z')}{m} \end{aligned}$$

Choosing $v = A(S^{(i)})$ and $u = A(S)$:

$$\begin{aligned} f_S(A(S^{(i)})) - f_S(A(S)) &= L_{S^{(i)}}(A(S^{(i)})) + \lambda||A(S^{(i)}||^2 - (L_{S^{(i)}}(A(S)) + \lambda||A(S)||^2) \\ &\quad + \frac{\ell(A(S^{(i)}), z_i) - \ell(A(S), z_i)}{m} + \frac{\ell(A(S), z') - \ell(A(S^{(i)}), z')}{m} \end{aligned}$$

If $A(S^{(i)})$ minimizes $L_{S^{(i)}}(\omega) + \lambda||\omega||^2$ (optimal coefficient):

$L_{S^{(i)}}(A(S^{(i)})) + \lambda||A(S^{(i)}||^2 \leq L_{S^{(i)}}(A(S)) + \lambda||A(S)||^2$

Thus,

$f_S(A(S^{(i)})) - f_S(A(S)) \leq \frac{\ell(A(S^{(i)}),z_i) - \ell(A(S),z_i)}{m} + \frac{\ell(A(S),z') - \ell(A(S^{(i)}),z')}{m}$

Thanks to the above lemma:

$\lambda||A(S^{(i)}) - A(S)||^2 \leq \frac{\ell(A(S^{(i)}),z_i)-\ell(A(S),z_i)}{m} + \frac{\ell(A(S),z')-\ell(A(S^{(i)}),z')}{m}$ (I)

By definition, if $\ell(.,z_i)$ is ρ-Lipschitz:

$\ell(A(S^{(i)}),z_i) - \ell(A(S),z_i) \leq \rho||A(S^{(i)} - A(S)||$

Similarly,

$\ell(A(S^{(i)}),z') - \ell(A(S),z') \leq \rho||A(S^{(i)}) - A(S)||$

Plugging these two equations in (I):

$\lambda||A(S^{(i)}) - A(S)||^2 \leq \frac{2\rho||A(S^{(i)})-A(S)||}{m}$

$||A(S^{(i)}) - A(S)|| \leq \frac{2\rho}{\lambda m}$

Using ρ-Lipschitz definition:

$\frac{1}{\rho}(\ell(A(S^{(i)}),z_i) - \ell(A(S),z_i)) \leq \frac{2\rho}{\lambda m}$

$\ell(A(S^{(i)}),z_i) - \ell(A(S),z_i) \leq \frac{2\rho^2}{\lambda m}$

Since it is true for any S, z', i we conclude:

$$\mathbb{E}_{S\sim\mathcal{D}^m}[L_{\mathcal{D}}(A(S)) - L_S(A(S))] \leq \frac{2\rho^2}{\lambda m}$$

We have shown that a regularized algorithm is stable and thus does not overfit.

Supervised learning

Supervised learning aims at **finding a predictor when we have data with their labels**.

Perceptron

The Perceptron is the most basic algorithm for binary classification.

We want to estimate $f : \mathcal{X} \to \mathcal{Y}$ where $X \in \mathcal{X} \subset \mathbb{R}^p$ and $\mathcal{Y} = \{-1, 1\}$

Hyperplane

The classification is linear. We look for an hyperplane that separates the best the observations.

$$\mathcal{H} = \{x \in \mathbb{R}^p, \hat{f}_\omega(x) := \omega_0 + \Sigma_{i=1}^p \omega_i x_i = 0\}$$

We can also write:

$$\mathcal{H}_\omega : \omega^T x + \omega_0$$

Perceptron classifier: $x \mapsto sign(\hat{f}_\omega(x))$

Loss function

We want to find ω such that the loss function ℓ is minimised:
$\mathbb{E}[\ell(\hat{f}_\omega(x), y)]$ is the theoric risk to minimize.
Recall that ERM consists of minimizing the empiric risk.

Empiric risk

The following is an example of loss function that can be used for the Perceptron problem:

$$\widehat{L}_n(\hat{f}_\omega(x)) = -\Sigma Y_i(\omega_0 + \Sigma_{i=1}^p \omega_i x_i)$$

(when Y_i (target) and $\omega_0 + \Sigma_{i=1}^p \omega_i x_i$ (predicted) have different signs, the loss function increases)

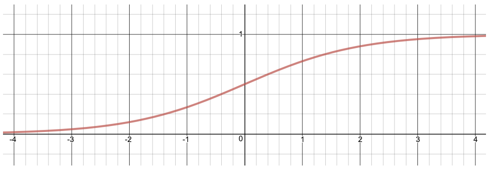
Logistic regression

Logistic regression is used for binary classification.
It is quite similar to a simple linear regression in the sense that the objective is to find optimal weights ω to predict a variable. However, in the logistic regression we use a sigmoïd function.

Rem: "logistic" because the logistic law has a sigmoïd function as a repartition function.

Rationale behind the use of the sigmoïd function:
We look for the *à posteriori* probability $\mathbb{P}(x|y = 1) = \pi(x) = \hat{y}$.
The predicted variable $\hat{y}$ is thus a probability.

The sigmoïd function: $\sigma : z \rightarrow \frac{1}{1+e^{-z}}$ is well adapted because we want an output that is included in $[0, 1]$.



Classification function: $\widehat{f_\omega}(x) = \sigma(\omega x)$ with a threshold

Loss function
If $y = 1$, we want $\sigma(\omega x)$ to be high => $1 - \sigma(\omega x)$ should be low. The loss function should be increasing with $1 - \sigma(\omega x) = 1 - \frac{1}{1+e^{-\omega x}} = \frac{1}{1+e^{\omega x}}$. Equivalently, it should be increasing with $1 + e^{-\omega x}$.
More generally, the loss function is defined as such: $\ell(f_\omega, (x, y)) = log(1 + e^{-y\omega x})$ (adding y in the expression allows to take into account cases when $y = 1$ and $y = -1$).
Recall that the log is a monotonic function.

Estimation
The advantage of the logistic loss function is that it is a convex function. Hence the ERM problem can be solved efficiently using standard methods.
Estimation is done using maximum likelihood. Maximum likelihood is finding the parameter that maximizes the probability to have a specific event (x_i, y_i). We want to maximize the *à posteriori* probability that depends on x:

$L(\omega, b) = \prod_{i=1}^{n} \pi(x_i)^{y_i}(1 - \pi(x_i))^{1-y_i}$

This equation has no analytic solution. We use a numeric method to find the optimal parameters (see optimizaton algorithms).
See *Neural Network* section for more details on optimization.

Note: logistic regression is really a linear model since the objective is to find ω that is the slope of the line $\omega^T x + b$.

KNN

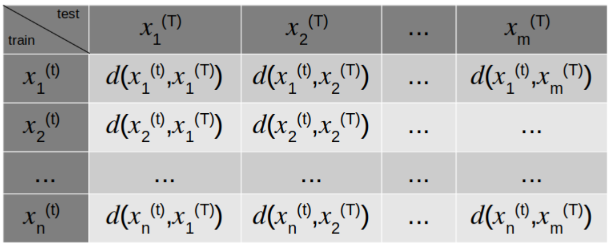
We want to approximate $f : \mathcal{X} \rightarrow \mathcal{Y}$, $\mathcal{X} \in \mathbb{R}^p$, $\mathcal{Y} = \{1, ..., L\}$

We note:
$x = (x_1, ..., x_p)^T \in \mathcal{X}$ a sample
The distance between 2 samples: $d : \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}$
$\mathcal{D}_n = \{(x_i, y_i), i = 1, ..., n\}$ the training set with n samples and labels.

For each new sample $x \in \mathbb{R}^p$, we determine the set of k nearest neighbors among all the train set.

Note: a sample is an observation, that is $x^T = (x_i^{(1)}, ..., x_i^{(p)})$. We thus compute the distances between **vectors** of dimension p.
The most basic distance metric is the Euclidean norm: $d(u, v) = ||u - v||^2 = \Sigma_{i=1}^{p}(u_i - v_i)^2$

The algorithm consists in building a distance matrix with the test sample in columns (power (t)) and the train sample in rows (power (T)) as shown below:



train \ test	$x_1^{(T)}$	$x_2^{(T)}$	…	$x_m^{(T)}$
$x_1^{(t)}$	$d(x_1^{(t)},x_1^{(T)})$	$d(x_1^{(t)},x_2^{(T)})$	…	$d(x_1^{(t)},x_m^{(T)})$
$x_2^{(t)}$	$d(x_2^{(t)},x_1^{(T)})$	$d(x_2^{(t)},x_2^{(T)})$	…	…
…	…	…	…	…
$x_n^{(t)}$	$d(x_n^{(t)},x_1^{(T)})$	$d(x_n^{(t)},x_2^{(T)})$	…	$d(x_n^{(t)},x_m^{(T)})$

Note: in this matrix, the training sample is of size n and the test sample is of size m.

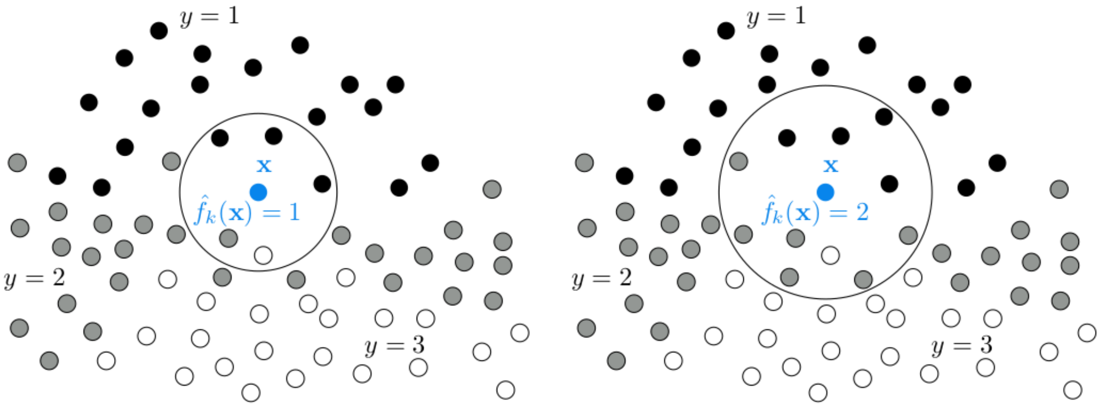
We then sort the matrix column-wise in ascending order in order to find train data points with smallest distance to the test data point.

For each test data point, we keep the k smallest distances (i.e. the k first rows of the sorted matrix) and we look at their labels. The Python function *argsort()* allows us to keep indices when sorting. We define the rank of a neighbor as:

$$r_k(x) = i^* \text{ if and only if:}$$
$$d(x_{i^*}, x) = \min_{1 \leq i \leq n,\ i \neq r_1, \ldots, r_{k-1}} d(x_i, x)$$

The most basic rule is to take the most frequent label among the k neighbors. In the below schema we consider $k = 5$ and $k = 11$ with three classes ($L = 3$) that are drawn respectively in black ($y = 1$), grey ($y = 2$) and white ($y = 3$).

$$\widehat{f}_k(x) = \underset{y \in \mathcal{Y}}{\operatorname{argmax}}(\Sigma_{j=1}^{k} \mathbb{1}\{y_{r_j} = y\})$$



(a) $k = 5$

(b) $k = 11$

Figure 1: Example of KNN prediction

Note: the *fit()* function of the *KNNClassifier* is just an affectation of the training data to the object; the distance computation is done in the *predict()*.

The biggest advantages of the KNN algorithm are:
- It's a simple and intuitive algorithm that can be easily explained and understood.
- Its parameter can be easily optimize e.g. using cross validation.
The biggest drawbacks are:
- The prediction can be greedy to compute because we need to compute all distances between test and train data.
- It has a high variance: the prediction can be strongly different if the data are slightly different.

Alternative

$$\widehat{f_k}(x) = \underset{y\in\mathcal{Y}}{\text{argmax}}(\Sigma_{j=1}^{k}\omega_j\mathbb{1}\{y_{r_j} = y\})$$

where:

$$\omega_j = e^{-d_j^2/h}$$

This alternative doesn't change the knn selection, it only changes the class attribution in the neighborhood. It gives more weights to very small distances. The higher h, the higher we favor small distances (*exp* function becomes steeper).

