1.1. Linear Models

The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the features. In mathematical notation, if $\hat{y}$ is the predicted value.

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

Across the module, we designate the vector $w = (w_1, ..., w_p)$ as `coef_` and w_0 as `intercept_`.

To perform classification with generalized linear models, see Logistic regression.

1.1.1. Ordinary Least Squares

`LinearRegression` fits a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_{w} || X w - y||_2^2$$

1.1.1.2. Ordinary Least Squares Complexity

The least squares solution is computed using the singular value decomposition of X. If X is a matrix of shape `(n_samples, n_features)` this method has a cost of $O(n_{\text{samples}} n_{\text{features}}^2)$, assuming that $n_{\text{samples}} \geq n_{\text{features}}$.

1.1.2. Ridge regression and classification

1.1.2.1. Regression

`Ridge` regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of the coefficients. The ridge coefficients minimize a penalized residual sum of squares:

$$\min_{w} || X w - y||_2^2 + \alpha ||w||_2^2$$

The complexity parameter $\alpha \geq 0$ controls the amount of shrinkage: the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.

1.1.3. Lasso

The `Lasso` is a linear model that estimates sparse coefficients. It is useful in some contexts due to its tendency to prefer solutions with fewer non-zero coefficients, effectively reducing the number of features upon which the given solution is dependent. For this reason, Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero coefficients (see Compressive sensing: tomography reconstruction with L1 prior (Lasso)).

Mathematically, it consists of a linear model with an added regularization term. The objective function to minimize is:

$$\min_{w} { \frac{1}{2n_{\text{samples}}} ||X w - y||_2 ^ 2 + \alpha ||w||_1}$$

The lasso estimate thus solves the minimization of the least-squares penalty with $\alpha ||w||_1$ added, where α is a constant and $||w||_1$ is the ℓ_1-norm of the coefficient vector.

The implementation in the class `Lasso` uses coordinate descent as the algorithm to fit the coefficients. See Least Angle Regression for another implementation:

1.1.5. Elastic-Net

`ElasticNet` is a linear regression model trained with both ℓ_1 and ℓ_2-norm regularization of the coefficients. This combination allows for learning a sparse model where few of the weights are non-zero like `Lasso`, while still maintaining the regularization properties of `Ridge`. We control the convex combination of ℓ_1 and ℓ_2 using the `l1_ratio` parameter.

Elastic-net is useful when there are multiple features that are correlated with one another. Lasso is likely to pick one of these at random, while elastic-net is likely to pick both.

A practical advantage of trading-off between Lasso and Ridge is that it allows Elastic-Net to inherit some of Ridge's stability under rotation.

The objective function to minimize is in this case

$$\min_{w} \frac{1}{2n_{\text{samples}}} ||Xw - y||_2^2 + \alpha\rho||w||_1 + \frac{\alpha(1-\rho)}{2}||w||_2^2$$

- **The learning algorithm finds the parameters that optimize an <u>objective function</u>, typically to minimize some kind of <u>loss function</u> of the predicted target values *vs.* actual target values.**

Ridge Regression

- **Ridge regression learns *w*, *b* using the same least-squares criterion but adds a penalty for large variations in *w* parameters**

$$RSS_{RIDGE}(\boldsymbol{w}, b) = \sum_{\{i=1\}}^{N} (y_i - (\boldsymbol{w} \cdot \boldsymbol{x}_i + b))^2 \boxed{+ \alpha \sum_{\{j=1\}}^{p} w_j^2}$$

- **Once the parameters are learned, the ridge regression <u>prediction</u> formula is the <u>same</u> as ordinary least-squares.**
- **The addition of a parameter penalty is called <u>regularization</u>. Regularization prevents overfitting by restricting the model, typically to reduce its complexity.**
- **Ridge regression uses <u>L2 regularization</u>: minimize sum of squares of *w* entries**
- **The influence of the regularization term is controlled by the α parameter.**
- **Higher alpha means more regularization and simpler models.**

Lasso regression is another form of regularized linear regression that uses an L1 regularization penalty for training (instead of ridge's L2 penalty)

- **L1 penalty: Minimize the sum of the absolute values of the coefficients**

$$RSS_{LASSO}(\mathbf{w},b) = \sum_{\{i=1\}}^{N} (y_i - (\mathbf{w}\cdot\mathbf{x}_i + b))^2 + \boxed{\alpha \sum_{\{j=1\}}^{p} |w_j|}$$

- **This has the effect of setting parameter weights in *w* to zero for the least influential variables. This is called a sparse solution: a kind of feature selection**
- **The parameter α controls amount of L1 regularization (default = 1.0).**
- **The prediction formula is the same as ordinary least-squares.**
- **When to use ridge vs lasso regression:**
  - *Many small/medium sized effects: use ridge.*
  - *Only a few variables with medium/large effect: use lasso.*

2.2.1 Update Equations

The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

where the hypothesis $h_\theta(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Recall that the parameters of your model are the θ values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \quad (\text{simultaneously update } \theta_j \text{ for all } j)$$

With each step of gradient descent, your parameters j come closer to the optimal values that will achieve the lowest cost $J(\theta)$.

3.3 Normal Equations

In the lecture videos, you learned that the closed-form solution to linear regression is

$$\theta = (X^T X)^{-1} X^T \vec{y}$$

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no "loop until convergence" like in gradient descent.

1.1.11. Logistic regression

Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

Logistic regression is implemented in `LogisticRegression`. This implementation can fit binary, One-vs-Rest, or multinomial logistic regression with optional ℓ_1, ℓ_2 or Elastic-Net regularization.

Note: Regularization is applied by default, which is common in machine learning but not in statistics. Another advantage of regularization is that it improves numerical stability. No regularization amounts to setting C to a very high value.

Sigmoid

Logistic Regression Model

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$h_\theta(x) = g(\theta^T x),$$

$$X\theta = \begin{bmatrix} -(x^{(1)})^T\theta & - \\ -(x^{(2)})^T\theta & - \\ \vdots & \\ -(x^{(m)})^T\theta & - \end{bmatrix} = \begin{bmatrix} -\ \theta^T(x^{(1)}) & - \\ -\ \theta^T(x^{(2)}) & - \\ \vdots & \\ -\ \theta^T(x^{(m)}) & - \end{bmatrix}$$

Cost function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2,$$

Gradient

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \qquad \text{for } j = 0,$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for } j \geq 1,$$

To vectorize this operation over the dataset, we start by writing out all the partial derivatives explicitly for all θ_j,

$$\begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m}\begin{bmatrix} \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)} \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_1^{(i)} \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_2^{(i)} \\ \vdots \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_n^{(i)} \end{bmatrix} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x^{(i)} = \frac{1}{m}X^T(h_\theta(x) - y) \quad (1)$$

```python
def gradient_descent(X, y, theta, lambda_, eps, alpha, max_iter): # alpha is learning rate
  losses = []
  i = 0
  print("Iteration: Cost")

  while(i < max_iter):
    i += 1
    grad = gradient(X, y, theta, lambda_)
    theta -= alpha * grad
    loss = cost(X, y, theta, lambda_)
    if (i % 1000 == 0):
      print("{}: {:.8f}".format(i, loss))

    len_losses = len(losses)
    if (len_losses == 0):
      diff = np.abs(loss)
    else :
      diff = np.abs(losses[len_losses-1] - loss)

    losses.append(loss)
    if(diff < eps):
      return theta, losses

  return theta, losses
```

Softmax

(5 points) Prove that softmax is invariant to constant offsets in the input, that is, for any input vector $\boldsymbol{x}$ and any constant c,

$$\mathrm{softmax}(\boldsymbol{x}) = \mathrm{softmax}(\boldsymbol{x} + c)$$

where $\boldsymbol{x} + c$ means adding the constant c to every dimension of $\boldsymbol{x}$. Remember that

$$softmax(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}} \tag{1}$$

Note: In practice, we make use of this property and choose $c = -\max_i x_i$ when computing softmax probabilities for numerical stability (i.e., subtracting its maximum element from all elements of $\boldsymbol{x}$).

Regularized Cost Function

$$\text{Cost} = -\frac{1}{m}\left\{\left[\sum_{i=1}^{m}\sum_{j=1}^{K} 1\{y_i=j\}\log\underbrace{\frac{e^{\theta_j^T x_i}}{\sum_{l=1}^{K} e^{\theta_l^T x_i}}}_{\text{softmax}}\right] + \lambda\cdot\sum_{i=1}^{K}\sum_{j=1}^{n}\theta_{ij}^2\right\}$$

m: number of samples n: number of features

k: number of classes

Gradient with L2 Regularization

$$\frac{\partial\, \text{cost}}{\partial \theta_j} = -\frac{1}{m}\left\{\left[\sum_{i=1}^{m} x_i\left(1\{y_i=j\} - p(y_i=j\mid x_i;\theta)\right)\right] - \lambda\cdot\theta_j\right\}$$

for $j \geq 1$.

$$\frac{\partial\, \text{Cost}}{\partial \theta} = -\frac{1}{m}\left[(y - \underbrace{p}_{\text{softmax}})^T X + \lambda\theta\right]$$

We should not regularize the θ_0

1.4. Support Vector Machines

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection.

