MODEL LIFECYCLE

BATALHA DE MODELOS II

DIEGO RODRIGUES DSC

INFNET

MODEL LIFECYCLE : BATALHA DE MODELOS II

PARTE 1 : TEORIA

- DATA PREPARATION
  - ANÁLISE DE COMPONENTES PRINCIPAIS
- MODELING
  - DISCRIMINANTES LINEARES E QUADRÁTICOS
  - REGRESSÃO LOGÍSTICA & REDES NEURAIS
  - ÁRVORE DE DECISÃO & RANDOM FORESTS
- EVALUATION
  - CURVA DE APRENDIZADO

Produzir Ação

CICLO DE VIDA DO MODELO

Baseado em Dados

AMBIENTE PYTHON

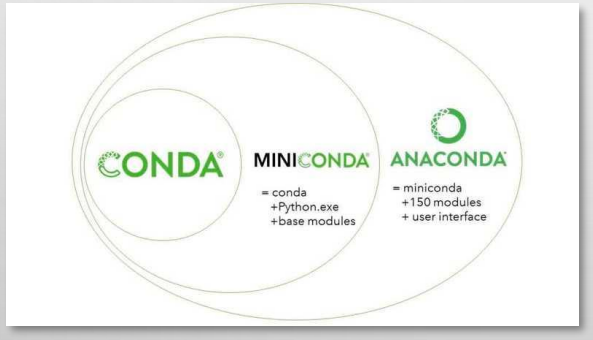
5. Visualização

6. Estimação e Inferência

7. Machine Learning

4. Variáveis Aleatórias

1. Editor de Código



2. Gestor de Ambiente

3. Ambiente Python do Projeto

3. Notebook Dinâmico

Cross Industry Standard Process for Data Mining - IBM

1) Requerimentos e Análise de Negócio

Entendimento do problema decisório, dados relacionados & revisão bibliográfica.

2) Preparação dos Dados

Entendimento das fontes de dados, dos tipos e elaboração da representação.

3) Modelagem

Análise Exploratória, Seleção de atributos e treinamento.

4) Avaliação

Seleção do melhor modelo.