Listing 1: KNN algorithm

```
class KNNClassifier(BaseEstimator, ClassifierMixin):
    """ Homemade kNN classifier class """
    def __init__(self, n_neighbors=1):
        self.n_neighbors = n_neighbors

    # the training step only consists in storing training data
    def fit(self, X, y):
        self.X = X
        self.Y = y
        return self

    def predict(self, X):
        n = len(self.X) # size of train set
        m = len(X) # size of test set
        dist_mat = []
        for i in range(n): # we loop on every element of the train set
            dist_vect = []
            for j in range(m): # we loop on every element of the test set
                dist_vect.append(euclidean_distance(self.X[i], X[j]))
            dist_mat.append(dist_vect)
          # len(dist_vect) = m (nb of features; all the distances for one observation)

        dist_mat = np.asarray(dist_mat)
        # dist_mat.shape = (n,m); T_test in column, X_train in row

        # dist_mat = metrics.pairwise.pairwise_distances(
                                X, Y=self.X, metric='euclidean', n_jobs=1
```

```
                        )

idx_sort = np.argsort(dist_mat, kind='mergesort', axis=0)
# idx_sort.shape = (n,m); dist_mat sort column-wise
# mergesort is a stable way to handle equal numbers:
# if equal, the order of indices in the output is the same as in the input

idx_sort_knn = idx_sort[:self.n_neighbors,:] # resize with the number of knn
return getBestClassFromCount(idx_sort_knn, self.Y)
```

Linear discriminant analysis

We focus on the binary case, that is when $Y = +1$ or $Y = -1$.
These two conditional laws need to be gaussians with same covariance:
$X|Y = +1 \sim \mathcal{N}(\mu_+, \Sigma)$ with density f_+
$X|Y = -1 \sim \mathcal{N}(\mu_-, \Sigma)$ with density f_-
Let π_+, π_- be the simple probabilities $P(Y = +1)$, $P(Y = -1)$

$\mathbb{P}(Y = +1|X = x) = \frac{\mathbb{P}(Y=+1,X=x)}{\mathbb{P}(X=x)}$
$\mathbb{P}(Y = +1|X = x) = \frac{\mathbb{P}(X=x|Y=+1)\mathbb{P}(Y=+1)}{\mathbb{P}(X=x)}$
$\mathbb{P}(Y = +1|X = x) = \frac{f_+\pi_+}{\mathbb{P}(X=x)}$
$\mathbb{P}(Y = +1|X = x) = \frac{f_+\pi_+}{\mathbb{P}(X=x|Y=+1)\mathbb{P}(Y=+1)+\mathbb{P}(X=x|Y=-1)\mathbb{P}(Y=-1)}$
$\mathbb{P}(Y = +1|X = x) = \frac{f_+\pi_+}{(f_+\pi_++f_-\pi_-)}$
Similarly,
$\mathbb{P}(Y = -1|X = x) = \frac{f_-(1-\pi_+)}{\mathbb{P}(X=x)}$
$\mathbb{P}(Y = -1|X = x) = \frac{f_-(1-\pi_+)}{(f_+\pi_++f_-\pi_-)}$

The result shows us that we can express the two conditionnal probabilities in terms of conditionnal densities and "simple" probabilities (π_+, π_-).
Recall that multivariable gaussian density is: $f(x) = \frac{1}{\sqrt{2\pi|\Sigma|}}e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$
In practice, μ_+, μ_-, π_+ and Σ are unknown. Thus we use empiric values:
$\widehat{\pi}_+ = m/n$
$\widehat{\mu}_+ = \frac{1}{m}\Sigma\mathbb{1}_{\{y_i=+1\}}x_i$
$\widehat{\mu}_- = \frac{1}{n-m}\Sigma\mathbb{1}_{\{y_i=-1\}}x_i$
$\widehat{\Sigma} = \frac{1}{n-2}((m-1)\widehat{\Sigma}_+ + (n-m-1)\widehat{\Sigma}_-)$
$\widehat{\Sigma}_+ = \frac{1}{m-1}\Sigma\mathbb{1}_{\{y_i=+1\}}(x_i - \widehat{\mu}_+)(x_i - \widehat{\mu}_+)^T$
$\widehat{\Sigma}_- = \frac{1}{n-m-1}\Sigma\mathbb{1}_{\{y_i=-1\}}(x_i - \widehat{\mu}_-)(x_i - \widehat{\mu}_-)^T$

Classification
We predict class = 1 when $\mathbb{P}(Y = +1|X) > \mathbb{P}(Y = -1|X)$
=> $\frac{\mathbb{P}(Y=+1|X)}{\mathbb{P}(Y=-1|X)} > 1$
=> $\log(\frac{\mathbb{P}(Y=+1|X)}{\mathbb{P}(Y=-1|X)}) > 0$

Using previous conditional probability expressions, we end up with the following prediction rule:

$$\begin{cases}1 & \text{if } x^T\widehat{\Sigma}^{-1}(\widehat{\mu}_+ - \widehat{\mu}_-) > \frac{1}{2}\widehat{\mu}_+^T\widehat{\Sigma}\widehat{\mu}_+ - \frac{1}{2}\widehat{\mu}_-^T\widehat{\Sigma}\widehat{\mu}_- + \log(1-m/n) - \log(m/n)\\ -1 & \text{otherwise}\end{cases} \tag{2}$$

$\widehat{\mu}_+$, $\widehat{\mu}_-$, $\widehat{\pi}_+$ and $\widehat{\Sigma}$ will be computed with *train data*.
x is the *test data*.

Note: $\widehat{\Sigma}^{-1}(\widehat{\mu}_+ - \widehat{\mu}_-)$ is the **Fisher function** (Saporta).

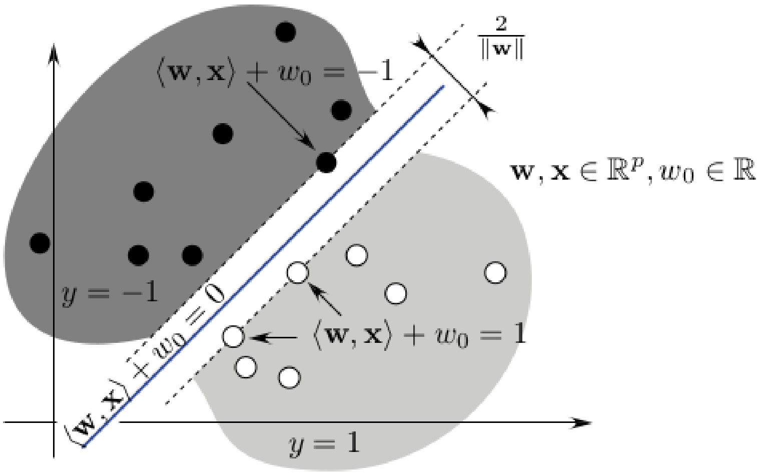
Listing 2: LDA algorithm

```
class LDAClassifier():

    def fit(self, X, y):

        X_p = X[y == 1, :]
        X_m = X[y == -1, :]

        X_p_x1 = X_p[:,0]
        X_p_x2 = X_p[:,1]
        X_m_x1 = X_m[:,0]
        X_m_x2 = X_m[:,1]

        n = len(X)
        m = len(X_p)

        mean_p_x1 = np.mean(X_p_x1)
        mean_p_x2 = np.mean(X_p_x2)
        mean_p = np.array([mean_p_x1,mean_p_x2]) # mu_plus (estimated)
        cov_p = np.cov(np.transpose(X_p))


        mean_m_x1 = np.mean(X_m_x1)
        mean_m_x2 = np.mean(X_m_x2)
        mean_m = np.array([mean_m_x1,mean_m_x2]) # mu_minus (estimated)
        cov_m = np.cov(np.transpose(X_m))

        cov_est = (1/(n-2))*( (m-1)* cov_p + (n-m-1)* cov_m) # sigma (estimated)
        inv_cov_est = np.linalg.inv(cov_est)

        a1 = np.dot(np.transpose(mean_p),inv_cov_est)
        a2 = np.dot(np.transpose(mean_m),inv_cov_est)

        # 2nd term in inequality
        self.alpha = 0.5*(np.dot(a1,mean_p) - 0.5*np.dot(a2,mean_m)) + np.log(1- m/n) - np.log
        # 1st term in inequality
        self.beta = np.dot(inv_cov_est,mean_p-mean_m)

        return self

    def predict(self, X):

        y_=[]
    for i in range(len(X)):
        X_pred = X[i]
        beta = np.dot(np.transpose(X_pred), self.beta)
        if (beta>self.alpha):
            Y_pred = 1
        else:
            Y_pred = -1
        y_.append(Y_pred)
    return np.array(y_)
```

SVM

Margin

This idea of SVM is to separate data as best as possible using a margin.


$\langle \mathbf{w}, \mathbf{x} \rangle + w_0 = -1$
$\frac{2}{\|\mathbf{w}\|}$
$\mathbf{w}, \mathbf{x} \in \mathbb{R}^p, w_0 \in \mathbb{R}$
$y = -1$
$\langle \mathbf{w}, \mathbf{x} \rangle + w_0 = 0$
$\langle \mathbf{w}, \mathbf{x} \rangle + w_0 = 1$
$y = 1$

Three planes:
- $H_1 : \omega^T x + b = 1$
- $H : \omega^T x + b = 0$
- $H_{-1} : \omega^T x + b = -1$

Computing $H_1 - H_{-1}$ we get:
$(x_1 - x_{-1})\omega^T = 2$
$=> ||x_1 - x_{-1}|| = \frac{2}{||\omega||}$

The SVM problem starts with a **margin maximization**. We want to maximize the distance between x_1 and x_{-1} **(to be double checked)** and it is equivalent to minimizing $||\omega||$. Thus the optimization problem is written as such:

$$min_{\omega,b} \ \frac{1}{2}||\omega||^2$$
$$s.t. \ y_i(\omega^T x_i + b) \geq 1 \ i = 1, ..., n$$

Primal formulation

The above problem reflects the case when all data are perfectly separable, this is not true in practice. We thus add an error term ξ_i for each observation. This leads to the **primal formulation** of the problem:

$$\begin{gathered} min_{\omega,b,\xi}\ \frac{1}{2}||\omega||^2 + C\Sigma_{i=1}^n \xi_i \\ s.t.\ y_i(\omega^T x_i + b) \geq 1 - \xi_i\ i = 1, ..., n \\ \xi_i \geq 0\ i = 1, ..., n \end{gathered}$$

Note: the problem can be rewritten with the hinge loss function

$$\begin{aligned} y_i(\omega^T x_i + b) \geq 1 - \xi_i &=> \xi_i \geq 1 - y_i(\omega^T x_i + b) \\ &=> \xi_i \geq 1 - y_i(\omega^T x_i + b) \geq 0 \text{ since } \xi_i \geq 0 \\ &=> \xi_i = max(0, 1 - y_i(\omega^T x_i + b)) \\ &=> \xi_i = (0, 1 - y_i(\omega^T x_i + b))_+ \end{aligned}$$

$=> \xi_i = hinge(f(x))$ where *hinge* is the *Hinge loss* function
$hinge(f(x)) = (1 - yf(x))_+$

$$min_{\omega,b}\ \frac{1}{2}||\omega||^2 + C\Sigma_{i=1}^n (0, 1 - y_i(\omega^T x_i + b))_+$$

Lagrange

We can write this problem using Lagrange formulation, that is, integrating the constraints into the main formula:
$\mathcal{L}(\omega, b, \xi, \alpha, \mu) = \frac{1}{2}\omega^T\omega + C\Sigma\xi_i + \Sigma\alpha_i(1 - \xi_i - y_i(\omega^T x_i + b)) - \Sigma\mu_i\xi_i$
$\alpha_i, \mu_i \geq 0$

First order conditions:
$\frac{\partial\mathcal{L}}{\partial\omega} = 0 => \omega - \Sigma\alpha_i y_i x_i = 0 => \omega = \Sigma\alpha_i y_i x_i$
$\frac{\partial\mathcal{L}}{\partial b} = 0 => -\Sigma\alpha_i y_i = 0$
$\frac{\partial\mathcal{L}}{\partial\xi_i} = 0 => C - \alpha_i - \mu_i = 0 => \alpha_i = C - \mu_i$
Since $\alpha_i, \mu_i \geq 0$ we have $C \geq \alpha_i \geq 0$