The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

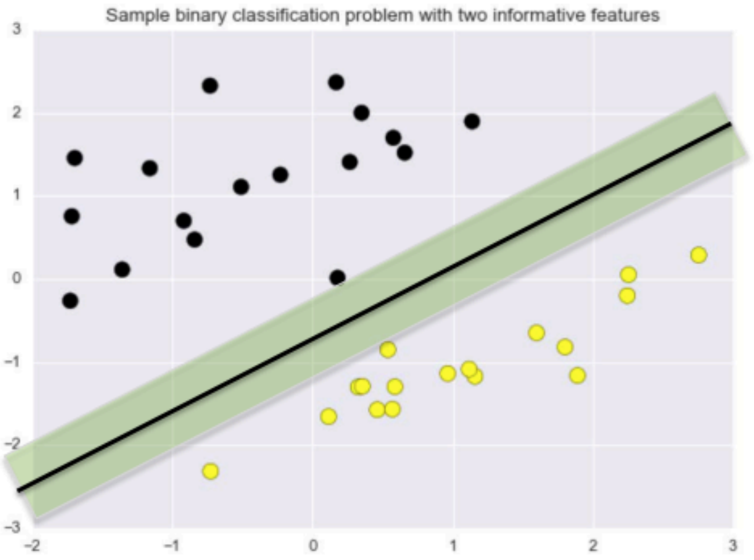
The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see Scores and probabilities, below).

Maximum margin linear classifier: Linear Support Vector Machines

$$x \xrightarrow{\quad} \boxed{f} \xrightarrow{\quad} \hat{y}$$

$$f(x, w, b) = sign(w \circ x + b)$$



Maximum margin classifier

The linear classifier with maximum margin is a linear Support Vector Machine (LSVM).

Given training vectors $x_i \in \mathbb{R}^p$, i=1,..., n, in two classes, and a vector $y \in \{1, -1\}^n$, our goal is to find $w \in \mathbb{R}^p$ and $b \in \mathbb{R}$ such that the prediction given by $\text{sign}(w^T\phi(x) + b)$ is correct for most samples.

SVC solves the following primal problem:

$$\begin{aligned} \min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i \\ \text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i, \\ \zeta_i \geq 0, i = 1, \ldots, n \end{aligned}$$

Intuitively, we're trying to maximize the margin (by minimizing $||w||^2 = w^T w$), while incurring a penalty when a sample is misclassified or within the margin boundary. Ideally, the value $y_i(w^T\phi(x_i) + b)$ would be ≥ 1 for all samples, which indicates a perfect prediction. But problems are usually not always perfectly separable with a hyperplane, so we allow some samples to be at a distance ζ_i from their correct margin boundary. The penalty term `C` controls the strength of this penalty, and as a result, acts as an inverse regularization parameter (see note below).

The dual problem to the primal is

$$\begin{aligned} \min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha \\ \text{subject to } y^T \alpha = 0 \\ 0 \leq \alpha_i \leq C, i = 1, \ldots, n \end{aligned}$$

where e is the vector of all ones, and Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv y_i y_j K(x_i, x_j)$, where $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. The terms α_i are called the dual coefficients, and they are upper-bounded by C. This dual representation highlights the fact that training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ: see kernel trick.

Once the optimization problem is solved, the output of decision_function for a given sample x becomes:

$$\sum_{i \in SV} y_i \alpha_i K(x_i, x) + b,$$

and the predicted class correspond to its sign. We only need to sum over the support vectors (i.e. the samples that lie within the margin) because the dual coefficients α_i are zero for the other samples.

These parameters can be accessed through the attributes `dual_coef_` which holds the product $y_i \alpha_i$, `support_vectors_` which holds the support vectors, and `intercept_` which holds the independent term b

Note: While SVM models derived from libsvm and liblinear use `C` as regularization parameter, most other estimators use `alpha`. The exact equivalence between the amount of regularization of two models depends on the exact objective function optimized by the model. For example, when the estimator used is **`Ridge`** regression, the relation between them is given as $C = \frac{1}{alpha}$.

Regularization for SVMs: the C parameter

- **The strength of regularization is determined by C**
- **Larger values of C: less regularization**
  - *Fit the training data as well as possible*
  - *Each individual data point is important to classify correctly*
- **Smaller values of C: more regularization**
  - *More tolerant of errors on individual data points*

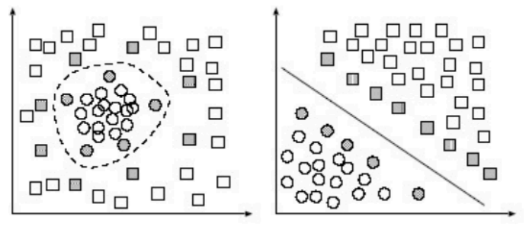
1.4.6. Kernel functions

The *kernel function* can be any of the following:

- linear: $\langle x, x' \rangle$.
- polynomial: $(\gamma \langle x, x' \rangle + r)^d$, where d is specified by parameter `degree`, r by `coef0`.
- rbf: $\exp(-\gamma \|x - x'\|^2)$, where γ is specified by parameter `gamma`, must be greater than 0.
- sigmoid $\tanh(\gamma \langle x, x' \rangle + r)$, where r is specified by `coef0`.

Radial Basis Function Kernel

$$K(x, x') = \exp\left[-\gamma \cdot \|x - x'\|^2\right]$$



A kernel is a similarity measure (modified dot product) between data points

Résoudre le problème de maximisation :

$$\boxed{\begin{array}{l} \underset{\lambda_1,\dots,\lambda_N}{\arg\max} \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{N} C_i C_j \lambda_i \lambda_j K_{ij} \\ \text{s.t. } \forall\, i \le i \le N.\quad \lambda_i \geq 0. \\ \qquad \sum_{i=1}^{N} C_i \lambda_i = 0. \end{array}}$$

⇒ Transformer le problème de maximisation vers le problème de minimisation

$$\boxed{\begin{array}{l} \underset{\lambda_1,\dots\lambda_N}{\arg\min} \ \frac{1}{2} \sum_{i,j=1}^{N} C_i C_j \lambda_i \lambda_j K_{ij} - \sum_{i=1}^{N} \lambda_i \\ \text{S.t. } \forall\ i \le i \le N \qquad -\lambda_i \leq 0 \\ \qquad \sum_{i=1}^{N} C_i \lambda_i = 0 \end{array}}$$

$$\min \frac{1}{2} \sum_{i,j=1}^{N} C_i C_j \lambda_i \lambda_j k_{ij} - \sum_{i=1}^{N} \lambda_i$$

$$\Rightarrow \min \frac{1}{2} \underbrace{\sum_{i=1}^{N} \lambda_i}_{\lambda^T} \sum_{i,j=1}^{N} C_i C_j k_{ij} \underbrace{\sum_{j=1}^{N} \lambda_j}_{\lambda} - \sum_{i=1}^{N} \lambda_i$$

Linear Models: Pros and Cons

Pros:

- **Simple and easy to train.**
- **Fast prediction.**
- **Scales well to very large datasets.**
- **Works well with sparse data.**
- **Reasons for prediction are relatively easy to interpret.**

Cons:

- **For lower-dimensional data, other models may have superior generalization performance.**
- **For classification, data may not be linearly separable (more on this in SVMs with non-linear kernels)**

1.9. Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n, :

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)}$$

Using the naive conditional independence assumption that

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y),$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule:

$$\begin{aligned} P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y) \\ \Downarrow \\ \hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y), \end{aligned}$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

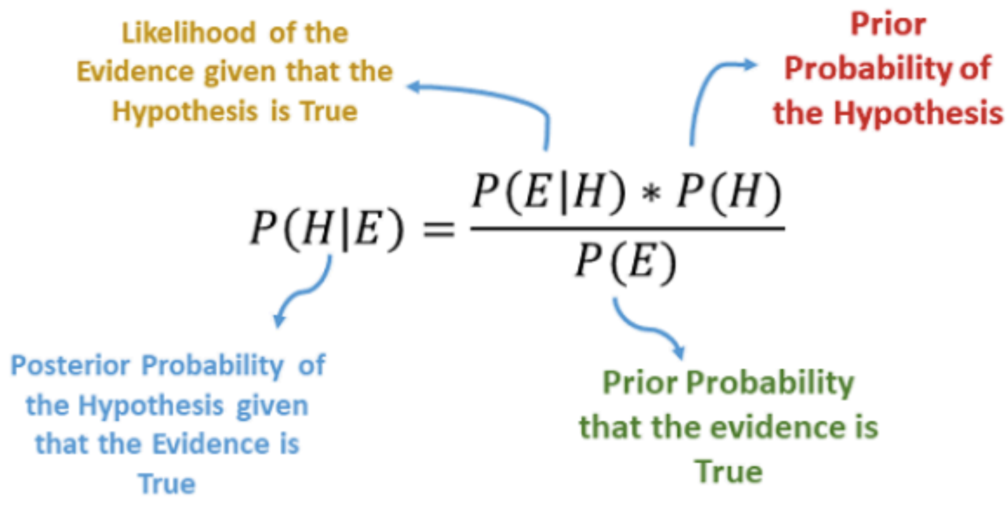
In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.)