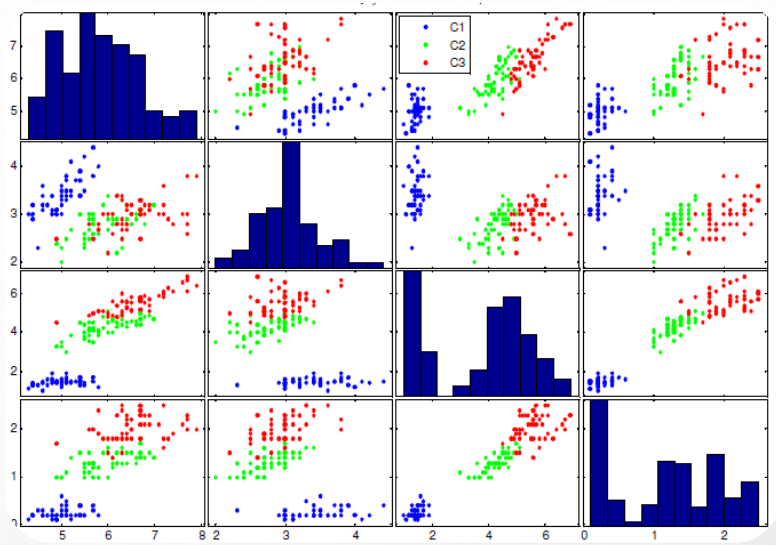
5) Liberação

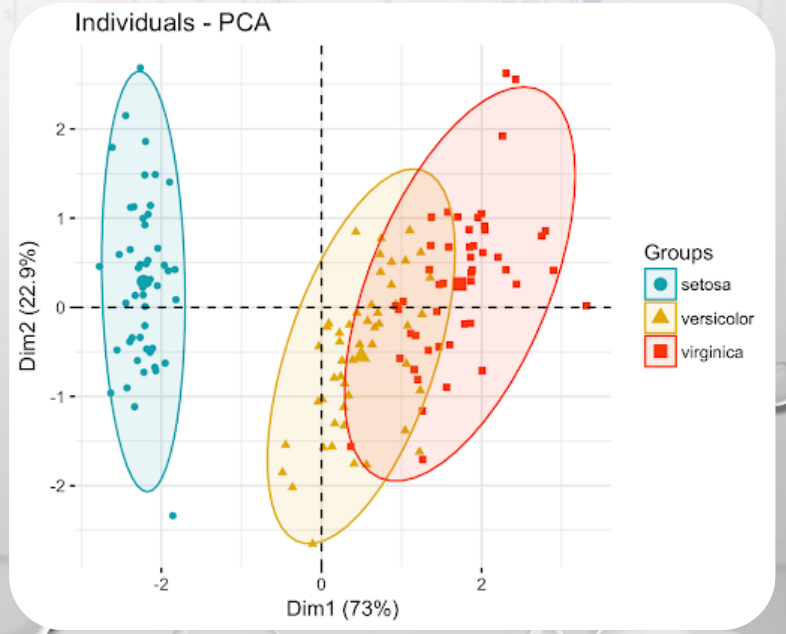
Liberação do modelo no ambiente de produção.

DATA PREPARATION

EXTRAÇÃO DE ATRIBUTOS ANÁLISE DE COMPONENTES PRINCIPAIS (PCA)

- Garantir que as variáveis independentes sejam descorrelacionadas.
- Identificar novas direções com maior concentração de energia / informação.
- Variáveis transformadas perdem o sentido físico.