Dual formulation

We can rewrite the problem using the first order conditions above:

$$\begin{aligned} \mathcal{L}(\omega, b, \xi, \alpha, \mu) &= \frac{1}{2}(\Sigma\alpha_i y_i x_i)^T(\Sigma\alpha_i y_i x_i) + C\Sigma\xi_i + \Sigma\alpha_i - \Sigma\alpha_i\xi_i - \Sigma\alpha_i y_i(\Sigma\alpha_i y_i x_i)^T x_i - \Sigma\alpha_i y_i b - \Sigma\mu_i\xi_i \\ &= -\frac{1}{2}(\Sigma\alpha_i y_i x_i)^T(\Sigma\alpha_i y_i x_i) + \Sigma(C - \alpha_i - \mu_i)\xi_i + \Sigma\alpha_i - \Sigma\alpha_i y_i b \\ &= -\frac{1}{2}\Sigma_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j + \Sigma\alpha_i \end{aligned}$$

The problem is convex with lineary inequality constraints, we can apply the saddle point theorem.
Note: a point is saddle if it's a maximum w.r.t. one axis and a minimium w.r.t. another axis
The saddle theorem allows us to solve the problem $min_\omega max_\alpha$ as $max_\alpha min_\omega$

This leads to the problem in its **dual formulation**:

$$\begin{gathered} max_\alpha\ \ -\frac{1}{2}\Sigma_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j + \Sigma\alpha_i \\ s.t.\ \ 0 \leq \alpha_i \leq C\ \ i = 1, ..., n \\ \Sigma\alpha_i y_i = 0\ \ i = 1, ..., n \end{gathered}$$

Using the above expression of ω (optimal condition), the classification function is:

$$f(x) = sign(\Sigma \alpha_i y_i x_i^T x + b)$$

Kernel

Kernels are used when separation is non linear.
Recall the primal formulation:

$$\begin{gathered} min_{\omega,b,\xi}\ \frac{1}{2}||\omega||^2 + C\Sigma_{i=1}^n \xi_i \\ s.t.\ y_i(\omega^T x_i + b) \geq 1 - \xi_i\ i = 1, ..., n \\ \xi_i \geq 0\ i = 1, ..., n \end{gathered}$$

When separation is non linear, we set ϕ as a non linear transformation. The constraint becomes: $y_i(\omega^T \phi(x_i) + b) \geq 1 - \xi_i \;\; i = 1, ..., n$

Dual formulation is now:

$$\begin{gathered} max_\alpha \;\; -\frac{1}{2}\Sigma_{i,j} \alpha_i \alpha_j y_i y_j \phi(x_i)^T \phi(x_j) + \Sigma \alpha_i \\ s.t.\;\; 0 \leq \alpha_i \leq C \;\; i = 1, ..., n \\ \Sigma \alpha_i y_i = 0 \;\; i = 1, ..., n \end{gathered}$$

The classification becomes:

$$f(x) = sign(\Sigma \alpha_i y_i \phi(x_i)^T \phi(x) + b)$$

To classify a new point, we thus need to be able to compute $\phi(x_i)^T \phi(x)$.

Kernel trick: there is no need to know an explicit expression of ϕ (i.e. knowing the coordinates of points in new set) since we are only looking at distances and angles, that is scalar product.

Kernel functions implement those scalar products: $K(x, x') = \phi(x)^T \phi(x')$ where ϕ is a transformation function into a Hilbertian set $\phi : \mathcal{X} \rightarrow \mathcal{F}$
Note: an Hilbertian is a set with scalar product: $\mathcal{F} = (\mathcal{H}, <.,.>)$

Most used kernels:
Linear kernel: $K(x, x') = x^T x'$ (we often call this setup a "no-kernel SVM")
Polynomial kernel: $K(x, x') = (x^T x' + c)^d$
Gaussian kernel: $K(x, x') = exp(-\gamma ||x - x'||^2)$

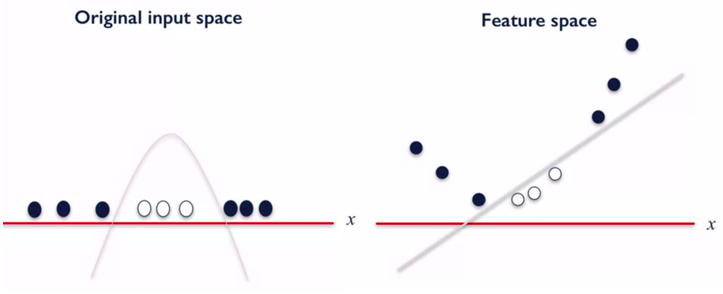
The optimisation problem can be written with kernel:

$$\begin{gathered} max_\alpha \;\; -\frac{1}{2}\Sigma_{i,j} \alpha_i \alpha_j y_i y_j K(x_i, x_j) + \Sigma \alpha_i \\ s.t.\;\; 0 \leq \alpha_i \leq C \;\; i = 1, ..., n \\ \Sigma \alpha_i y_i = 0 \;\; i = 1, ..., n \end{gathered}$$

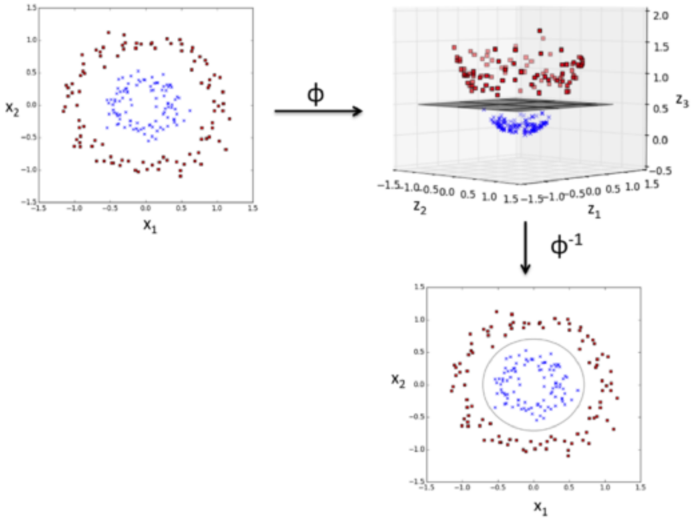
Summary

SVM allows to find complex non linear separations in transforming the problem into a higher dimension where data are linearly separable.

From 1D to 2D:



From 2D to 3D:



Neural network

Logistic regression with a neural network mindset
Based on the example from coursera (*deeplearning.ai*)
Let us take the example of an image that we want to classify in a **binary** way: man/woman
The picture is vectorized as a vector of pixels : $\begin{pmatrix} x_1 \\ \dots \\ x_p \end{pmatrix}$
We use a regression to predict if it's a man/woman:
$y = \omega^T x + b$
Note: x are all the pixels of **one** image.

We want a probability in output (if it's ≥ 0.5 then we say it's a man).
We thus want the output to be $\widehat{y} = \sigma(\omega^T x + b) = \mathbb{P}(y|x) \in [0,1]$
(see regression part to get more details on the sigmoid)

Now since it's a binary classification, we want the y (real value) to be 0 or 1.
Thus, the loss function is:

$$\mathcal{L}(y, \widehat{y}) = -y\log(\widehat{y}) + (1-y)\log(1-\widehat{y})$$

The cost function is the empiric loss on all examples:

$$J(\omega, b) = \frac{1}{m}\Sigma_{i=1}^{m}\mathcal{L}(\widehat{y}^{(i)}, y^i)$$

Forward propagation

$$x_1, x_2, \omega_1, \omega_p, b \rightarrow z = \omega_1 x_1 + \omega_2 x_2 + b \rightarrow \widehat{y} = a = \sigma(z) \rightarrow \mathcal{L}(a, y)$$

- First arrow: regression
- Second arrow: probability
- Third arrow: error

Backward propagation
The idea is: with the error computed on the last step, we go backward in order to correct the parameters ω and b.

$$x_1, x_2, \omega_1, \omega_p, b \leftarrow z = \omega_1 x_1 + \omega_2 x_2 + b \leftarrow \widehat{y} = a = \sigma(z) \leftarrow \mathcal{L}(a, y)$$

Example: we want to find ω_1 that minimizes the cost function:
$\frac{d\mathcal{L}}{d\omega_1} = "d\omega_1" = \frac{d\mathcal{L}}{da}\frac{da}{dz}\frac{dz}{d\omega_1} = ... = (a-y)x_1 = dzx_1$
Steps:
We compute all the derivatives, then we apply the gradient descent

Listing 3: Gradient descent (logistic regression with a NN mindset)

```
for i in range(num_iterations):

    # Cost and gradient calculation
    grads, cost = propagate(w, b, X_train, Y_train) # propagation on ALL the training sample

    # Retrieve derivatives from grads
    dw = grads["dw"]
    db = grads["db"]

    # update parameters
    w = w - learning_rate * dw
    b = b - learning_rate * db

    # Record the costs
    costs.append(cost)
```

Listing 4: Propagation (logistic regression with a NN mindset)

```
def propagate(w, b, X, Y):

    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    A = sigmoid(np.dot(w.T,X)+b)
    cost = (- 1 / m) * np.sum(Y * np.log(A) + (1 - Y) * (np.log(1 - A))

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = (1/m)*np.dot(X,(A-Y).T)
    db = (1/m)*np.sum(A-Y)
```

Trees

Intro

We want to approximate $f : \mathcal{X} \to \mathcal{Y}$ where $\mathcal{Y}$ are the labels and $\mathcal{X}$ is the training set.
The objective is to partitionate $\mathcal{X}$ the best. At the start, $\mathcal{X}$ is the first node (root).

Training step: we build a tree that best classifies the training samples.
Predict step: we perform the tests of the tree that was built during the training step.

Impurity measure: Gini index

To determine a good split element, we measure the impurity. One way to measure impurity is the Gini index.

The Gini index is the probability of incorrectly classifying a random point. Thus, a good separator has a **low** Gini index.

Mathematically, the Gini index H (hyperplan) represents the disparity in a set S:

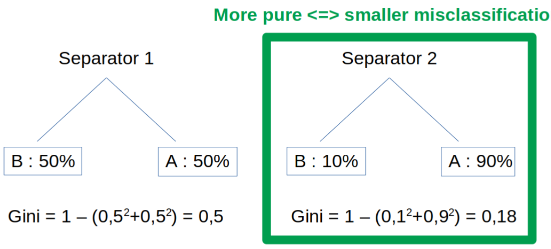
$$H(S) = \Sigma_{\ell=1}^{C} p_\ell(S)(1 - p_\ell(S))$$

Or (easier to code):

$$H(S) = \Sigma_{\ell=1}^{C} p_\ell(S) - \Sigma_{\ell=1}^{C} p_\ell(S) * p_\ell(S) = 1 - \Sigma_{\ell=1}^{C} p_\ell(S)^2$$

Where $p_c(S) = \frac{1}{n}\Sigma_{i=1}^{n} 1\{y_i = c\}$ is the frequency of label c in a set S. C is the set of labels.

Note: if all elements belong to the same class, then it is called pure. Intuitively, to have more purity, we will favor a high count of elements in a small number of classes versus a low count of elements in a large number of classes. This is because $p_\ell(S)$ is at power 2. E.g. $3^2 + 4^2 < 7^2$ => we prefer to have the 7 elements in the same class (more pure).


More pure <=> smaller misclassification risk
Separator 1
B : 50%
A : 50%
Gini = 1 – (0,5^2+0,5^2) = 0,5
Separator 2
B : 10%
A : 90%
Gini = 1 – (0,1^2+0,9^2) = 0,18

Loss function

To assess the split quality, we use *information gain* (= loss function) that compares the impurity (Gini index) between two trees. We compare the current tree with the one where we applied the split. The trees are weighted by the child importance.

$$\underset{j\in\{1,\ldots,p\},\tau\in\mathbb{R}}{\operatorname{argmin}} \frac{n_r}{n}H(R(S,j,\tau)) + \frac{n_l}{n}H(L(S,j,\tau))$$

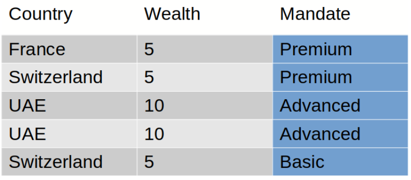
(j,τ) are the possible feature values. $H()$ is the Gini index function as seen previously.
R and L are the Right and Left branches. We want a **low** disparity inside each of them.

Why linearity?

$R(S,j,\tau) = \{(x,y), t_{j,\tau} \geq 0\}$ is the set of elements at the right of the node. The inequality $t_{j,\tau} \geq 0$ represents the split element which is a **linear separator**. Trees thus use several linear separators to build non linear decision functions.