1.9.1. Gaussian Naive Bayes

`GaussianNB` implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma^2_y}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma^2_y}\right)$$

The parameters σ_y and μ_y are estimated using maximum likelihood.


Likelihood of the Evidence given that the Hypothesis is True
Prior Probability of the Hypothesis
$P(H|E) = \frac{P(E|H) * P(H)}{P(E)}$
Posterior Probability of the Hypothesis given that the Evidence is True
Prior Probability that the evidence is True

can then use Bayes rule to derive the posterior distribution on y given x:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}.$$

Here, the denominator is given by $p(x) = p(x|y = 1)p(y = 1) + p(x|y = 0)p(y = 0)$ (you should be able to verify that this is true from the standard properties of probabilities), and thus can also be expressed in terms of the quantities $p(x|y)$ and $p(y)$ that we've learned. Actually, if were calculating $p(y|x)$ in order to make a prediction, then we don't actually need to calculate the denominator, since

$$\begin{aligned} \arg\max_y p(y|x) &= \arg\max_y \frac{p(x|y)p(y)}{p(x)} \\ &= \arg\max_y p(x|y)p(y). \end{aligned}$$

MLE: Maximum likelihood estimation

$$\begin{aligned}
\hat{\theta}_{MLE} &= \underset{\theta}{\operatorname{argmax}}\ p(D|\theta) \\
&= \underset{\theta}{\operatorname{argmax}}\ \log\left(p(D|\theta)\right) \rightarrow \text{Log-vraisemblance } \mathcal{L}(\theta) = \log p(D|\theta) \\
&= \underset{\theta}{\operatorname{argmax}}\ \log\left(\prod_{i=1}^{m} p(x_i|\theta)\right) \\
&= \underset{\theta}{\operatorname{argmax}}\ \sum_{i=1}^{m} \log\left(p(x_i|\theta)\right) \\
&= \underset{\theta}{\operatorname{argmin}}\ \frac{1}{m}\sum_{i=1}^{m} -\log\left(p(x_i|\theta)\right) \\
&= \underset{\theta}{\operatorname{argmin}}\ \frac{1}{m}\sum_{i=1}^{m} L\left(f_\theta(x_i)\right)
\end{aligned}$$

réécriture sous la forme de minimisation du risque empirique

où $f_\theta = p(x_i|\theta)$

et $L(f_\theta(x_i)) = -\log(f_\theta(x_i))$

↳ log-vraisemblance négative

MAP: maximum a posteriori

MAXIMUM A POSTERIORI :

$$\hat{\theta}_{MAP} = \underset{\theta}{\operatorname{argmax}}\ p(\theta|D)$$

$$= \underset{\theta}{\operatorname{argmax}}\ \frac{\overset{\text{VRAISEMBLANCE}}{p(D|\theta)}\ \overset{\text{PRIOR}}{p(\theta)}}{\underset{\text{independant de } \theta}{p(D)}} \quad \text{(Règle de Bayes)}$$

$$= \underset{\theta}{\operatorname{argmax}}\ p(D|\theta)\, p(\theta) \quad \text{(car } p(D) \text{ independant de } \theta)$$

Comparing both MLE and MAP equation, the only thing differs is the inclusion of prior $P(\theta)$ in MAP, otherwise they are identical. What it means is that, the likelihood is now weighted with some weight coming from the prior.

Let's consider what if we use the simplest prior in our MAP estimation, i.e. uniform prior. This means, we assign equal weights everywhere, on all possible values of the θ. The implication is that the likelihood equivalently weighted by some constants. Being constant, we could be ignored from our MAP equation, as it will not contribute to the maximization.

Let's be more concrete, let's say we could assign six possible values into θ. Now, our prior $P(\theta)$ is $\frac{1}{6}$ everywhere in the distribution. And consequently, we could ignore that constant in our MAP estimation.

$$\begin{aligned}\theta_{MAP} &= \arg\max_{\theta} \sum_i \log P(x_i|\theta) + \log P(\theta) \\ &= \arg\max_{\theta} \sum_i \log P(x_i|\theta) + const \\ &= \arg\max_{\theta} \sum_i \log P(x_i|\theta) \\ &= \theta_{MLE}\end{aligned}$$

We are back at MLE equation again!

If we use different prior, say, a Gaussian, then our prior is not constant anymore, as depending on the region of the distribution, the probability is high or low, never always the same.

What we could conclude then, is that MLE is a special case of MAP, where the prior is uniform!

Le classifieur de Bayes estime $p(Y|X)$ en utilisant le théorème de Bayes :

$$\underset{c=1\dots m}{\operatorname{argmax}} \underbrace{p(Y=c|X=x)}_{\text{POSTERIOR}} = \underset{c}{\operatorname{argmax}} \frac{p(X=x|Y=c)\,p(Y=c)}{\underbrace{p(X=x)}_{\text{independant de } c}}$$

$$= \underset{c}{\operatorname{argmax}} \underbrace{p(X=x|Y=c)}_{\text{VRAISEMBLANCE}} \underbrace{p(Y=c)}_{\text{PRIOR}}$$

- On doit estimer $p(X|Y)$ et $p(Y)$ à partir de D.
  - Pour chaque $c=1\dots m$, soit $D_c = \{(x_i,y_i)\in D \mid y_i = c\}$
    ↳ Jeu de données IID tiré suivant $p(X|Y=c)$
  - Estimer $\hat{p}(Y=c) = \dfrac{|D_c|}{m}$
  - Utiliser une méthode d'estimation de densité pour estimer $p(X|Y=c)$ à partir de $D_c \rightarrow \hat{p}(X|Y=c)$

Règle de classification :

$$f(x) = \underset{c=1\dots m}{\operatorname{argmax}} \hat{p}(X=x|Y=c)\,\hat{p}(Y=c)$$

2) Classifieur de Bayes naïf (Naive Bayes)

On fait l'hypothèse que les composantes de $X\in\mathbb{R}^d$ sont indépendantes sachant Y :

$$p(X|Y=c) = p([X]_1,[X]_2,\dots,[X]_d \mid Y=c)$$
$$= p([X]_1|Y=c)\,p([X]_2|Y=c)\cdots p([X]_d|Y=c)$$

⚠ C'est une hypothèse très forte ⚠

↳ CAPACITÉ DE BAYES NAÏF < CAPACITÉ DU CLASSIFIEUR DE BAYES

↳ Chaque $p([X]_i|Y=c)$ est une distribution scalaire
$\Rightarrow$ facile à estimer

Naïve Bayes Classifiers: a simple, probabilistic classifier family

- These classifiers are called 'Naïve' because they assume that features are conditionally independent, given the class.
- In other words: they assume that, for all instances of a given class, the features have little/no correlation with each other.
- Highly efficient learning and prediction.
- But generalization performance may worse than more sophisticated learning methods.
- Can be competitive for some tasks.

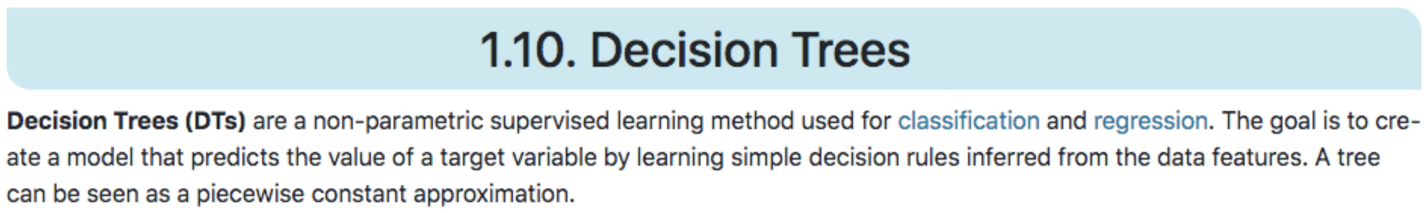
Naïve Bayes classifiers: Pros and Cons

Pros:

- Easy to understand
- Simple, efficient parameter estimation
- Works well with high-dimensional data
- Often useful as a baseline comparison against more sophisticated methods