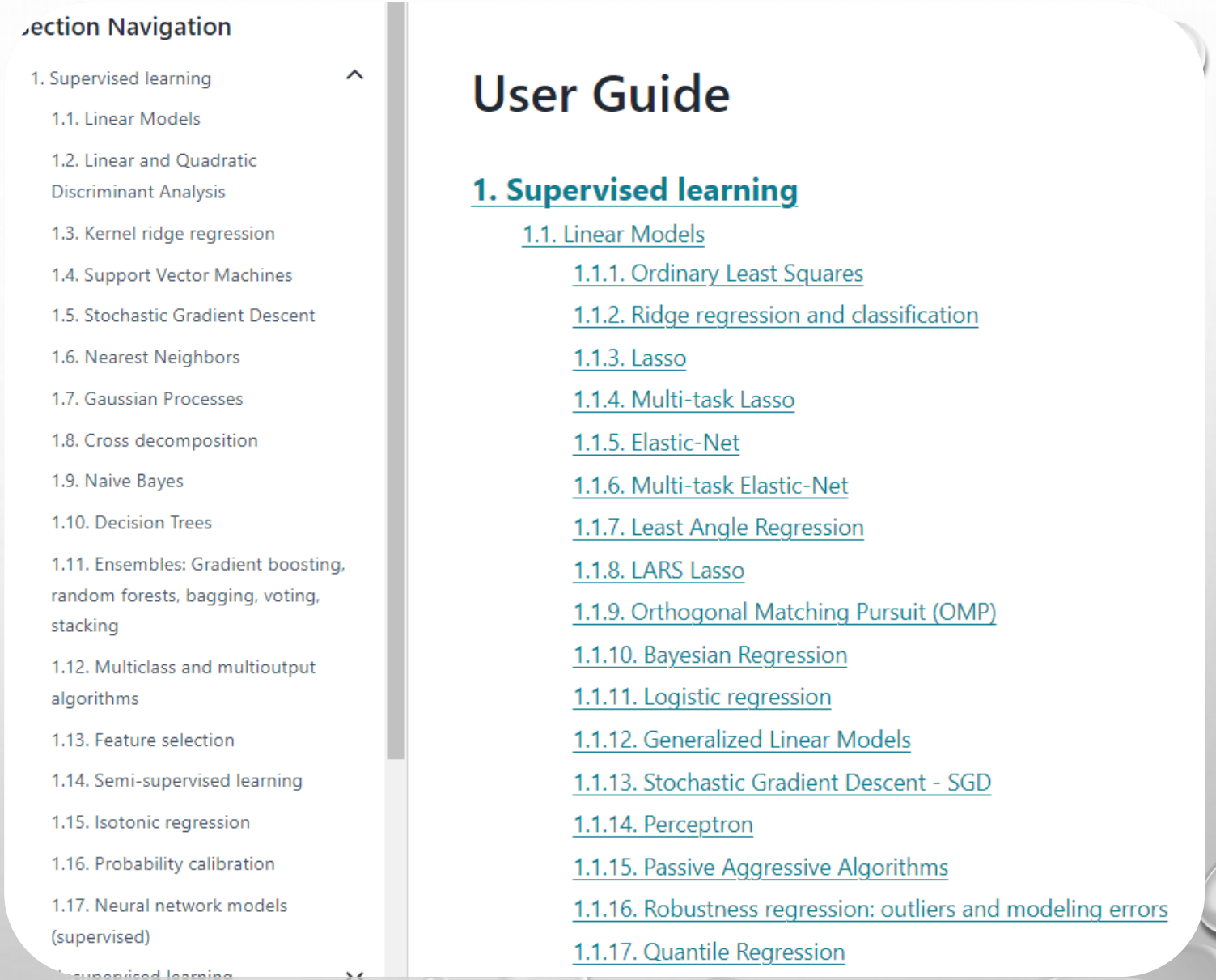




MODELING

DISCRIMINANTES LINEARES E QUADRÁTICOS

USER GUIDE DO SKLEARN

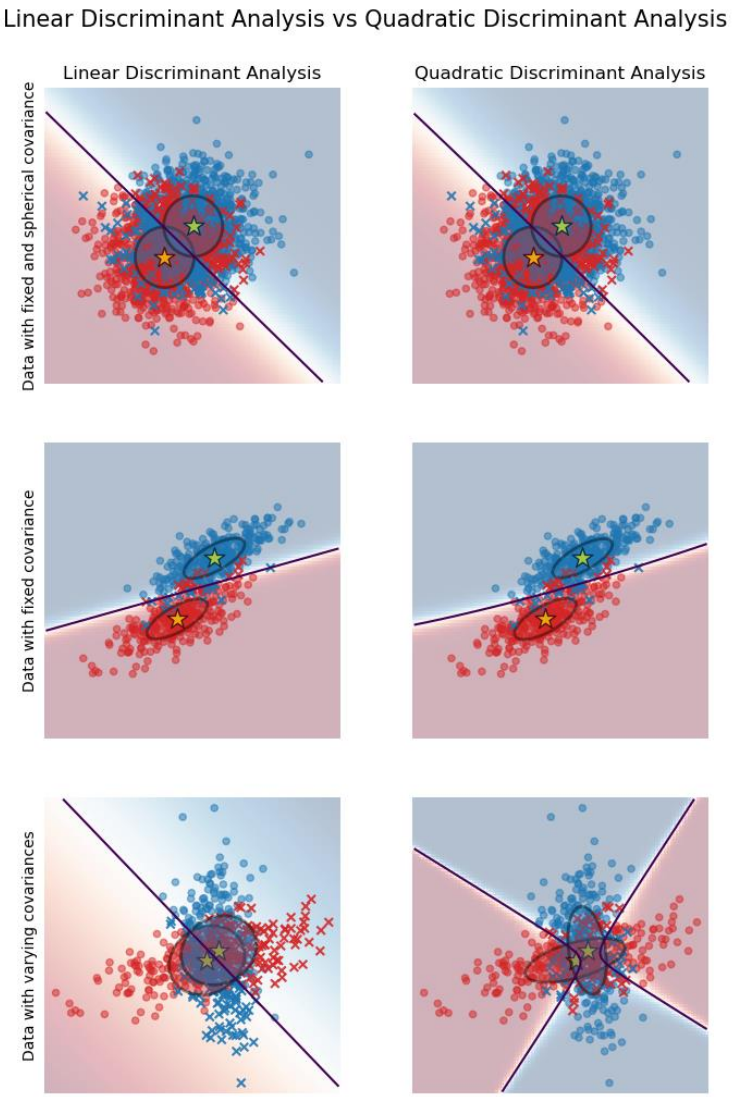


DISCRIMINANTES LINEARES E QUADRÁTICOS

1.2. Linear and Quadratic Discriminant Analysis

Linear Discriminant Analysis (`LinearDiscriminantAnalysis`) and Quadratic Discriminant Analysis (`QuadraticDiscriminantAnalysis`) are two classic classifiers, with, as their names suggest, a linear and a quadratic decision surface, respectively.

These classifiers are attractive because they have closed-form solutions that can be easily computed, are inherently multiclass, have proven to work well in practice, and have no hyperparameters to tune.



QDA E LDA: FORMULAÇÃO

1.2.2. Mathematical formulation of the LDA and QDA classifiers

Both LDA and QDA can be derived from simple probabilistic models which model the class conditional distribution of the data $P(X|y=k)$ for each class k. Predictions can then be obtained by using Bayes' rule, for each training sample $x \in \mathcal{R}^d$:

$$P(y=k|x) = \frac{P(x|y=k)P(y=k)}{P(x)} = \frac{P(x|y=k)P(y=k)}{\sum_l P(x|y=l) \cdot P(y=l)}$$

and we select the class k which maximizes this posterior probability.