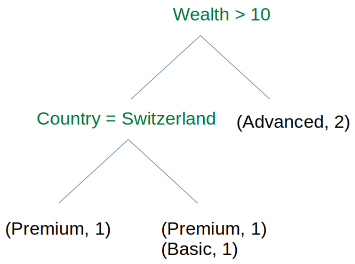
Example

In our example, we want to build a tree to classify client mandates the best as possible.



Country	Wealth	Mandate
France	5	Premium
Switzerland	5	Premium
UAE	10	Advanced
UAE	10	Advanced
Switzerland	5	Basic

The expected tree for this training set is:



The leafs are dictionaries of occurences. Those dictionaries can be used as a reliability measure for a prediction. E.g. if the client has a wealth $<$ 10 and is located in Switzerland, there are

50-50% chance that the client is Premium or Basic.

Algorithm

The main steps of the algorithm are:
1. Loop on all features and values to find the element that best splits the data.
2. Create a partition thanks to this element.

3. Perform the same two first steps for the left and the right of the element.
4. Stop when you arrive to a Leaf.

The main function *build_ tree()* is called recursively.

Algorithm 4 Trees

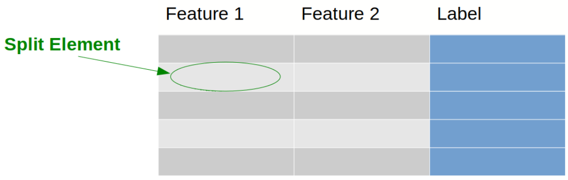
```
procedure BUILD_TREE(ROWS)
    gain, split_element = FIND_BEST_SPLIT(rows)
    if gain==0 then
        return Leaf(rows)
    end if
    right_rows, left_rows = PARTITION(rows, split_element)
    right_branch = BUILD_TREE(right_rows)
    left_branch = BUILD_TREE(left_rows)
    return Decision_Node(split_element, right_branch, left_branch)
end procedure
```

A full implementation (largely inspired by Google tutorial) can be found on my github account.

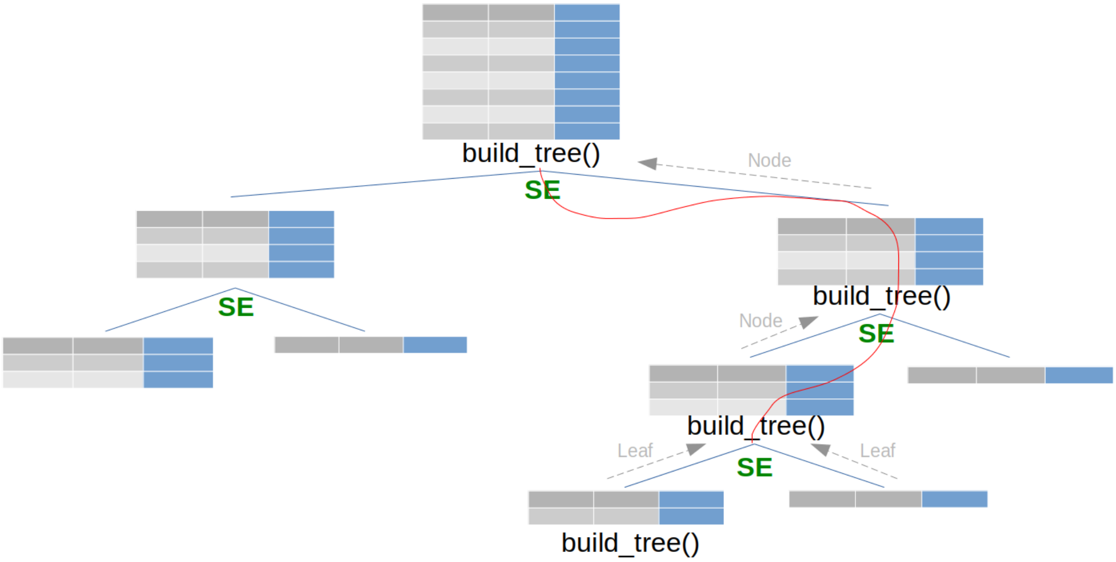
Important notes

The final tree is a *node*. Browsing the tree can be done as such:
my_tree.right_branch.right_branch.leaf.
This gives a dictionary of occurences for a branch with depth 3 (number of nodes).

The function FIND_BEST_SPLIT() loops on all features and all values to find the best split element. A split element is thus a couple (value, feature):



The following tree illustrates the recursivity. In green are the *split element*.



Variance

Decision trees have a high variance meaning that the classification results can highly vary for different datasets. In other words, there is a high sensitivity to training data. If we draw Decision Trees for slightly different datasets (that we would however consider the same), we will have a large dispersion of the labels:

$$Var_m(y) = \frac{1}{M}\Sigma_{i=1}^{M}(y_i - \overline{y})^2$$

Where M are the considered datasets (almost similar) and y_i are all the labels of one tree.
Intuitively, this problem lies in the hierarchy process of a tree: a small change in a node is echoed in every child nodes.

Feature importance

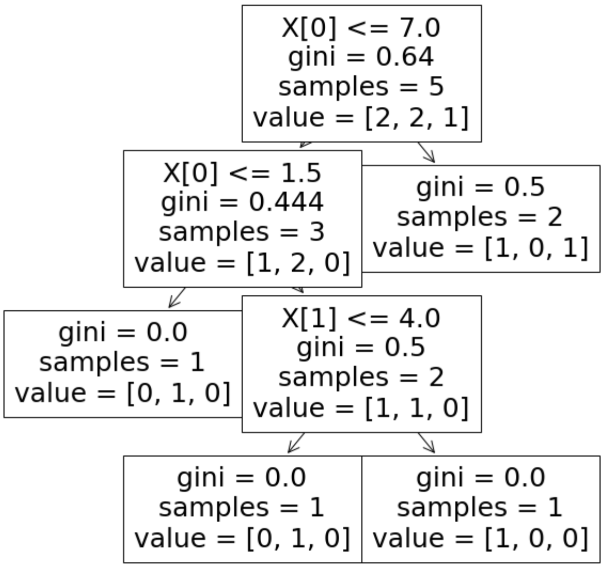
scikit-learn provides a *feature_importance_* method associated with the Tree classifier. Feature importance is computed as such:

$$Node\ Importance\ (node) = NI(node) = \frac{N_t}{N}(Gini_t - \frac{N_{t,right}}{N_t}Gini_{right} - \frac{N_{t,left}}{N_t}Gini_{left})$$

The terms $\frac{N_t}{N}$, $\frac{N_{t,right}}{N_t}$, $\frac{N_{t,left}}{N_t}$ are the probabilities to reach, respectively, the node we are interested in, its right child and left child.

$$Feature\ Importance\ (feat) = FI(node) = \frac{\Sigma_{k\in \text{ nodes with } \textit{feat} \text{ as separator}} NI(node\ k)}{\Sigma_{k\in \text{ all nodes}} NI(node\ k)}$$

Example:



$NI_0^{(1)} = \frac{5}{5}(0.64 - \frac{2}{5}0.5 - \frac{3}{5} * 0.444) \approx 0.174$

$NI_0^{(2)} = \frac{3}{5} * (0.444 - \frac{2}{3} * 0.5 - \frac{1}{3} * 0) \approx 0.066$
$NI_1 = \frac{2}{5} * (0.5 - \frac{1}{2} * 0 - \frac{1}{2} * 0) \approx 0.2$

$FI_0 = \frac{NI_0^{(1)} + NI_0^{(2)}}{NI_0^{(1)} + NI_0^{(2)} + NI_1} \approx 0.545$
$FI_1 = \frac{NI_1}{NI_0^{(1)} + NI_0^{(2)} + NI_1} \approx 0.454$

The full code for this example can be found on my Github account.

Ensemble methods

Ensemble methods are good to reduce variance.

There are several types of ensemble methods:

Bagging (Random Forests)

Bagging methods reduce variance and handle overfitting.
Predictors are chosen independently.

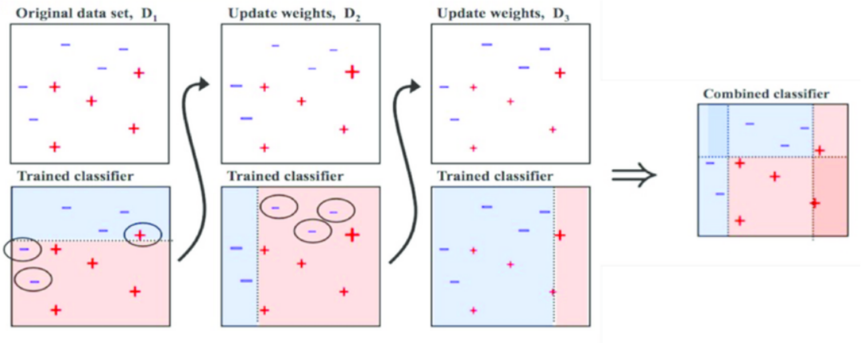
Boosting (AdaBoost, Gradient Boosting)

Boosting methods have the advantage of reducing the variance and the bias. However, these algorithms can overfit.
Predictors are chosen sequentially: predictors learn from the mistakes of the previous ones. The choice of the stopping criteria is critical to prevent overfitting.

AdaBoost

Boosting is an algorithmic paradigm addressing two major issues in machine learning:
- It optimizes the bias-complexity trade-off. The learning starts with a basic class (large approximation error) and as it progresses the hypothesis class becomes more complex.
- It allows to find predictors that are usually computationally infeasible to find.

Main idea: weak learners are "boosted" to become stronger altogether.



Weak learner (or γ-weak-learner): it's an **algorithm** returning a function h such that $L_{\mathcal{D}}(h) \leq 1/2 - \gamma$. In other words, it returns a simple binary predictor that does slightly better than a random guess.

Algorithm 5 AdaBoost

```
Input: training set $S = (x_1, y_1), ..., (x_m, y_m)$, weak learner WL, number of rounds $T$
Initialize $D^{(1)} = (1/m, ..., 1/m)$ // same weights for all observations
for t = 1,...,T do
    invoke weak learner $h_t = WL(D^{(t)}, S)$
    compute $\epsilon_t = \Sigma_{i=1}^{m} D_i^{(t)} \mathbb{1}_{[y_i \neq h_t(x_i)]}$ // misclassification error
    let $\omega_t = \frac{1}{2} log(\frac{1}{\epsilon_t} - 1)$ // $\epsilon_t < 1$
    update $D_i^{(t+1)} = \frac{D_i^{(t)} exp(-\omega_t y_i h_t(x_i))}{\Sigma_{j=1}^{m} D_j^{(t)} exp(-\omega_t y_j h_t(x_j))}$ for all $i = 1, ..., m$
end for
Output: the hypothesis $h_s(x) = sign(\Sigma_{t=1}^{T} \omega_t h_t(x))$
```

We note:
- Final predictor = weighted sum of weak predictors
- More weights are given to observations that gave wrong prediction. In doing so, the classifier of the next round will focus on these observations. Warning: to see this, focus on the variation of $D_i^{(t+1)}$ and not just w_t.

Theorem: the training error of the output hypothesis decreases **exponentially fast** with the number of boosting rounds.

GBM

Gradient Boosting Method is an agorithm that is part of ensemble methods.
There are many different implementation of this algorithm; the below one (src: StatQuest) is one of the most generic and understandable.