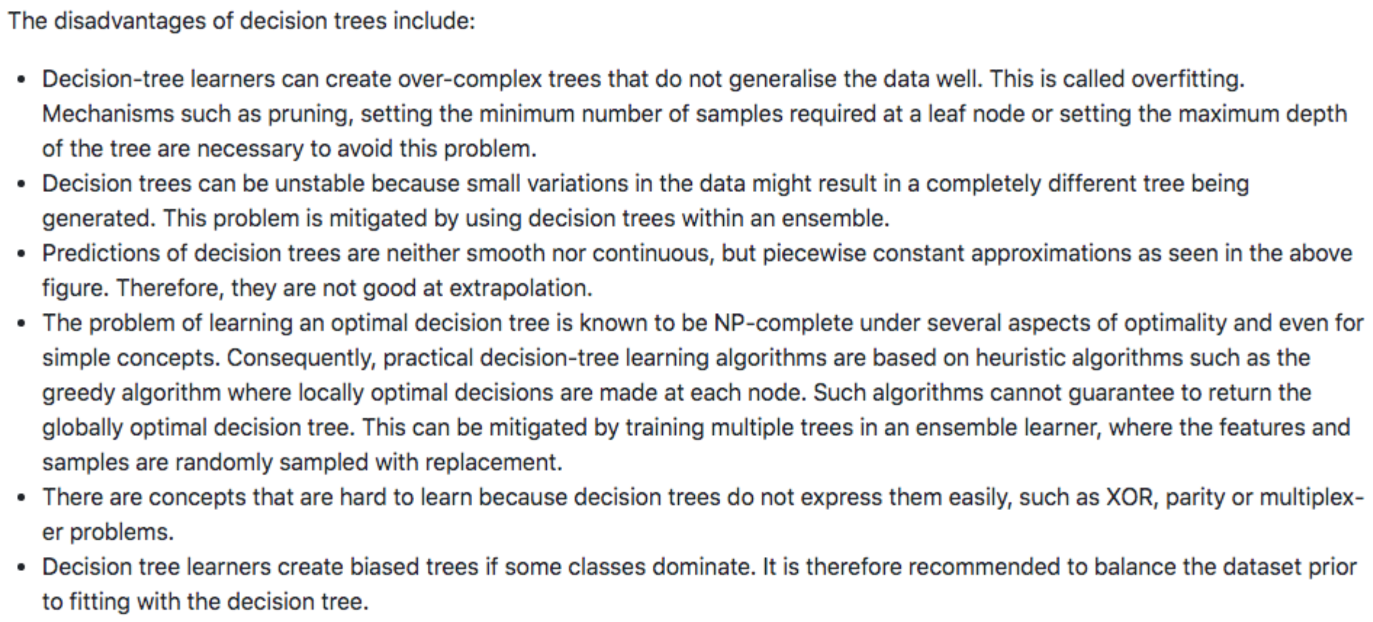
Cons:

- Assumption that features are conditionally independent given the class is not realistic.
- As a result, other classifier types often have better generalization performance.
- Their confidence estimates for predictions are not very accurate.



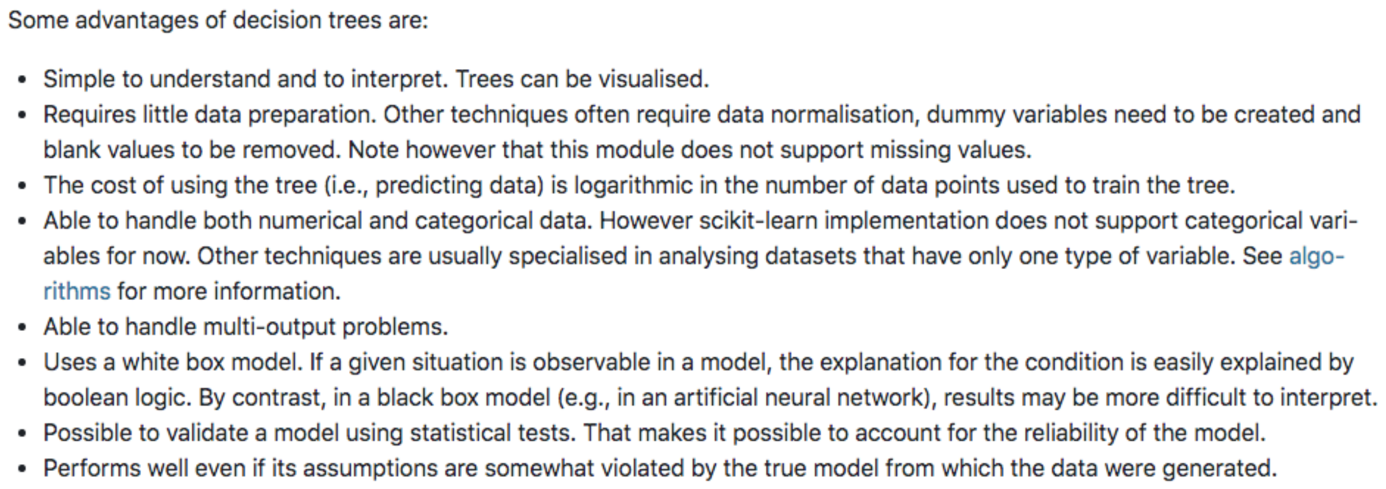
1.10. Decision Trees

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.



The disadvantages of decision trees include:

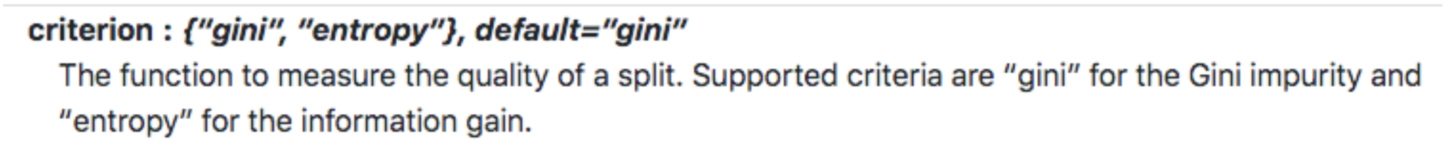
- Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximations as seen in the above figure. Therefore, they are not good at extrapolation.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.



Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualised.
- Requires little data preparation. Other techniques often require data normalisation, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. However scikit-learn implementation does not support categorical variables for now. Other techniques are usually specialised in analysing datasets that have only one type of variable. See algorithms for more information.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

Réprésente une partition de l'espace (partition of space of training dataset).



criterion : *{"gini", "entropy"}, default="gini"*

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

maximize the information gain.

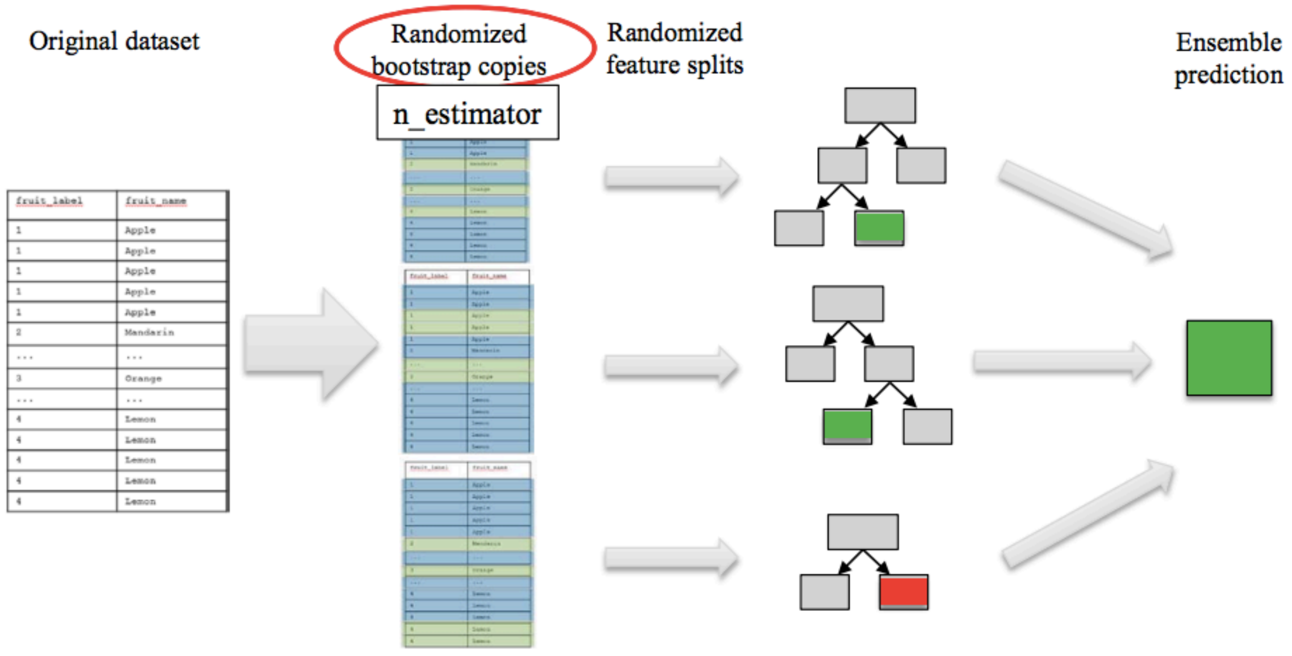
- **`max_depth`: controls maximum depth (number of split points). Most common way to reduce tree complexity and overfitting.**

- **`min_samples_leaf`: threshold for the minimum # of data instances a leaf can have to avoid further splitting.**

- **`max_leaf_nodes`: limits total number of leaves in the tree.**

- **In practice, adjusting only one of these (e.g. `max_depth`) is enough to reduce overfitting.**

Random Forest Process



1.11.2.1. Random Forests

In random forests (see `RandomForestClassifier` and `RandomForestRegressor` classes), each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.

Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size `max_features`. (See the parameter tuning guidelines for more details).

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit. The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant hence yielding an overall better model.

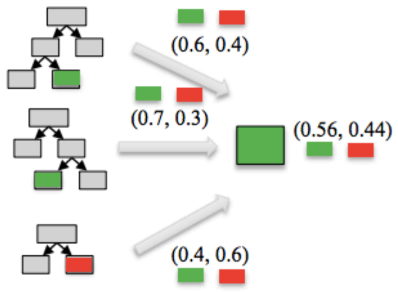
In contrast to the original publication [B2001], the scikit-learn implementation combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

Prediction Using Random Forests

1. Make a prediction for every tree in the forest.

2. Combine individual predictions

- *Regression: mean of individual tree predictions.*
- *Classification:*
  - *Each tree gives probability for each class.*
  - *Probabilities averaged across trees.*
  - *Predict the class with highest probability.*



Random Forest: Pros and Cons

Pros:

- **Widely used, excellent prediction performance on many problems.**
- **Doesn't require careful normalization of features or extensive parameter tuning.**
- **Like decision trees, handles a mixture of feature types.**
- **Easily parallelized across multiple CPUs.**

Cons:

- **The resulting models are often difficult for humans to interpret.**
- **Like decision trees, random forests may not be a good choice for very high-dimensional tasks (e.g. text classifiers) compared to fast, accurate linear models.**

Random Forests: RandomForestClassifier Key Parameters

- `n_estimators`**: number of trees to use in ensemble (default: 10).**
  - *Should be larger for larger datasets to reduce overfitting (but uses more computation).*
- `max_features`**: has a strong effect on performance. Influences the diversity of trees in the forest.**
  - *Default works well in practice, but adjusting may lead to some further gains.*
- `max_depth`**: controls the depth of each tree (default: None. Splits until all leaves are pure).**
- `n_jobs`**: How many cores to use in parallel during training.**

- **Choose a fixed setting for the random_state parameter if you need reproducible results.**

1.6. Nearest Neighbors

`sklearn.neighbors` provides functionality for unsupervised and supervised neighbors-based learning methods. Unsupervised nearest neighbors is the foundation of many other learning methods, notably manifold learning and spectral clustering. Supervised neighbors-based learning comes in two flavors: classification for data with discrete labels, and regression for data with continuous labels.

The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbors-based methods are known as *non-generalizing* machine learning methods, since they simply "remember" all of its training data (possibly transformed into a fast indexing structure such as a Ball Tree or KD Tree).

Despite its simplicity, nearest neighbors has been successful in a large number of classification and regression problems, including handwritten digits and satellite image scenes. Being a non-parametric method, it is often successful in classification situations where the decision boundary is very irregular.

2.5.1. Principal component analysis (PCA)

2.5.1.1. Exact PCA and probabilistic interpretation

PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance. In scikit-learn, `PCA` is implemented as a *transformer* object that learns n components in its `fit` method, and can be used on new data to project it on these components.

`sklearn.cluster`: Clustering

The `sklearn.cluster` module gathers popular unsupervised clustering algorithms.

User guide: See the Clustering and Biclustering sections for further details.

Classes

`cluster.AffinityPropagation`(*[, damping, ...])	Perform Affinity Propagation Clustering of data.
`cluster.AgglomerativeClustering`([...])	Agglomerative Clustering.
`cluster.Birch`(*[, threshold, ...])	Implements the BIRCH clustering algorithm.
`cluster.DBSCAN`([eps, min_samples, metric, ...])	Perform DBSCAN clustering from vector array or distance matrix.
`cluster.FeatureAgglomeration`([n_clusters, ...])	Agglomerate features.
`cluster.KMeans`([n_clusters, init, n_init, ...])	K-Means clustering.
`cluster.MiniBatchKMeans`([n_clusters, init, ...])	Mini-Batch K-Means clustering.
`cluster.MeanShift`(*[, bandwidth, seeds, ...])	Mean shift clustering using a flat kernel.
`cluster.OPTICS`(*[, min_samples, max_eps, ...])	Estimate clustering structure from vector array.
`cluster.SpectralClustering`([n_clusters, ...])	Apply clustering to a projection of the normalized Laplacian.
`cluster.SpectralBiclustering`([n_clusters, ...])	Spectral biclustering (Kluger, 2003).
`cluster.SpectralCoclustering`([n_clusters, ...])	Spectral Co-Clustering algorithm (Dhillon, 2001).

`sklearn.manifold`: Manifold Learning

The `sklearn.manifold` module implements data embedding techniques.

User guide: See the Manifold learning section for further details.

`manifold.Isomap`(*[, n_neighbors, …])	Isomap Embedding.
`manifold.LocallyLinearEmbedding`(*[, …])	Locally Linear Embedding.
`manifold.MDS`([n_components, metric, n_init, …])	Multidimensional scaling.
`manifold.SpectralEmbedding`([n_components, …])	Spectral embedding for non-linear dimensionality reduction.
`manifold.TSNE`([n_components, perplexity, …])	T-distributed Stochastic Neighbor Embedding.

`manifold.locally_linear_embedding`(X, *, …)	Perform a Locally Linear Embedding analysis on the data.
`manifold.smacof`(dissimilarities, *[, …])	Compute multidimensional scaling using the SMACOF algorithm.
`manifold.spectral_embedding`(adjacency, *[, …])	Project the sample on the first eigenvectors of the graph Laplacian.
`manifold.trustworthiness`(X, X_embedded, *[, …])	Expresses to what extent the local structure is retained.

Bagging - Bootstrap AGGregatING

- Learn n base learners in **parallel**, combine to reduce model variance
- Each base learner is trained on a bootstrap sample
  - Given a dataset of m examples, create a sample by randomly sampling m examples with replacement
  - Around $1 - 1/e \approx 63\,\%$ unique examples will be sampled use the out-of-bag examples for validation
- Combine learners by averaging the outputs (regression) or majority voting (classification)
- Random forest: bagging with decision trees
  - usually select random subset of features for each bootstrap sample