More specifically, for linear and quadratic discriminant analysis, $P(x|y)$ is modeled as a multivariate Gaussian distribution with density:

$$P(x|y=k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^t \Sigma_k^{-1}(x-\mu_k)\right)$$

where d is the number of features.

1.2.2.1. QDA

According to the model above, the log of the posterior is:

$$\begin{aligned}\log P(y=k|x) &= \log P(x|y=k) + \log P(y=k) + Cst \\ &= -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^t\Sigma_k^{-1}(x-\mu_k) + \log P(y=k) + Cst,\end{aligned}$$

where the constant term Cst corresponds to the denominator $P(x)$, in addition to other constant terms from the Gaussian. The predicted class is the one that maximises this log-posterior.

1.2.2.2. LDA

LDA is a special case of QDA, where the Gaussians for each class are assumed to share the same covariance matrix: $\Sigma_k = \Sigma$ for all k. This reduces the log posterior to:

$$\log P(y=k|x) = -\frac{1}{2}(x-\mu_k)^t\Sigma^{-1}(x-\mu_k) + \log P(y=k) + Cst.$$

The term $(x-\mu_k)^t\Sigma^{-1}(x-\mu_k)$ corresponds to the Mahalanobis Distance between the sample x and the mean μ_k. The Mahalanobis distance tells how close x is from μ_k, while also accounting for the variance of each feature. We can thus interpret LDA as assigning x to the class whose mean is the closest in terms of Mahalanobis distance, while also accounting for the class prior probabilities.

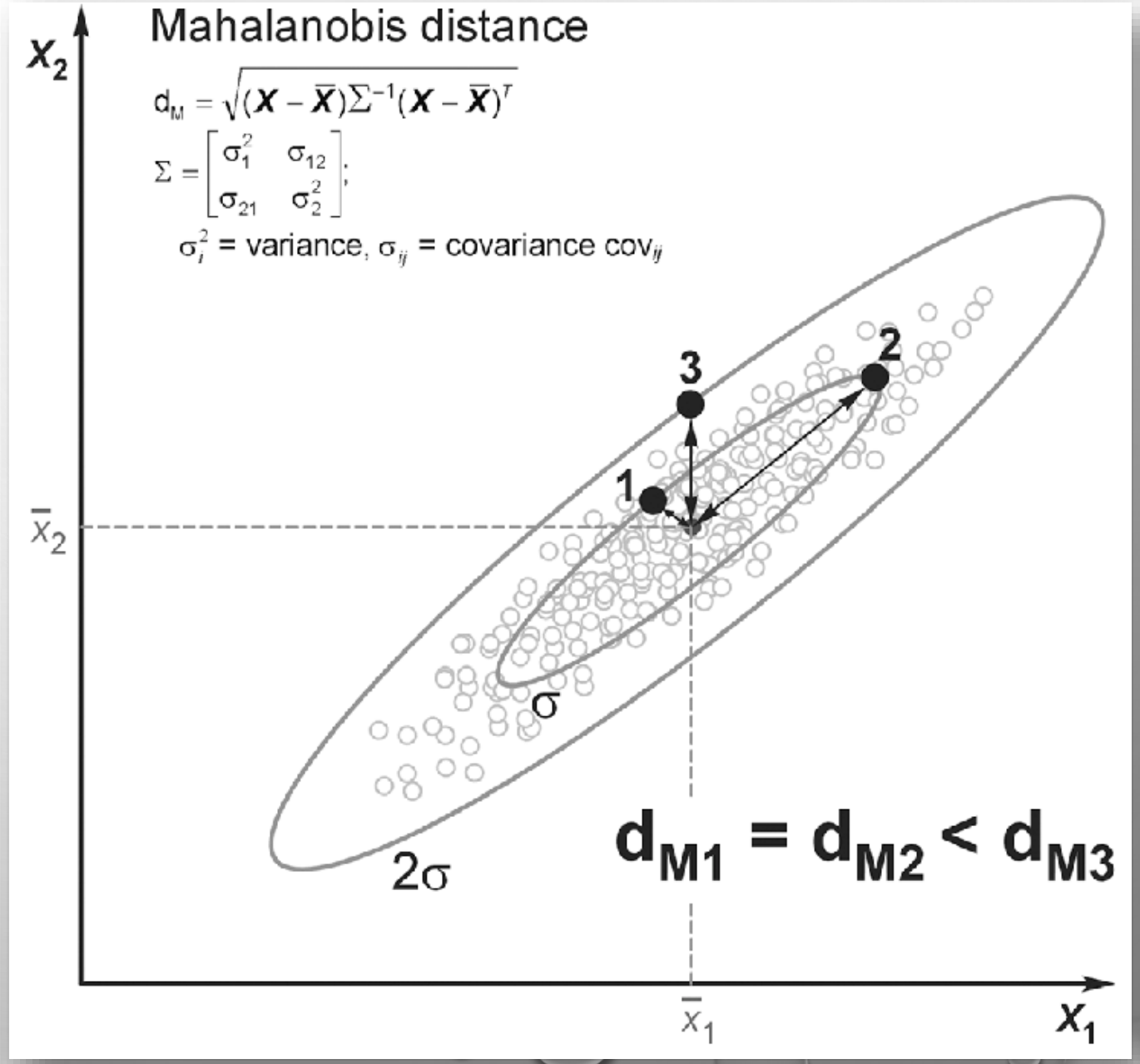
The log-posterior of LDA can also be written [3] as:

$$\log P(y=k|x) = \omega_k^t x + \omega_{k0} + Cst.$$

where $\omega_k = \Sigma^{-1}\mu_k$ and $\omega_{k0} = -\frac{1}{2}\mu_k^t\Sigma^{-1}\mu_k + \log P(y=k)$. These quantities correspond to the `coef_` and `intercept_` attributes, respectively.

From the above formula, it is clear that LDA has a linear decision surface. In the case of QDA, there are no assumptions on the covariance matrices Σ_k of the Gaussians, leading to quadratic decision surfaces. See [1] for more details.

DISTÂNCIA DE MAHALANOBIS



QDA E LDA: ESTIMAÇÃO

1.2.5. Estimation algorithms

Using LDA and QDA requires computing the log-posterior which depends on the class priors $P(y = k)$, the class means μ_k, and the covariance matrices.

The 'svd' solver is the default solver used for `LinearDiscriminantAnalysis`, and it is the only available solver for `QuadraticDiscriminantAnalysis`. It can perform both classification and transform (for LDA). As it does not rely on the calculation of the covariance matrix, the 'svd' solver may be preferable in situations where the number of features is large. The 'svd' solver cannot be used with shrinkage. For QDA, the use of the SVD solver relies on the fact that the covariance matrix Σ_k is, by definition, equal to $\frac{1}{n-1} X_k^t X_k = \frac{1}{n-1} V S^2 V^t$ where V comes from the SVD of the (centered) matrix: $X_k = U S V^t$. It turns out that we can compute the log-posterior above without having to explicitly compute Σ: computing S and V via the SVD of X is enough. For LDA, two SVDs are computed: the SVD of the centered input matrix X and the SVD of the class-wise mean vectors.

The 'lsqr' solver is an efficient algorithm that only works for classification. It needs to explicitly compute the covariance matrix Σ, and supports shrinkage and custom covariance estimators. This solver computes the coefficients $\omega_k = \Sigma^{-1} \mu_k$ by solving for $\Sigma \omega = \mu_k$, thus avoiding the explicit computation of the inverse Σ^{-1}.

The 'eigen' solver is based on the optimization of the between class scatter to within class scatter ratio. It can be used for both classification and transform, and it supports shrinkage. However, the 'eigen' solver needs to compute the covariance matrix, so it might not be suitable for situations with a high number of features.