1. Initialize model with constant value
$F_0 = \underset{\hat{y}}{argmin}\ \Sigma_{i=1}^{n} \mathcal{L}(y_i, \hat{y})$

```
2. for $m = 1$ to $M$:
    (A) Compute pseudo-residuals:
    $r_{im} = -\frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)}$ for $i = 1, ..., n$
    (B) Fit a weak learner with $r_m$ as target values
    (C) For $j = 1, ..., J_m$ (leafs) compute $\gamma_{jm} = \underset{\gamma}{argmin}\ \Sigma_{x_i \in R_{ij}} \mathcal{L}(y_i, F_{m-1}(x_i) + \gamma)$
    (D) Update $F_m(x) = F_{m-1}(x) + \alpha \Sigma_{j=1}^{J_m} \gamma_{jm} \mathbb{1}(x \in R_{jm})$ for all records $x$
3. Output $F_M$
```

Step 1: initialize model with constant value

$$F_0 = \underset{\hat{y}}{argmin} \Sigma_{i=1}^{n} \mathcal{L}(y_i, \hat{y})$$

We note that $\hat{y}$ is constant and doesn't depend on i. Consequently, optimizing this equation using MSE will lead to $\hat{y} = \bar{y}$:
Note: $MSE := \frac{1}{n}\Sigma_{i=1}^{n}(y_i - \hat{y})^2$ thus to use MSE in our case we need to have $\mathcal{L} : (y, \hat{y}) \rightarrow (y - \hat{y})^2$ but we usually choose $\mathcal{L} : (y, \hat{y}) \rightarrow \frac{1}{2}(y - \hat{y})^2$

$\frac{\partial MSE}{\partial \hat{y}} = 0$
$=> \frac{2}{n}(y_1 - \hat{y}) + ... + \frac{2}{n}(y_n - \hat{y}) = 0$
$=> \hat{y} = \Sigma_{i=1}^{n} \frac{y_i}{n} = \bar{y}$

Step 2: loop on the number of estimators

(A) Pseudo-residuals

$$r_{im} = -\frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)} \text{ for } i = 1, ..., n$$

The gradient of the loss function is $\frac{\partial \mathcal{L}}{\partial \hat{y}} = -(y - \hat{y})$
Residuals as seen in linear regression are typically written as such: $res = y - \hat{y}$. Thus, gradient boosting uses negative residuals, also called **pseudo-residuals**.
Note 1: pseudo-residuals are just the derivatives of the loss function; that's why we call it **gradient** boosting.
Note 2: $F(x_i)$ is equivalent to writing $\hat{y}$. It's the prediction of each record.
Reminder: the prediction of a decison tree is found in navigating from the root to the leaf where the sample is classified.

(B) Fit a weak learner with r_m as target values

Listing 5: Fit base learner

```
h = tree.fit(x, res)
```

Weak learners (see definition in Adaboost section) used in gradient boosting are typically decision trees. These learners are trained on residuals.
Thus, this step consists in building a small tree.

(C) Compute a unique output for each leaf

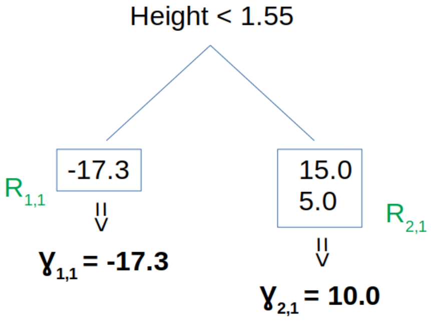
We could call this step: "harmonization of predicted values".

For $j = 1, ..., J_m$:

$$\gamma_{jm} = \underset{\gamma}{argmin} \ \Sigma_{x_i \in R_{ij}} \mathcal{L}(y_i, F_{m-1}(x_i) + \gamma)$$

The loop "For $j = 1, ..., J_m$" means we iterate on all leafs of the previously built tree.
This step consists in building a unique output for each of its leaf. $R_{j,m}$ are the terminal regions, that is, the leafs of the tree.

In the below picture the first weak learner is shown. Here $J_m = 2$ since there are 2 leafs only. We also consider the MSE as a loss function, so γ_{jm} is equal to the averages of the leaf outputs.



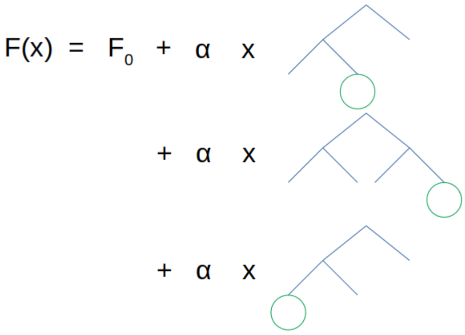
(D) Update the function

$$F_m(x) = F_{m-1}(x) + \alpha \Sigma_{j=1}^{J_m} \gamma_{jm} \mathbb{1}(x \in R_{jm})$$

α is the *learning rate* used to reduce overfitting.
The summation is here only in case one record x ends up in several leafs.

Step 3: Output the function

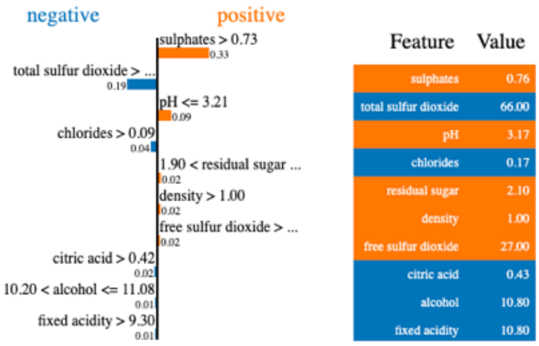
F is the output of the algorithm and is just a combination of several trees. To find the predicted value of a record, we would need to sum all predicted values of each tree (weighted by the learning rate α). Below is an illustration of the computation of the final prediction for a record x:



LIME

Local Interpretable Model-agnostic Explanations

LIME is a model allowing to explain why an algorithm made a specific decision on a specific observation. It is model-agnostic since the method can explain decisions without understanding how the classifier works. By "explaining" we mean displaying feature importance for the decision. Typical output is:



Feature	Value
sulphates	0.76
total sulfur dioxide	66.00
pH	3.17
chlorides	0.17
residual sugar	2.10
density	1.00
free sulfur dioxide	27.00
citric acid	0.43
alcohol	10.80
fixed acidity	10.80

On this picture, we see the features contributing to wine quality: sulphates positively contributes to the wine quality while sulfur dioxyde contributes negatively.

Model

The main idea behind LIME is to perturb data and learn an interpretable model locally.

Input: observation we want to explain x_0.

Step 1: local perturbation of x_0
A neighborhood is created around x_0. By default, LIME uses a Gaussian sampling: it generated variables from a normal distribution. Then it de-standardizes generated samples using mean and standard deviation from the observation to explain.
Note: by default, LIME generates a neighborhood of 5000 samples.

Step 2: weight computation
Weights π_{x_0} are computed according to the distance between x_0 and its neighbors. LIME uses an exponential smoothing kernel:

$$K(x, y) = e^{\frac{||x-y||_2^2}{\sigma^2}}$$

Reminder: $||x - y||_2 = \sqrt{\Sigma_{i=1}^{n}(x_i - y_i)^2}$

Step 3: local classification
A linear model is used to classify samples around x_0. A linear model is used because it's an *interpretable* model (such as decision trees).
Loss minimization to explain x_0:

$$\xi(x_0) = \underset{g \in G}{\text{argmin}}\ \mathcal{L}(f, g, \pi_{x_0}) + \Omega(g)$$

Weights π_{x_0} are taken into account in the loss function.
$\Omega()$ is a complexity measure. By default, LIME uses a Ridge regression, thus $\Omega()$ is the 2-norm. It would also make sense to use a lasso regularized regression (1-norm) in order to allow zero coefficients (and thus have variable selection).

Step 4: display explanations
Feature importance is displayed looking at regression coefficients.

SHAP

The use of SHAP values is a simple method to measure the impact of features on any model. Intuitively, it consists in adding/removing a feature and measure the impact of the removal on the prediction. Thus, it aims at finding *marginal* contributions of features.
The concept of SHAP is based on Lloyd Shapley's results on game theory. Shapley's main idea is that members should receive payments proportional to their marginal contributions to the society.
Note: the explanation of the game theory part is based on Stanford tutorial.
The main question is: what weighting system should we use?

Let N be a coalition with all members involved, and S a coalition with any member(s).
v is the value brought to the society by a specific coalition.

Shapley considers several axioms:

Symmetry: for all S that contains neither i nor j, i and j are interchangeable if $v(S \cup \{i\}) = v(S \cup \{j\})$. Then interchangeable agents should receive the same payments: $\phi_i(N, v) = \phi_j(N, v)$. Intuitively, agents who contribute the same should receive the same payments.

Dummy players: i is a dummy player if the amount that i contributes to any coalition is 0: for all S, $v(S \cup \{i\}) = v(S)$. Then for any v, if i is a dummy player: $\phi_i(N, v) = 0$.
Intuitively, dummy players should receive nothing.

Additivity: if we can separate a game in two parts: $v = v_1 + v_2$, then we should be able to decompose the payments: $\phi_i(N, v_1 + v_2) = \phi_i(N, v_1) + \phi_i(N, v_2)$.
For instance, we can consider a game that consists in producing fruits during two days. An agent will receive equally the fruits production of day 1 and day 2 than receiving the total production for these two days.

Based on the previous axioms, it turns out that there is only one way of defining the payment of each agent:

$$\phi_i(N,v) = \frac{1}{|N|!}\Sigma_{S\subseteq N\setminus\{i\}}|S|!(|N|-|S|-1)!(v(S\cup\{i\})-v(S))$$

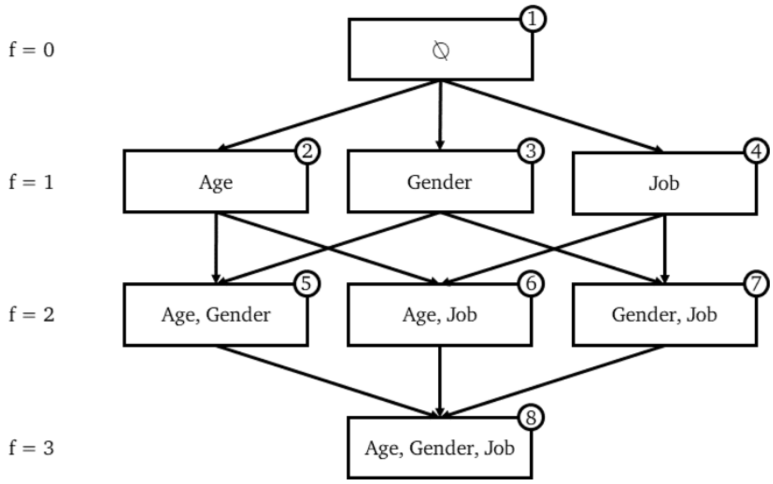
Warning: here $|.|$ is the *cardinality* of a set (not the absolute value function!)

$v(S\cup\{i\})-v(S)$ is the marginal contribution of agent i.
The rest of the formula consists in weighting the marginal contribution with all possible coalitions. More specifically:
$|S|!$ is the number of times S could have been formed.
$(|N|-|S|-1)!$ is the number of times the remaining agents could have been added.
$\frac{1}{|N|!}$ is used to average across all different possible coalitions.

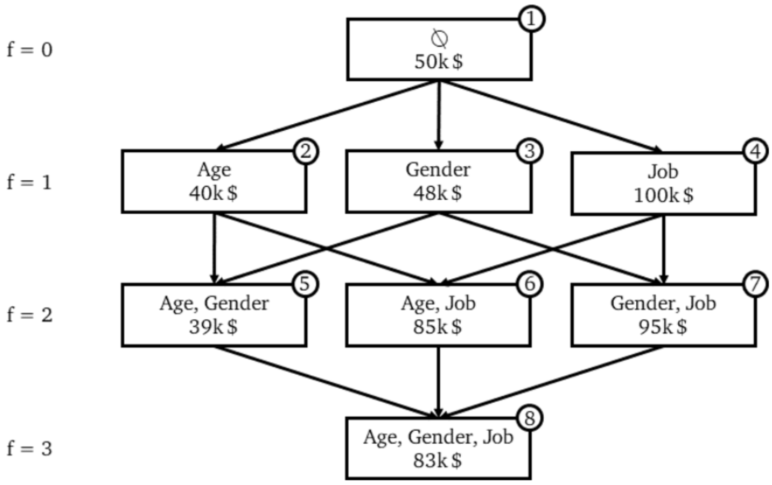
The next part of this section is inspired by a very good tutorial on how the Shapley value is applied to machine learning.

Here, the "game" is what produces the outcome of a model. The "agents" or "players" are the features of the model.

Let's consider a model that consists in predicting the income of an individual based on the age, the gender and the job.
To define all the possible coalitions, we use the mathematical concept *power set*:



Once the training is done, we compute the prediction for each coalition:

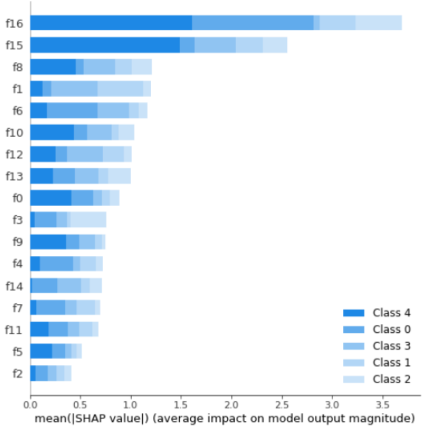


We note that an edge (or an arrow) is the effect of a marginal contribution.