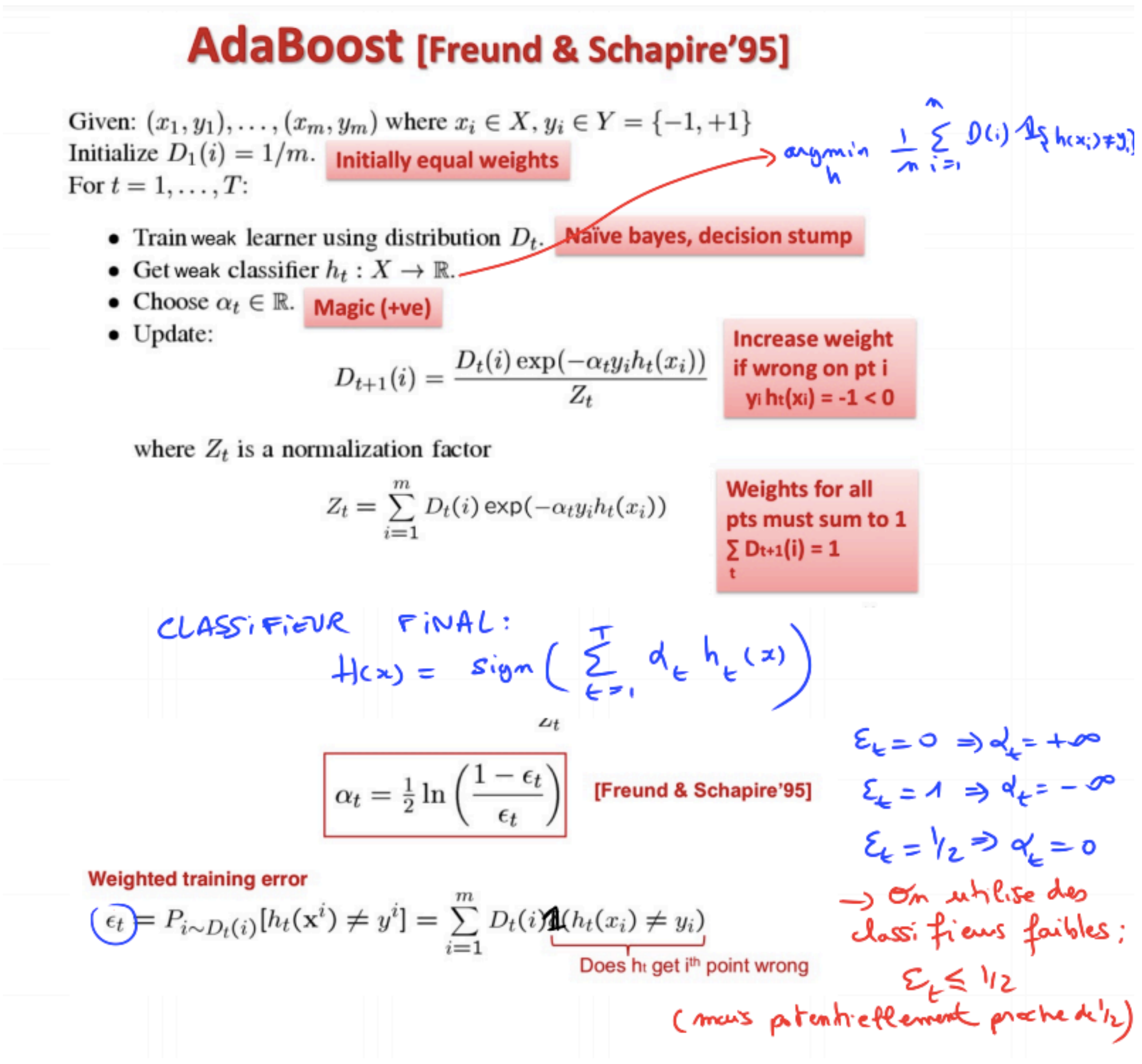
Boosting

- Learn n weak learners **sequentially**, combine to reduce model bias
- At step t, repeat:
  - Evaluate the existing learners' errors ϵ_t
  - Train a weak learner $\hat{f}_t$, focus on wrongly predicted examples
    - AdaBoost: Re-sample data according to ϵ_t
    - Gradient boosting: Train learner to predict ϵ_t
  - **Additively** combining existing weak learners with $\hat{f}_t$

1.11.3. AdaBoost

The module `sklearn.ensemble` includes the popular boosting algorithm AdaBoost, introduced in 1995 by Freund and Schapire [FS1995].

The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights w_1, w_2, ..., w_N to each of the training samples. Initially, those weights are all set to $w_i = 1/N$, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence [HTF].



AdaBoost [Freund & Schapire'95]

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X, y_i \in Y = \{-1, +1\}$
Initialize $D_1(i) = 1/m$. Initially equal weights
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t. Naïve bayes, decision stump
- Get weak classifier $h_t : X \to \mathbb{R}$. $\longrightarrow \underset{h}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} D(i) \mathbb{1}_{\{h(x_i) \neq y_i\}}$
- Choose $\alpha_t \in \mathbb{R}$. Magic (+ve)
- Update:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

Increase weight if wrong on pt i $y_i h_t(x_i) = -1 < 0$

where Z_t is a normalization factor

$$Z_t = \sum_{i=1}^{m} D_t(i) \exp(-\alpha_t y_i h_t(x_i))$$

Weights for all pts must sum to 1 $\sum_t D_{t+1}(i) = 1$

CLASSIFIEUR FINAL:

$$H(x) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x) \right)$$

$$\boxed{\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)}$$

[Freund & Schapire'95]

$\varepsilon_t = 0 \Rightarrow \alpha_t = +\infty$

$\varepsilon_t = 1 \Rightarrow \alpha_t = -\infty$

$\varepsilon_t = 1/2 \Rightarrow \alpha_t = 0$

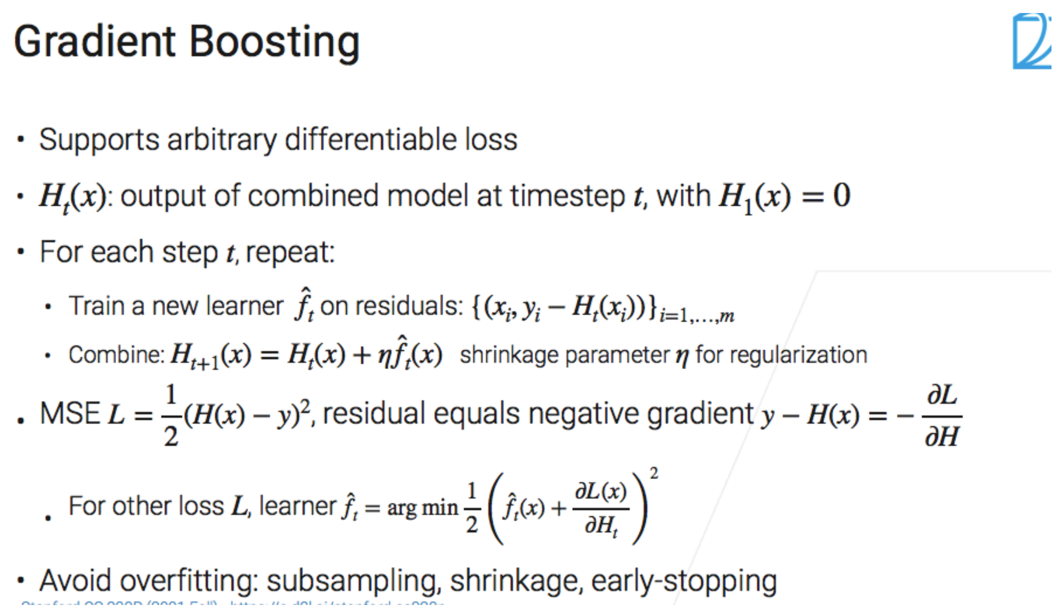
→ On utilise des classifieurs faibles ;

$\varepsilon_t \leq 1/2$

(mais potentiellement proche de 1/2)

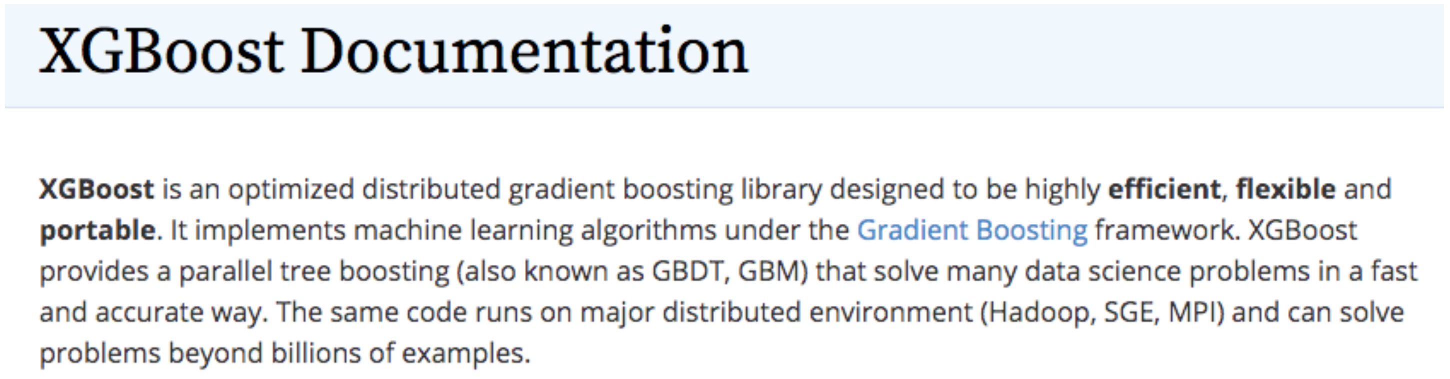
Weighted training error

$$\epsilon_t = P_{i \sim D_t(i)}[h_t(\mathbf{x}^i) \neq y^i] = \sum_{i=1}^{m} D_t(i) \underbrace{\mathbb{1}(h_t(x_i) \neq y_i)}_{\text{Does } h_t \text{ get } i^{th} \text{ point wrong}}$$