SVD

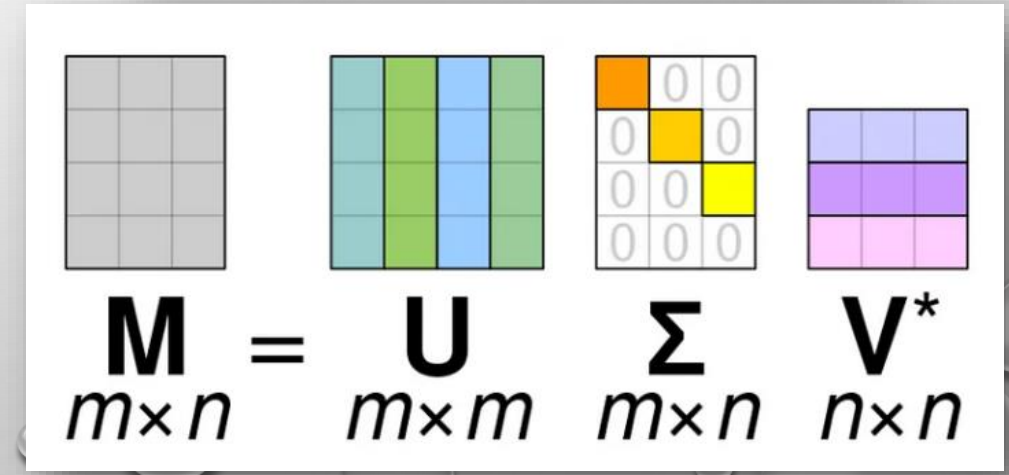
Singular value decomposition (SVD) is a matrix factorization method that generalizes the eigendecomposition of a square matrix (n x n) to any matrix (n x m) (source).

If you don't know what is eigendecomposition or eigenvectors/eigenvalues, you should google it or read this post. This post assumes that you are familiar with these concepts.

SVD is similar to Principal Component Analysis (PCA), but more general. PCA assumes that input square matrix, SVD doesn't have this assumption. General formula of SVD is:

$M=U\Sigma V^t$, where:

- **M**-is original matrix we want to decompose
- **U**-is left singular matrix (columns are left singular vectors). **U** columns contain eigenvectors of matrix MM^t
- **Σ**-is a diagonal matrix containing singular (eigen)values
- **V**-is right singular matrix (columns are right singular vectors). **V** columns contain eigenvectors of matrix M^tM