With machine learning notation, the marginal contribution is:

$$MC_{feat1,\{feat1,feat2\}} = Predict_{feat1,feat2}(X) - Predict_{feat2}(X)$$

Note: for classification problems that output probabilities, the marginal contribution is the difference in probabilities for each class. The final output of SHAP is split by classes:.



$MC_{feat1,\{feat1,feat2\}}$ should be read: "the effect of feature 1 in coalition {feat 1, feat 2}". In the above example, $MC_{Age,\{Age\}}(X) = 40k\$ - 50k\$ = -10k\$$.

The SHAP value is defined as such:

$$\begin{aligned} SHAP_{Age}(X) &= \omega_1 MC_{Age,\{Age\}}(X) \\ &= +\omega_2 MC_{Age,\{Age,Gender\}}(X) \\ &= +\omega_3 MC_{Age,\{Age,Job\}}(X) \\ &= +\omega_4 MC_{Age,\{Age,Gender,Job\}}(X) \end{aligned}$$

Under the constraint: $\omega_1 + \omega_2 + \omega_3 + \omega_4 = 1$

Thus, the SHAP value is the average of all impacts a feature brings to every coalition.

But how to find the weights?

The weights are based on the number of edges for the specific level of the power set we are looking at. Mathematically, it's the inverse of the function $f * C_f^F$. For example, for the level $f = 2$, $\omega_3 = \frac{1}{2 * C_2^3} = \frac{1}{6}$.

Warning: if feature 1 contributes more than feature 2, it doesn't mean that feature 1 improves more the model than feature 2! As mentioned by the author, "SHAP values of a model's output explain how features impact the output of the model, not if that impact is good or bad".

One of the main limits of SHAP is that it has a lot to compute. **The cardinality of a power set is 2^n.** This is thus the number of model to train (with the same parameters and input data) to compute the Shapley value (n = number of features = F in our case). Fortunately, the method used approximation without computing all possible cases.

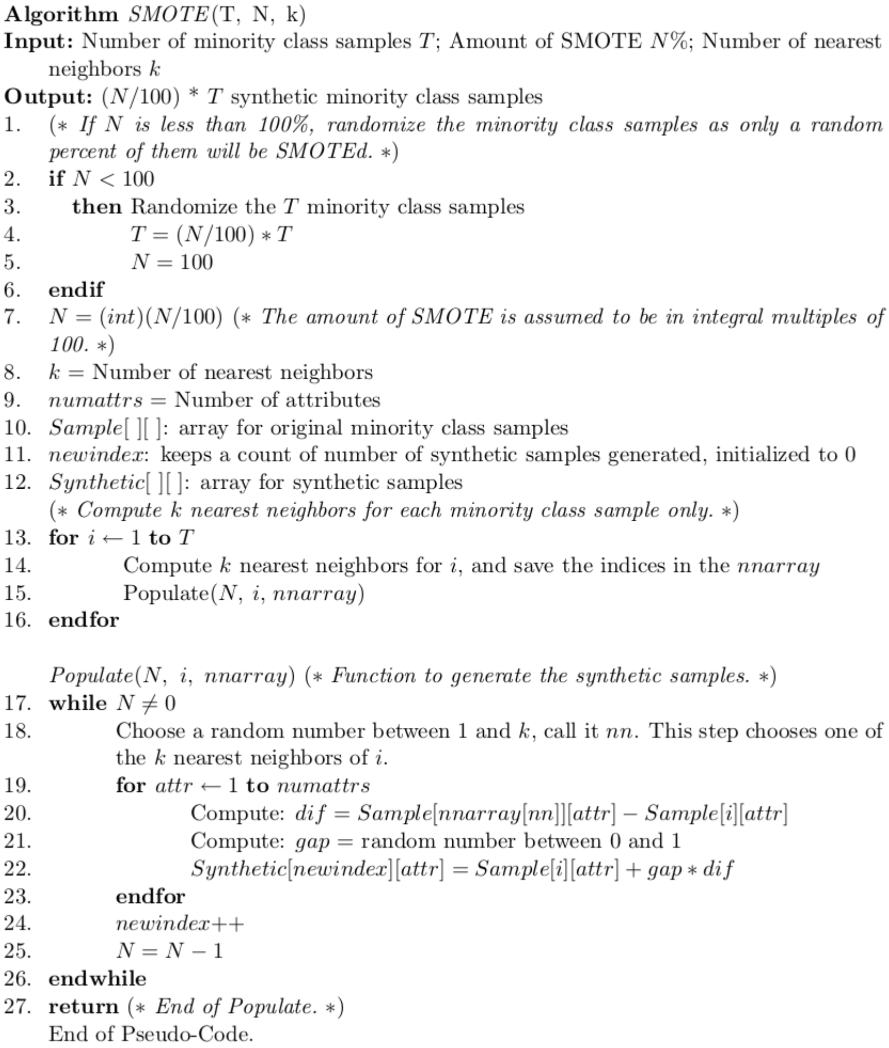
Unsupervised learning

Unsupervised learning aims at **learning some underlying hidden structure of the data when we don't have the labels**.
Unsupervised models can be used as a pre step for supervised learning, e.g.:
- reduce the training sample (dimensionality reduction: forward selection, PCA, autoencoders)
- give output for unlabeled data (clustering, autoencoders)
- grow the training sample (generative models)

SMOTE

SMOTE is an oversampling method to generate data by drawing a distance between a point and a random neighbor. It is used to grow the size of a minority class.

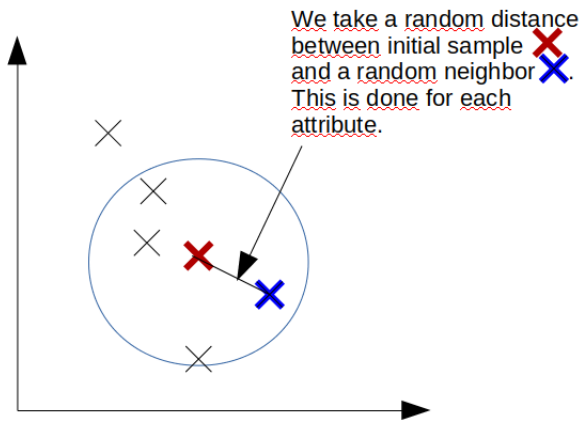
Note: the first part (until line 6) is the algorithm preliminaries. It allows to define what part of the minority class is going to be used for the new data generation.



```
Algorithm SMOTE(T, N, k)
Input: Number of minority class samples $T$; Amount of SMOTE $N$%; Number of nearest
    neighbors $k$
Output: $(N/100) * T$ synthetic minority class samples
(* If N is less than 100%, randomize the minority class samples as only a random
    percent of them will be SMOTEd. *)
if $N < 100$
    then Randomize the $T$ minority class samples
        $T = (N/100) * T$
        $N = 100$
endif
$N = (int)(N/100)$ (* The amount of SMOTE is assumed to be in integral multiples of
    100. *)
$k$ = Number of nearest neighbors
$numattrs$ = Number of attributes
$Sample[\,][\,]$: array for original minority class samples
$newindex$: keeps a count of number of synthetic samples generated, initialized to 0
$Synthetic[\,][\,]$: array for synthetic samples
(* Compute k nearest neighbors for each minority class sample only. *)
for $i \leftarrow 1$ to $T$
        Compute $k$ nearest neighbors for $i$, and save the indices in the $nnarray$
        Populate($N$, $i$, $nnarray$)
endfor

Populate($N$, $i$, $nnarray$) (* Function to generate the synthetic samples. *)
while $N \neq 0$
        Choose a random number between 1 and $k$, call it $nn$. This step chooses one of
        the $k$ nearest neighbors of $i$.
        for $attr \leftarrow 1$ to $numattrs$
                Compute: $dif = Sample[nnarray[nn]][attr] - Sample[i][attr]$
                Compute: $gap$ = random number between 0 and 1
                $Synthetic[newindex][attr] = Sample[i][attr] + gap * dif$
        endfor
        $newindex$++
        $N = N - 1$
endwhile
return (* End of Populate. *)
End of Pseudo-Code.
```

Algorithm 6 SMOTE (simplified)

```
for $sample$ in $all\_samples$ do
    Choose a random neighbor
    for $attribute$ in $all\_attributes$ do
        Compute a random weighted distance between $sample$ attribute and $k$ attribute
        Assign this distance to the newly generated point
    end for
end for
```


We take a random distance between initial sample ✕ and a random neighbor ✕. This is done for each attribute.

Expectation-Maximization (EM) in the case of GMM (Gaussian Mixture Model)

(for more details, see document *gmm.pdf* in Cloud folder)

GMM problem aims at estimating parameters of a sample distribution. ("Generative models" p. 295 Understanding Machine Learning).

A GMM sample is composed of j Gaussian variables (*clusters*) distributed with proportions $(\pi_1, ..., \pi_k)$ $(\Sigma\pi_i = 1)$
We can write:

$$X \sim \mathcal{N}(\mu_Z, \Sigma_Z) \qquad with \quad Z \sim \pi$$

π is not really a law but more the proportions of each Gaussian categories.
Thus, X has a density which is a weighted-average of all Gaussian densities:

$$p_\theta(x) = \Sigma_{j=1}^{k} \pi_j f_j(x) \qquad (*)$$

Estimation
We want to estimate $\theta = (\pi, \mu, \Sigma)$ where:
$\pi = (\pi_1, ..., \pi_k)$, $\mu = (\mu_1, ..., \mu_k)$, $\Sigma = (\Sigma_1, ..., \Sigma_k)$

To do so, we use the maximum likelihood method (product of densities across all samples):

$$p_\theta(x) = \Pi_{i=1}^{n} p_\theta(x_i)$$

$$l(\theta) = log(\Pi_{i=1}^{n} p_\theta(x_i)) = \Sigma_{i=1}^{n} log(p_\theta(x_i))$$

We thus need to find $argmax(l(\theta))$
Problem: the likelihood function is not convex!

The expectation-maximization problem is used when we have *latent variables* (= variables for which we don't know their associated distribution).

Let $z = (z_1, ..., z_k)$ be the vector of latent variables. We can express the density $(*)$ as a joint function with respect to z:

$$p_\theta(x, z) = p_\theta(z)p_\theta(x|z)$$

$$l(\theta, z) = ... = \Sigma(log\pi_{z_i}) + \Sigma(logf_{z_i}(x_i))$$

A classic optimization (in case of Gaussians) give us empirical values as solutions e.g. $\hat{\pi_j} = \frac{n_j}{n}$
Problem: we don't know j!

We will thus use the *expected* log-likelihood method.
Let us find another expression of the likelihood:

$$p_\theta(x, z) = p_\theta(x)p_\theta(z|x)$$

As seen previously: $p_\theta(x, z) = \Pi\pi_{z_i}f_{z_i}(x_i)$
$p_\theta(z|x) = \Pi p_\theta(z_i|x_i) = \frac{\Pi\pi_{z_i}fz_i(x_i)}{p_\theta(x_i)} \propto \Pi\pi_{z_i}fz_i(x_i)$

Given an initial parameter θ_0, the *expected* log-likelihood is written as such:

$$\mathbb{E}_{\theta_0}[l(\theta; z)] = \Sigma p_{\theta_0}(z|x)l(\theta; z)$$

$$\mathbb{E}_{\theta_0}[l(\theta; z)] = \Sigma_j\Sigma_i p_{ij}(log\pi_j + logf_j(x_i))$$

We now have an expression that doesn't depend on z but only on p_{ij} and we know that $n_j = \Sigma_i p_{ij}$

K-means

(see kmeans.pdf from OneDrive folders for more details)

Objective: group data into k clusters so that samples in the same cluster are close to each other w.r.t. the Euclidean distance.

The cost function minimization is written as such:

$$\underset{C_1,...,C_k;\mu_1,...,\mu_k}{\text{argmin}} \quad \Sigma_{j=1}^k\Sigma_{i\in C_j}||x_i - \mu_j||^2$$

Where μ_j is the mean, also called gravity center or cluster center:

$$\mu_j = \frac{1}{|C_j|}\Sigma_{i\in C_j}x_i$$

The quality of the clustering strongly depends on the initial center values. This is why the algorithm is generally run multiple times for different initial values. The best clustering (i.e., that of minimum cost) is returned.