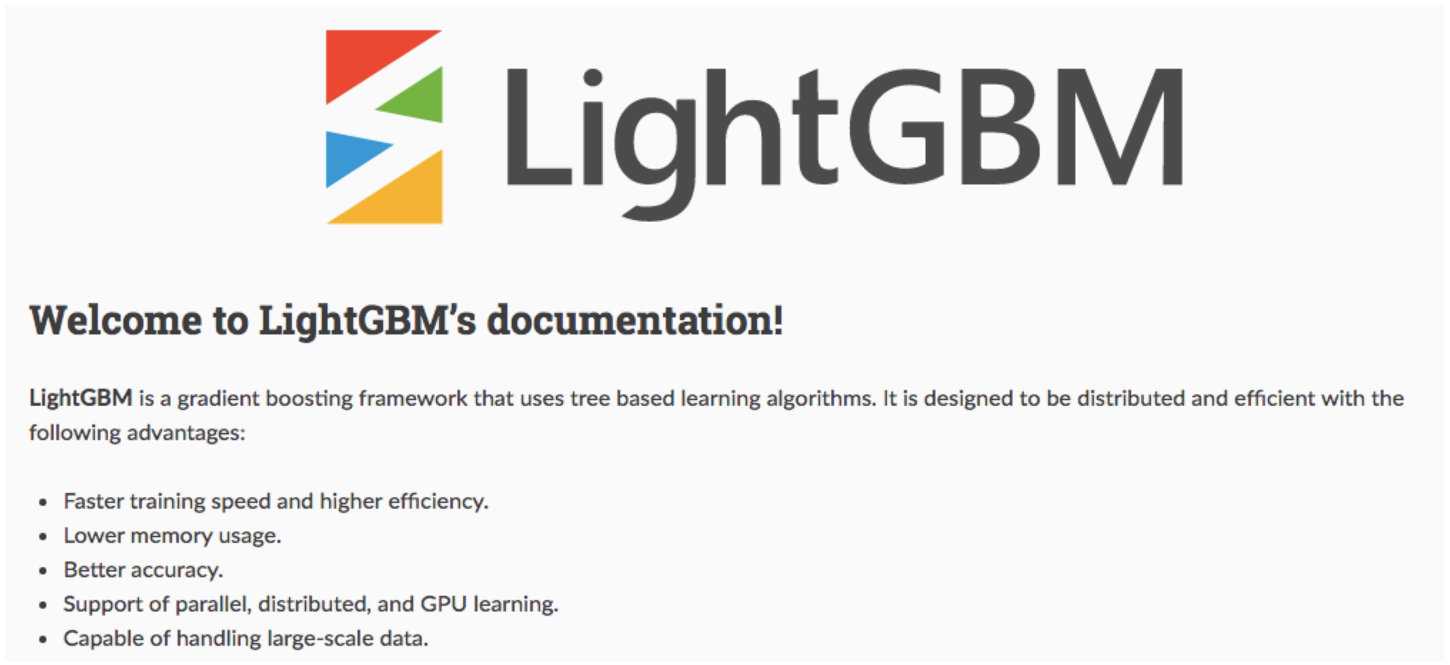
Gradient Boosting

- Supports arbitrary differentiable loss
- $H_t(x)$: output of combined model at timestep t, with $H_1(x) = 0$
- For each step t, repeat:
  - Train a new learner $\hat{f}_t$ on residuals: $\{(x_i, y_i - H_t(x_i))\}_{i=1,\ldots,m}$
  - Combine: $H_{t+1}(x) = H_t(x) + \eta \hat{f}_t(x)$ shrinkage parameter η for regularization
- MSE $L = \frac{1}{2}(H(x) - y)^2$, residual equals negative gradient $y - H(x) = -\frac{\partial L}{\partial H}$
  - For other loss L, learner $\hat{f}_t = \arg\min \frac{1}{2}\left(\hat{f}_t(x) + \frac{\partial L(x)}{\partial H_t}\right)^2$
- Avoid overfitting: subsampling, shrinkage, early-stopping



XGBoost Documentation

XGBoost is an optimized distributed gradient boosting library designed to be highly **efficient**, **flexible** and **portable**. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.



LightGBM

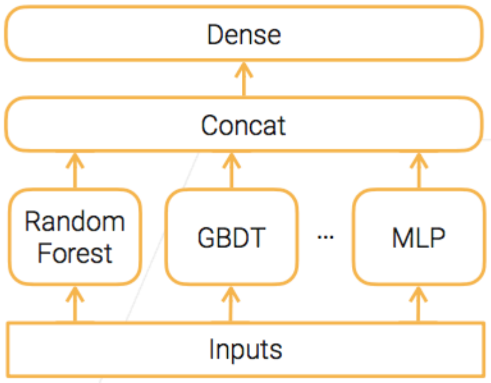
Welcome to LightGBM's documentation!

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages:

- Faster training speed and higher efficiency.
- Lower memory usage.
- Better accuracy.
- Support of parallel, distributed, and GPU learning.
- Capable of handling large-scale data.

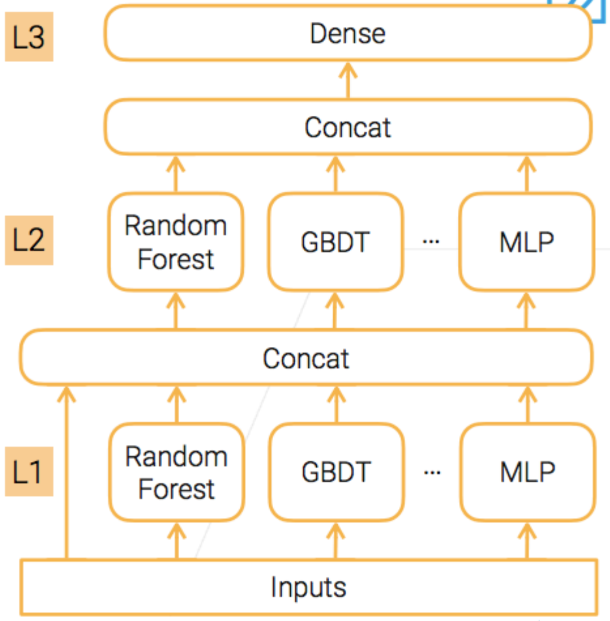
Stacking

- Combine multiple base learners to reduce variance
  - Base learners can be different model types
  - Linearly combine base learners outputs by learned parameters
- Widely used in competitions
- bagging VS stacking
  - Bagging: bootstrap samples to get diversity
  - Stacking: different types of models extract different features



Multi-layer Stacking

- Stacking base learners in multiple levels to reduce bias
  - Can use a different set of base learners at each level
- Upper levels (e.g. L2) are trained on the outputs of the level below (e.g. L1)
  - Concatenating original inputs helps



Overfitting in Multi-layer Stacking

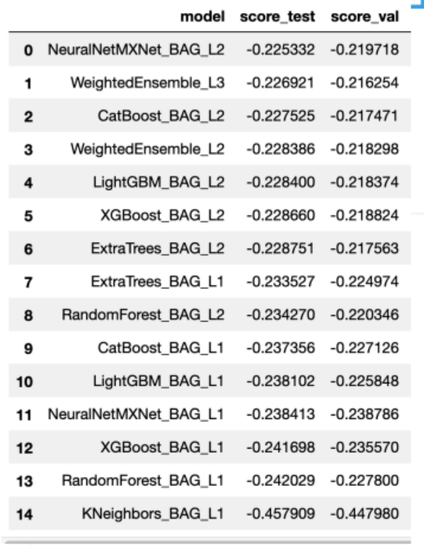
- Train leaners from different levels on different data to alleviate overfitting
  - Split training data into *A* and *B*, train L1 learners on *A*, run inference on *B* to generate training data for L2 learners
- Repeated k-fold bagging:
  - Train k models as in k-fold cross validation
  - Combine predictions of each model on out-of-fold data
  - Repeat step 1,2 by n times, average the n predictions of each example for the next level training

Multi-layer Stacking Results

- Use 1 additional staked level, with 5-fold repeated bagging
  - Error: 0.229 → 0.227
  - Training time: 39 sec → 207 sec (5x)

```python
from autogluon.tabular import TabularPredictor

predictor = TabularPredictor(label=label).fit(
    train, num_stack_levels=1, num_bag_folds=5)
```



	model	score_test	score_val
0	NeuralNetMXNet_BAG_L2	-0.225332	-0.219718
1	WeightedEnsemble_L3	-0.226921	-0.216254
2	CatBoost_BAG_L2	-0.227525	-0.217471
3	WeightedEnsemble_L2	-0.228386	-0.218298
4	LightGBM_BAG_L2	-0.228400	-0.218374
5	XGBoost_BAG_L2	-0.228660	-0.218824
6	ExtraTrees_BAG_L2	-0.228751	-0.217563
7	ExtraTrees_BAG_L1	-0.233527	-0.224974
8	RandomForest_BAG_L2	-0.234270	-0.220346
9	CatBoost_BAG_L1	-0.237356	-0.227126
10	LightGBM_BAG_L1	-0.238102	-0.225848
11	NeuralNetMXNet_BAG_L1	-0.238413	-0.238786
12	XGBoost_BAG_L1	-0.241698	-0.235570
13	RandomForest_BAG_L1	-0.242029	-0.227800
14	KNeighbors_BAG_L1	-0.457909	-0.447980

Reduce	Bias	Variance	Computation Cost	Parallelization
Bagging	-	Y	*n*	*n*
Boosting	Y	-	*n*	*1*
Stacking	-	Y	*n*	*n*
K-fold multi-level stacking	Y	Y	*nxlxk*	*nxk*

n: number of learners, *l*: number of levels, *k*: k-fold

Generalization, Overfitting, and Underfitting

- **Generalization ability refers to an algorithm's ability to give accurate predictions for new, previously unseen data.**
- **Assumptions:**
  - *Future unseen data (test set) will have the same properties as the current training sets.*
  - *Thus, models that are accurate on the training set are expected to be accurate on the test set.*
  - *But that may not happen if the trained model is tuned too specifically to the training set.*
- **Models that are too complex for the amount of training data available are said to overfit and are not likely to generalize well to new examples.**
- **Models that are too simple, that don't even do well on the training data, are said to underfit and also not likely to generalize well.**

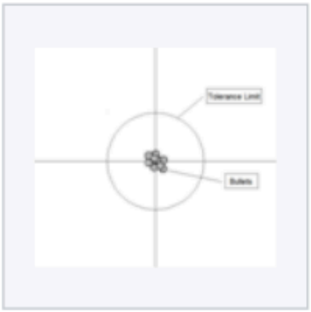
Bias–variance tradeoff

From Wikipedia, the free encyclopedia

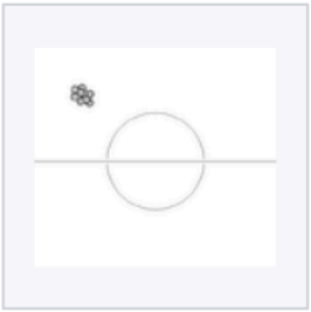
In statistics and machine learning, the **bias–variance tradeoff** is the property of a model that the variance of the parameter estimated across samples can be reduced by increasing the bias in the estimated parameters. The **bias–variance dilemma** or **bias–variance problem** is the conflict in trying to simultaneously minimize these two sources of error that prevent supervised learning algorithms from generalizing beyond their training set:[1][2]

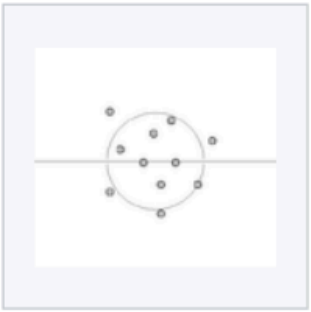
- The *bias* error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).
- The *variance* is an error from sensitivity to small fluctuations in the training set. High variance may result from an algorithm modeling the random noise in the training data (overfitting).