https://towardsdatascience.com/simple-svd-algorithms-13291ad2eef2

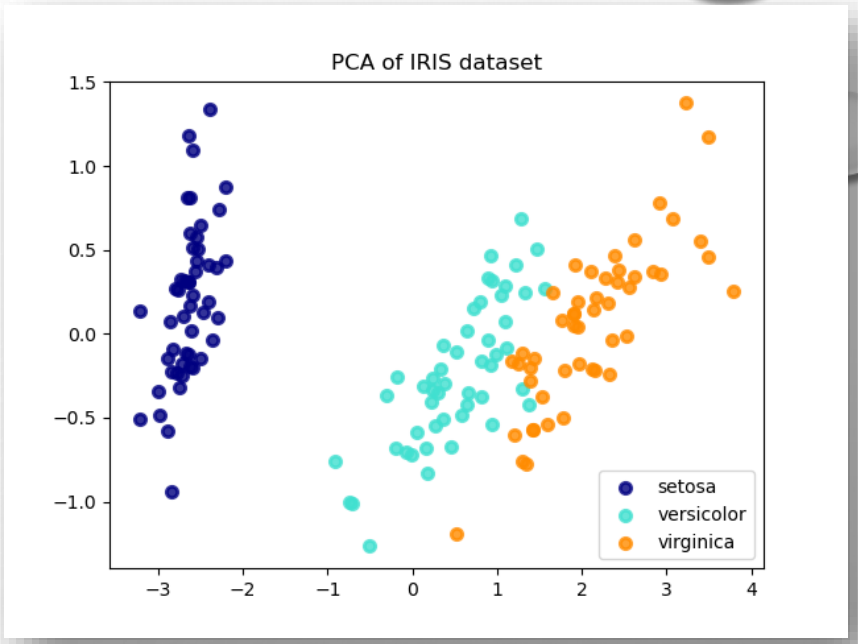
QDA E LDA: PROJEÇÃO

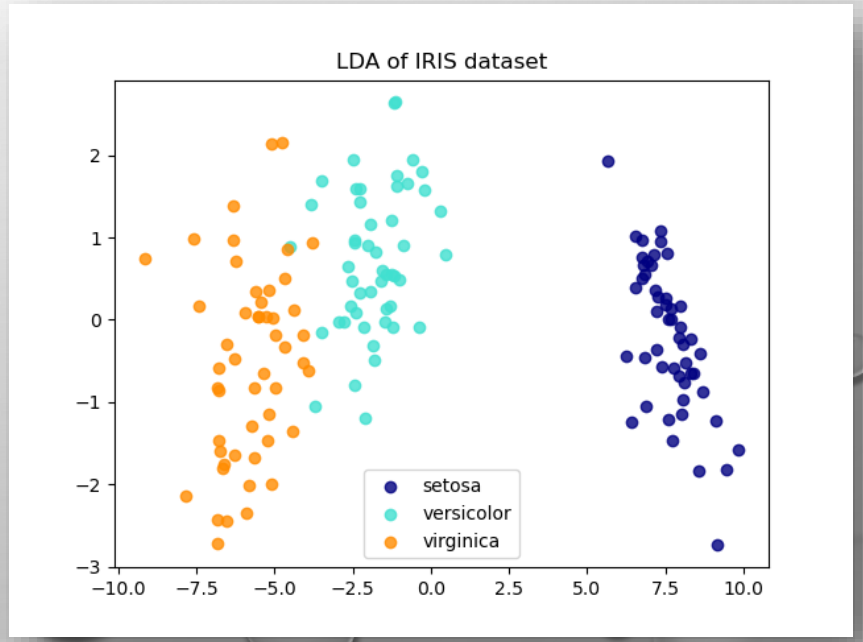
Comparison of LDA and PCA 2D projection of Iris dataset

The Iris dataset represents 3 kind of Iris flowers (Setosa, Versicolour and Virginica) with 4 attributes: sepal length, sepal width, petal length and petal width.

Principal Component Analysis (PCA) applied to this data identifies the combination of attributes (principal components, or directions in the feature space) that account for the most variance in the data. Here we plot the different samples on the 2 first principal components.

Linear Discriminant Analysis (LDA) tries to identify attributes that account for the most variance *between classes*. In particular, LDA, in contrast to PCA, is a supervised method, using known class labels.





REGRESSÃO LOGÍSTICA

REGRESSÃO LOGÍSTICA

A **logistic function** or **logistic curve** is a common S-shaped curve (sigmoid curve) with the equation

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

where

L is the carrying capacity, the supremum of the values of the function;
k is the logistic growth rate, the steepness of the curve; and
x_0 is the x value of the function's midpoint.[1]

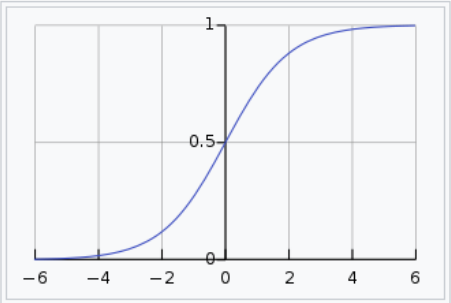
The logistic function has domain the real numbers, the limit as $x \to -\infty$ is 0, and the limit as $x \to +\infty$ is L.

The **standard logistic function**, depicted at right, where $L = 1, k = 1, x_0 = 0$, has the equation

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

and is sometimes simply called **the sigmoid**.[2] It is also sometimes called the **expit**, being the inverse of the logit.[3][4]

The logistic function finds applications in a range of fields, including biology (especially ecology), biomathematics, chemistry, demography, economics, geoscience, mathematical psychology, probability, sociology, political science, linguistics, statistics, and artificial neural networks. There are various generalizations, depending on the field.



Standard logistic function where $L = 1, k = 1, x_0 = 0$.

1.1.11. Logistic regression

The logistic regression is implemented in `LogisticRegression`. Despite its name, it is implemented as a linear model for classification rather than regression in terms of the scikit-learn/ML nomenclature. The 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.

This implementation can fit binary, One-vs-Rest, or multinomial logistic regression with optional ℓ_1, ℓ_2 or Elastic-Net regularization.

Note

Regularization

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.

REGRESSÃO LOGÍSTICA: CASO BINÁRIO

1.1.11.1. Binary Case

For notational ease, we assume that the target y_i takes values in the set $\{0, 1\}$ for data point i. Once fitted, the `predict_proba` method of `LogisticRegression` predicts the probability of the positive class $P(y_i = 1|X_i)$ as

$$\hat{p}(X_i) = \operatorname{expit}(X_i w + w_0) = \frac{1}{1 + \exp(-X_i w - w_0)}.$$

As an optimization problem, binary class logistic regression with regularization term $r(w)$ minimizes the following cost function:

$$\min_w \frac{1}{S} \sum_{i=1}^{n} s_i \left(-y_i \log(\hat{p}(X_i)) - (1 - y_i) \log(1 - \hat{p}(X_i))\right) + \frac{r(w)}{SC}, \tag{1}$$

where s_i corresponds to the weights assigned by the user to a specific training sample (the vector s is formed by element-wise multiplication of the class weights and sample weights), and the sum $S = \sum_{i=1}^{n} s_i$.