K-means++
To improve the quality of the clustering, we choose the initial cluster centers far from each other:
- select the first cluster center uniformly at random among the n data samples

Algorithm 7 K-means

```
Input: $x_1, ..., x_n$
Output: clusters $C_1, ..., C_k$
Random values for $\mu_1, ..., \mu_k$
while no convergence do
    // Step 1: Update clusters
    $C_1, ..., C_k \leftarrow \emptyset$
    for $i = 1$ to $n$ do
        $j \leftarrow argmin_l ||x_i - \mu_l||$
        $C_j \leftarrow C_j + \{i\}$ // We add observation $i$ to the cluster $C_j$
    end for
    // Step 2: Update cluster centers
    for $j = 1$ to $k$ do
        $\mu_j \leftarrow 0$
        $n_j \leftarrow 0$
        for $i$ in $C_j$ do // We loop on all observations of each cluster
            $\mu_j \leftarrow \mu_j + x_i$
            $n_j \leftarrow n_j + 1$
        end for
        $\mu_j \leftarrow \mu_j / n_j$
    end for
end while
```

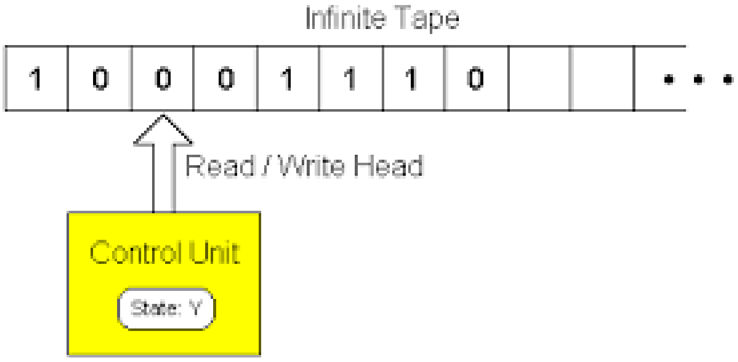
- select the following cluster centers at random, with a probability proportional to the square distance to the closest current cluster center

Listing 6: K-means++ initial centers selection

```
centers = []
centers.append(X[np.random.randint(X.shape[0])]) # inital center = one random sample
distance = np.full(X.shape[0], np.inf) # a vector (n,1) with only infinity terms
for j in range(1,self.n_clusters):
    distance = np.minimum(np.linalg.norm(X - centers[-1], axis=1), distance) # size (n,1);
# distance = the smallest distance associated with
# the last added center
    p = np.square(distance) / np.sum(np.square(distance)) # probability vector [p1,...,pn]
# the highest probability in p is associated
# with the biggest distance w.r.t the last added center
    sample = np.random.choice(X.shape[0], p = p) # one sample is
                                                 # selected according to probabilities
    centers.append(X[sample])
```

Note: this problem is called *NP-hard problem*. It means that its complexity is at least equal to the complexity of an NP-problem
NP-problem: a problem is NP if it can be determined by a non-deterministic Turing machine in polynomial time. Intuitively, a problem is NP if we can quickly verify if one candidate is a solution of the problem. E.g. "travelling salesman problem" = let d be a distance and n be a number of cities. Is there an itinerary with distance $\geq d$ stopping by every city? -> easy to check...

Turing machine (1936)



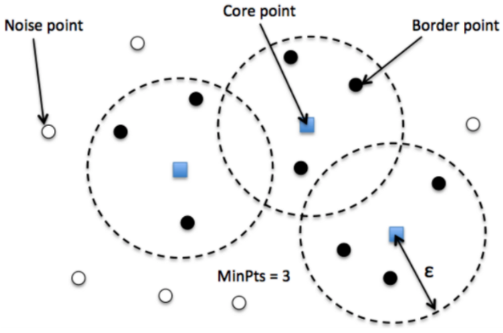
"non-deterministic turing machine": itinerary can be represented by a tree...

DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an algorithm allowing to find clusters based on a density and classication of points.
Note: density can be seen as the number of points in one's neighborhood.

The algorithm has two main parameters:
ϵ is the size of the neighborhood.
$MinPts$ is the minimum points in ϵ-neighborhood to define a cluster.

Every point is classified among three categories:
- A *core* point has at least $MinPts$ points in its ϵ-neighborhood.
- A *border* point has less than $MinPts$ points in its neighborhood but at least one core point is present.
- An *outlier* is neither a core nor a border point.



DBSCAN uses *density-reachability* to navigate through points and identify clusters.
Density-reachability: a point y is *density-reachable* from x if there is a path $p_1, ..., p_n$ with $p_1 = x$ and $p_n = y$ where each p_{i+1} on the path must be core points with the possible exception of p_n.

Algorithm 8 DBSCAN (simplified)

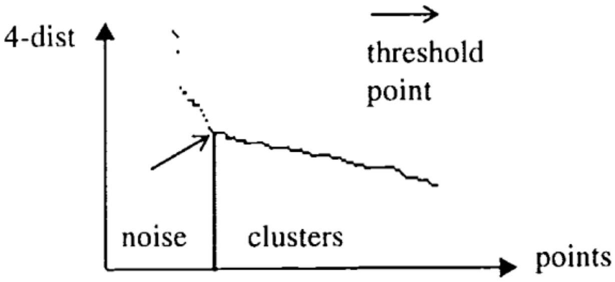
```
while some points are unclassified do
    pick random point
    if classified & core point then
        for neighbor in density_reachable_neighbors do
            if outlier then
                change to border
            end if
            add to cluster
        end for
    else
        if core point then
            label as core point
        else
            label as outlier
        end if
    end if
end while
```

Its main advantages over k-means is that it can find non convex clusters as well as detecting outliers (noise).

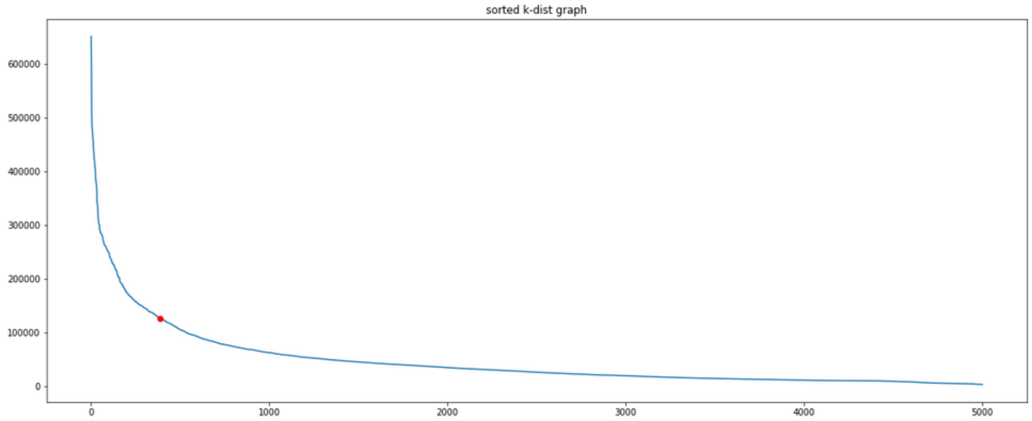
Parameter estimation

High $MinPts$ or low ϵ means higher density is necessary to form a cluster.

The original DBSCAN paper proposes a method to define ϵ. The method consists in plotting the k-th distance to each point in decreasing order. The largest values (left of the graph) are associated with outliers; smaller values are associated with cluster points. The inflection point is the **cluster point** with the highest k-distance; it corresponds to the epsilon we are looking for.



The authors mention that finding the threshold point automatically is rather difficult. They suggest that the user has to estimate the percentage of noise. I propose the following method to determine the threshold dynamically:
- Compute the slope of the line that links the first and last point in the graph.
- Find the two successive points with the closest slope.
The threshold will be the first of the two found points.



Local Outlier Factor

Local Outlier Factor is an unsupervised method used in anomaly detection. It consists of comparing local density of train observations VS local density of test observations.

Reachability distance

$reachability\text{-}distance_k(A, B) = max\{k\text{-}distance(B), d(A, B)\}$ = reachability of A *from* B.

$k\text{-}distance(B)$: distance from B to its kth nearest neighbor.
The reachability distance of A from B is *at least* the distance between A and B or *at least* the distance of B's neighbor.
When A is very far from B, it's simply the distance between the two points.
When A is very close to B, it's the distance between B and its neighbor.

The distance can be computed using different metrics: Euclidean distance, Mahalanobis distance, etc.

Local reachability density

$$lrd_k(A) = \frac{1}{\Sigma_{B \in N_k(A)} reachability\text{-}distance_k(A,B)/|N_k(A)|}$$

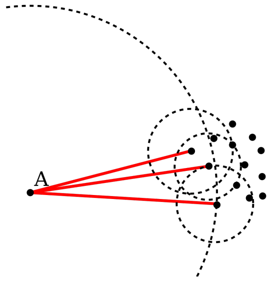
It's the inverse of the average of reachability-distances of A from B.
When A is very far from its neighbors: sum of reachability distances is high => local reachability density is small.

Local Outlier Factor

LOF computation consists of comparing the local densities of a point VS its neighbors.

$$LOF_k(A) := \frac{\frac{\Sigma_{B \in N_k(A)} lrd_k(B)}{lrd_k(A)}}{|N_k(A)|}$$

$LOF_k(A) > 1$: A is an outlier. Local reachability density of A is small compared to its neighbors.
$LOF_k(A) < 1$: A is an inlier.



On this figure, $k = 3$. We can see that the reachability distance of A from its neighbors is high (red segments). The local reachability density of A will thus be **low**.
On the contrary, the local reachability densities of its neighors is **high** because each neighbor can be easily reached from their own neighbors.
As a result, LOF would be high so A is an outlier.

sklearn algorithm

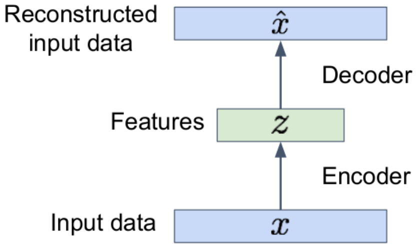
To score an observation, *fit* simply memorizes the train observations (same as in knn).
score_samples first finds the k-nearest neighbors from the train set thanks to the given distance metric. It then computes the local outlier factor for each test observation comparing the test observation local density with its closest k-neighbors local densities in the train set.

Variational Auto-Encoder

Variational autoencoders are a combination of three things:
1. Autoencoders
2. Variational Approximation & Variational Lower Bound
3. "Reparameterization" Trick

1. Autoencoders
Autoencoders are used to extract features from unlabeled training data. They are new methods for **dimensionality reduction** and part of neural networks branch.

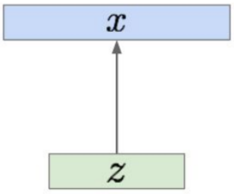
Note: autoencoders can be used to replace older dimensionality reduction methods such as PCA for several reasons:
- For very large data sets that can't be stored in memory, PCA will not be able to be performed. The autoencoder construction using keras can easily be batched resolving memory limitations.
- PCA is restricted to linear separation while autoencoders are capable of modelling complex non linear functions.



Learning can be done using a loss function such as $||x - \widehat{x}||^2$. In a similar way than neural network, optimization is typically done with backpropagation.

2. Variational Approximation & Variational Lower Bound

We assume x is generated from unobserved (latent) z:



Practical example: x can be seen as images and z as the main attributes (orientation, colors, etc.)

$x \sim p_{\theta^*}(x|z)$ where $p_{\theta^*}(x|z)$ is called *true conditional*
$z \sim p_{\theta^*}(z)$ where $p_{\theta^*}(z)$ is called *true prior*

Objective: estimating $p_\theta(x)$. We thus need to estimate θ^*.