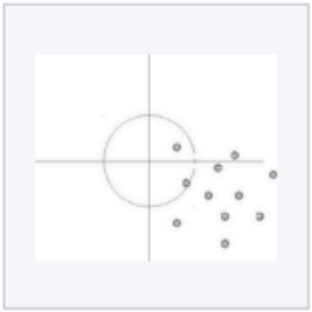
Motivation [edit]



bias low, variance low


bias high,
variance low:


bias low,
variance high:


bias high,
variance high:

Dilemme biais-variance

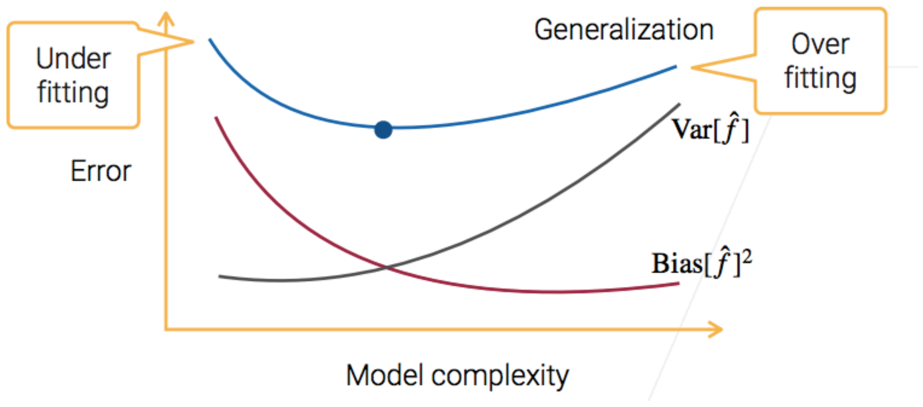
En statistique et en apprentissage automatique, le **dilemme (ou compromis) biais–variance** est le problème de minimiser simultanément deux sources d'erreurs qui empêchent les algorithmes d'apprentissage supervisé de généraliser au-delà de leur échantillon d'apprentissage :

- Le *biais* est l'erreur provenant d'hypothèses erronées dans l'algorithme d'apprentissage. Un biais élevé peut être lié à un algorithme qui manque de relations pertinentes entre les données en entrée et les sorties prévues (sous-apprentissage).
- La *variance* est l'erreur due à la sensibilité aux petites fluctuations de l'échantillon d'apprentissage. Une variance élevée peut entraîner un surapprentissage, c'est-à-dire modéliser le bruit aléatoire des données d'apprentissage plutôt que les sorties prévues.

Bias-Variance Tradeoff

MSE :mean squared error

$$\mathrm{E}_D\left[(y-\hat{f}_D(x))^2\right] = \mathrm{Bias}[\hat{f}_D]^2 + \mathrm{Var}[\hat{f}_D] + \epsilon^2$$



Reduce Bias & Variance

$$E_D\left[(y - \hat{f}_D(x))^2\right] = \text{Bias}[\hat{f}_D]^2 + \text{Var}[\hat{f}_D] + \epsilon^2$$

- Reduce bias
  - A more complex model
    - e.g. increase #layers, #hidden units of MLP
  - Boosting
  - Stacking
- Reduce variance
  - A simpler model
    - e.g. regularization
  - Bagging
  - Stacking
- Reduce σ^2
  - Improve data

Ensemble learning: train and combine multiple models to improve predictive performance

Binary prediction outcomes

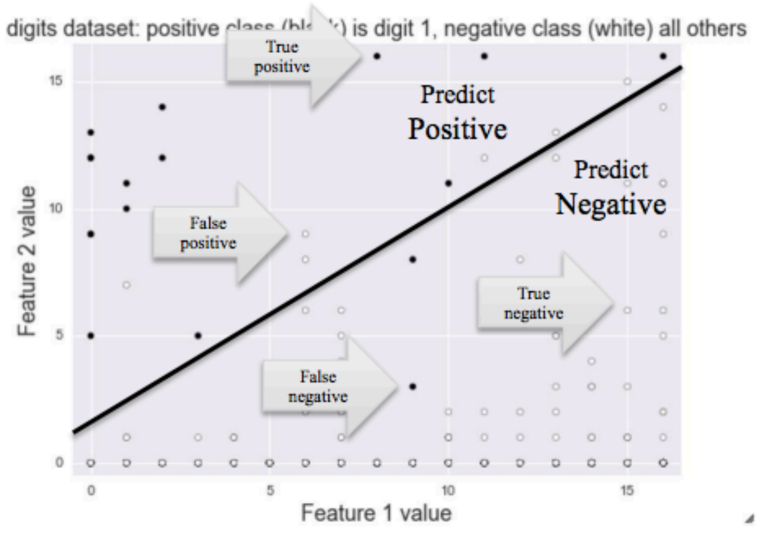
	Predicted negative	**Predicted** positive
True negative	TN	FP
True positive	FN	TP

Label 1 = positive class (class of interest)

Label 0 = negative class (everything else)

TP = true positive
FP = false positive (Type I error)
TN = true negative
FN = false negative (Type II error)

Visualization of Different Error Types



TN = 429	FP = 6
FN = 2	TP = 13

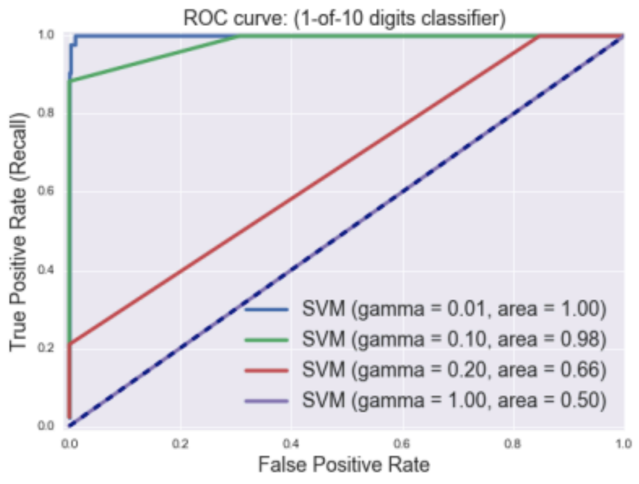
$$\text{Accuracy} = \frac{TN+TP}{TN+TP+FN+FP} \qquad \text{Recall} = \frac{TP}{TP+FN} \qquad \text{Precision} = \frac{TP}{TP+FP}$$

F1-score: combining precision & recall into a single number

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot TP}{2 \cdot TP + FN + FP}$$

Summarizing an ROC curve in one number: Area Under the Curve (AUC)

- AUC = 0 (worst) AUC = 1 (best)
- AUC can be interpreted as:
  1. The total area under the ROC curve.
  2. The probability that the classifier will assign a higher score to a randomly chosen positive example than to a randomly chosen negative example.
- Advantages:
  - Gives a single number for easy comparison.
  - Does not require specifying a decision threshold.
- Drawbacks:
  - As with other single-number metrics, AUC loses information, e.g. about tradeoffs and the shape of the ROC curve.
  - This may be a factor to consider when e.g. wanting to compare the performance of classifiers with overlapping ROC curves.



Macro-average:
- Each class has equal weight.

1. Compute metric within each class
2. Average resulting metrics across classes

Micro-average:
- Each instance has equal weight.
- Largest classes have most influence

1. Aggregrate outcomes across all classes
2. Compute metric with aggregate outcomes

Macro-Average vs Micro-Average

- If the classes have about the same number of instances, macro- and micro-average will be about the same.
- If some classes are much larger (more instances) than others, and you want to:
  - Weight your metric toward the largest ones, use micro-averaging.
  - Weight your metric toward the smallest ones, use macro-averaging.
- If the micro-average is much lower than the macro-average then examine the larger classes for poor metric performance.
- If the macro-average is much lower than the micro-average then examine the smaller classes for poor metric performance.