We can do it through maximum likelihood. The marginal density is $p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz$

Note: a marginal likelihood function is a likelihood function in which some parameter variables have been marginalized. Marginalization consists in summing over the possible values of one variable in order to determine the contribution of another. E.g., $\mathbb{P}(X) = \Sigma_y \mathbb{P}(X, Y = y)$ or in continuous probabilities $p(x) = \int p(x, y)dy$. Also, if we don't know the joint probability, we can express this using conditional probabilities: $p(x) = \int p(x|y)p(y)dy$

Problem: impossible to compute $p(x|z)$ for every z (computationally too expensive) => problem is said **intractable**

Solution: use another encoder learning $q_\phi(z|x)$ that approximates $p_\theta(z|x)$

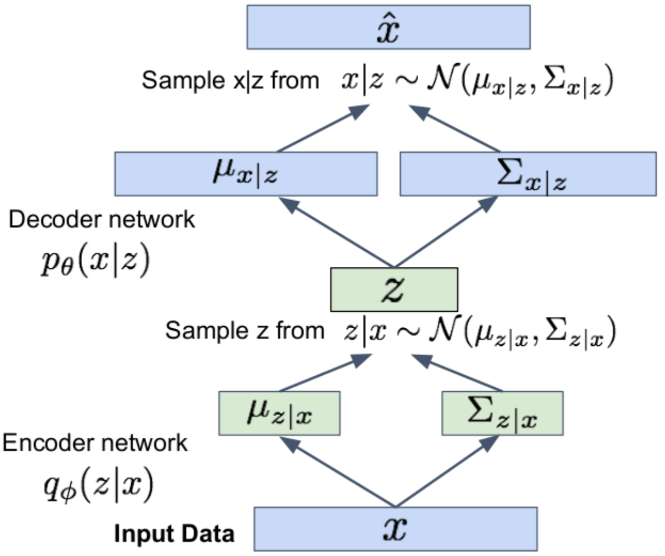
$$\begin{aligned}
\log p_\theta(x^{(i)}) &= \mathbf{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta(x^{(i)})\right] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z)\\
&= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)}\mid z)p_\theta(z)}{p_\theta(z\mid x^{(i)})}\right] \quad (\text{Bayes' Rule})\\
&= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)}\mid z)p_\theta(z)}{p_\theta(z\mid x^{(i)})}\frac{q_\phi(z\mid x^{(i)})}{q_\phi(z\mid x^{(i)})}\right] \quad (\text{Multiply by constant})\\
&= \mathbf{E}_z\left[\log p_\theta(x^{(i)}\mid z)\right] - \mathbf{E}_z\left[\log \frac{q_\phi(z\mid x^{(i)})}{p_\theta(z)}\right] + \mathbf{E}_z\left[\log \frac{q_\phi(z\mid x^{(i)})}{p_\theta(z\mid x^{(i)})}\right] \quad (\text{Logarithms})\\
&= \mathbf{E}_z\left[\log p_\theta(x^{(i)}\mid z)\right] - D_{KL}(q_\phi(z\mid x^{(i)})\,||\,p_\theta(z)) + D_{KL}(q_\phi(z\mid x^{(i)})\,||\,p_\theta(z\mid x^{(i)}))
\end{aligned}$$

$\mathbb{E}_z[\log p_\theta(x^{(i)}|z]$: we can estimate this term through sampling
$D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z))$: differentiable term
$D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z|x^{(i)}))$: $p(z|x)$ intractable but we know that $D_{KL} \geq 0$

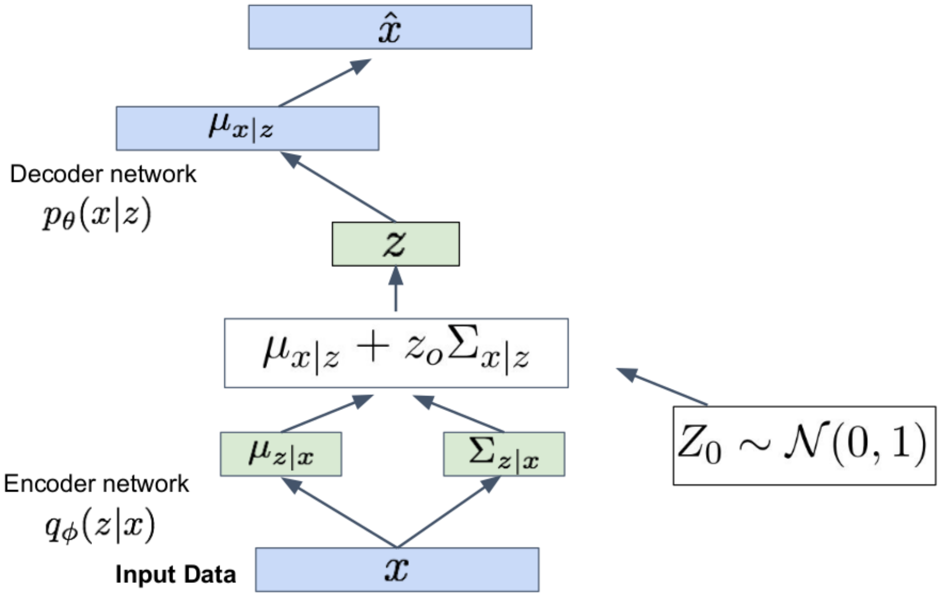
Let $\mathcal{L}(x^{(i)}, \theta, \phi) = \mathbb{E}_z[\log p_\theta(x^{(i)}|z] - D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z))$ = **tractable lower bound** that we can optimize

We know that $p_\theta(x^{(i)}) \geq \mathcal{L}(x^{(i)}, \theta, \phi)$ since $D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z|x^{(i)})) \geq 0$
Thus the maximum likelihood problem becomes: $\theta^*, \phi^* = argmax_{\theta,\phi}\Sigma_{i=1}^N \mathcal{L}(x^{(i)}, \theta, \phi)$

We can minimize $D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z))$ making posterior distribution close to prior. To do so, we make encoder network predicting $\mu_{z|x}$ and $\Sigma_{z|x}$ and then we sample $z|x \sim \mathcal{N}(\mu_{z|x}, \Sigma_{z|x})$



Problem: sampling $z|x \sim \mathcal{N}(\mu_{z|x}, \Sigma_{z|x})$ and $x|z \sim \mathcal{N}(\mu_{x|z}, \Sigma_{x|z})$ is not differentiable (why?).
=> we use **reparametrization trick**: we sample $z_0 \sim \mathcal{N}(0, 1)$ to have $z = \mu_{x|z} + z_0\Sigma_{x|z} \sim \mathcal{N}(\mu_{x|z}, \Sigma_{x|z})$



Optimization through forward and backward propagation!

Reinforcement learning

Reinforcement learning is inspired on human logic: we learn which strategy to take thanks to rewards we receive.

Reinforcement learning uses Markov Decision Processes (MDP).
A Markov Decision Process is defined by:
- an initial state s_0
- the reward distribution $r_t \sim p(r|s_t, a_t)$ (stochastic)
- the transition probabilities, $s_{t+1} \sim p(s|s_t, a_t)$ (stochastic)
=> **MDP: we know the state and the reward from the previous state only**.

A *policy* is an action for each state, for a given MDP.
=> **a policy can be seen as a strategy: we know where to go at each state**. In practice the policy is actually the **transition probability matrix**.

The *value function* is the gain we earn at a state, for a specific policy: $\forall s, V_\pi(s) = \mathbb{E}_\pi[G|s_0 = s]$
=> **The expected gain takes into account the transition probability.**
The gain is calculated as such: $G = r_0 + \gamma r_1 + \gamma^2 r_2 + ... = \Sigma_t \gamma^t r_t$ where γ is the *discount factor*.
V can also be written $V_\pi(s) = \mathbb{E}_\pi[r_0 + \gamma V(s_1)|s_0 = s]$. This expression is called **Bellman equation**.

We can find the best policy:

$\forall s, \pi^*(s) = a^* \in argmax_a \mathbb{E}_\pi[r_0 + \gamma V_*(s_1)|s_0 = s, a_0 = a]$
Where V_* is solution of the **Bellman optimality equation**:
$\forall s, V(s) = max_a \mathbb{E}[r_0 + \gamma V(s_1)|s_0 = s]$

Online estimation

The expected value function is approached using empiric estimator (sum).
Two ways to do it:
- Monte-Carlo update: if we can memorize all the paths, we update the sum at each step $S \leftarrow x_t$. At the end we compute the mean $X \leftarrow \frac{S}{t}$
- TD-learning: we update the value function using temporal differences $\forall s, V(s_t) \overset{\alpha}{\leftarrow} r_t + \gamma V(s_{t+1})$
Where $X \overset{\alpha}{\leftarrow} x_t <=> X = X + \alpha(x_t - X)$ (α is usually $1/t$)

Online control

Recall that value function is $\forall s, V_\pi(s) = \mathbb{E}[G|s_0 = s]$
Unknown model <=> "unknown strategy" <=> we don't know the rewards in advance (we need to learn *online*).
=> we cannot compute the expectation $\mathbb{E}$
=> instead of the value function, we will estimate the **value-action function** thanks to the Bellman equation:

$$\forall s, a, \qquad Q_\pi(s, a) = \mathbb{E}_\pi[r_0 + \gamma Q(s_1, a_1)|s_0 = s, a_0 = a]$$

=> for an initial state and a fixed action, we can compute the Q function
=> the value-action function can be seen as the expected (=> use of transition matrix (probabilities)) gain we get at a state when doing a specific action
Note: the policy π takes account only from next state s_1

We want to *control* the policy and find the optimal one: $\pi^*(a) = a^* \in argmax_a Q_*(s, a)$

The optimal Bellman equation becomes:

$$\forall s, a, \qquad Q_\pi(s, a) = \mathbb{E}_\pi[r_0 + \gamma max_{a'} Q(s_1, a')|s_0 = s, a_0 = a]$$

How to choose initial state a_0?
–> Pure exploitation: we find the best action knowing the current system $\pi(s) \leftarrow argmax_a Q(s, a)$
=> problem: Q is estimated, thus we have a chance to miss good actions
–> Pure exploration: $\pi(s) \leftarrow random$
=> problem: we can waste time on bad actions and thus have bad estimation quality
Solution: trade-off exploitation/exploitation (SARSA, Q-learning, ...).

SARSA

This algorithm is based on ϵ-greedy algorithm.

$$\pi(s) \leftarrow \begin{cases} a^* \in argmax_a Q(s, a) \text{ with probability } 1 - \epsilon \\ random \text{ with probability } \epsilon \end{cases} \qquad (3)$$

=> with a small probability (ϵ is usually lower than 5%), we do random actions. Otherwise we take the best action of the known ones.

Estimation is done through *TD-learning* (temporal differences - see Online Estimation):

$$\forall t, Q(s_t, a_t) \xleftarrow{\alpha} r_t + \gamma Q(s_{t+1}, a_{t+1})$$

Listing 7: SARSA

```
# action is identified with new state (after move) except teleportation (action = same state)

def sarsa(Q, model = model, alpha = 0.1, eps = 0.1, n_iter = 100):
    states = model.states
    terminals = model.terminals
    rewards = model.rewards
    gamma = model.gamma
    # random state (not terminal)
    state = np.random.choice(np.setdiff1d(np.arange(len(states)), terminals))
    # random action
    action = np.random.choice(Q[state].indices)
    new_state = action
    for t in range(n_iter):
        state_prev = state
        action_prev = action
        state = new_state
        if state in terminals: # if the new action gives terminal state
                # we start again from a new random state
                    state = np.random.choice(np.setdiff1d(np.arange(len(model.states)),
                                                                         terminals))
                    action = np.random.choice(Q[state].indices)
                    Q[state_prev, action_prev] =
                (1 - alpha) * Q[state_prev, action_prev] + alpha * rewards[action_prev]
                  # we removed the second term since state_prev was the last step
        else:
                  # we choose the best action that has the highest expected value-action
                    best_action = Q[state].indices[np.argmax(Q[state].data)]
                    if np.random.random() < eps:
                # with probability of epsilon, we choose a random action
                        action = np.random.choice(Q[state].indices)
                    else:
                        action = best_action
                    Q[state_prev, action_prev] =
                        (1 - alpha) * Q[state_prev, action_prev]
                        + alpha * (rewards[action_prev] + gamma * Q[state, action])
        new_state = action
    return Q
```

Q-learning

This algorithm is also based on ϵ-greedy algorithm.

$$\pi(s) \leftarrow \begin{cases} a^* \in argmax_a Q(s,a) \text{ with probability } 1-\epsilon \\ random \text{ with probability } \epsilon \end{cases} \tag{4}$$

Estimation: unlike SARSA, Q-learning aims at updating the estimator using the best action at each iteration:

$$\forall t, Q(s_t, a_t) \overset{\alpha}{\leftarrow} r_t + \gamma max_a Q(s_{t+1}, a)$$

The only modification from SARSA is the following line:

Listing 8: Q-Learning

```
Q[state_prev, action_prev] = (1 - alpha) * Q[state_prev, action_prev]
                             + alpha * (rewards[action_prev] + gamma * np.max(Q[state].data))
```