We currently provide four choices for the regularization term $r(w)$ via the `penalty` argument:

penalty	$r(w)$
`None`	0
ℓ_1	$\|w\|_1$
ℓ_2	$\frac{1}{2}\|w\|_2^2 = \frac{1}{2} w^T w$
`ElasticNet`	$\frac{1-\rho}{2} w^T w + \rho \|w\|_1$

For ElasticNet, ρ (which corresponds to the `l1_ratio` parameter) controls the strength of ℓ_1 regularization vs. ℓ_2 regularization. Elastic-Net is equivalent to ℓ_1 when $\rho = 1$ and equivalent to ℓ_2 when $\rho = 0$.

Note that the scale of the class weights and the sample weights will influence the optimization problem. For instance, multiplying the sample weights by a constant $b > 0$ is equivalent to multiplying the (inverse) regularization strength `C` by b.

REGRESSÃO LOGÍSTICA: CASO MULTIVARIADO

1.1.11.2. Multinomial Case

The binary case can be extended to K classes leading to the multinomial logistic regression, see also log-linear model.

Note

It is possible to parameterize a K-class classification model using only $K-1$ weight vectors, leaving one class probability fully determined by the other class probabilities by leveraging the fact that all class probabilities must sum to one. We deliberately choose to overparameterize the model using K weight vectors for ease of implementation and to preserve the symmetrical inductive bias regarding ordering of classes, see [16]. This effect becomes especially important when using regularization. The choice of overparameterization can be detrimental for unpenalized models since then the solution may not be unique, as shown in [16].

As a log-linear model [edit]

The formulation of binary logistic regression as a log-linear model can be directly extended to multi-way regression. That is, we model the logarithm of the probability of seeing a given output using the linear predictor as well as an additional normalization factor, the logarithm of the partition function:

$$\ln \Pr(Y_i = k) = \boldsymbol{\beta}_k \cdot \mathbf{X}_i - \ln Z \quad , \quad k \leq K.$$

As in the binary case, we need an extra term $-\ln Z$ to ensure that the whole set of probabilities forms a probability distribution, i.e. so that they all sum to one:

$$\sum_{k=1}^{K} \Pr(Y_i = k) = 1$$

The reason why we need to add a term to ensure normalization, rather than multiply as is usual, is because we have taken the logarithm of the probabilities. Exponentiating both sides turns the additive term into a multiplicative factor, so that the probability is just the Gibbs measure:

$$\Pr(Y_i = k) = \frac{1}{Z} e^{\boldsymbol{\beta}_k \cdot \mathbf{X}_i} \quad , \quad k \leq K.$$

The quantity Z is called the partition function for the distribution. We can compute the value of the partition function by applying the above constraint that requires all probabilities to sum to 1:

$$1 = \sum_{k=1}^{K} \Pr(Y_i = k) = \sum_{k=1}^{K} \frac{1}{Z} e^{\boldsymbol{\beta}_k \cdot \mathbf{X}_i} = \frac{1}{Z} \sum_{k=1}^{K} e^{\boldsymbol{\beta}_k \cdot \mathbf{X}_i}$$

Therefore:

$$Z = \sum_{k=1}^{K} e^{\boldsymbol{\beta}_k \cdot \mathbf{X}_i}$$

Note that this factor is "constant" in the sense that it is not a function of Y_i, which is the variable over which the probability distribution is defined. However, it is definitely not constant with respect to the explanatory variables, or crucially, with respect to the unknown regression coefficients $\boldsymbol{\beta}_k$, which we will need to determine through some sort of optimization procedure.

The resulting equations for the probabilities are

$$\Pr(Y_i = k) = \frac{e^{\boldsymbol{\beta}_k \cdot \mathbf{X}_i}}{\sum_{j=1}^{K} e^{\boldsymbol{\beta}_j \cdot \mathbf{X}_i}} \quad , \quad k \leq K.$$

Or generally:

$$\Pr(Y_i = c) = \frac{e^{\boldsymbol{\beta}_c \cdot \mathbf{X}_i}}{\sum_{j=1}^{K} e^{\boldsymbol{\beta}_j \cdot \mathbf{X}_i}}$$

REGRESSÃO LOGÍSTICA: SOLVERS

1.1.11.3. Solvers

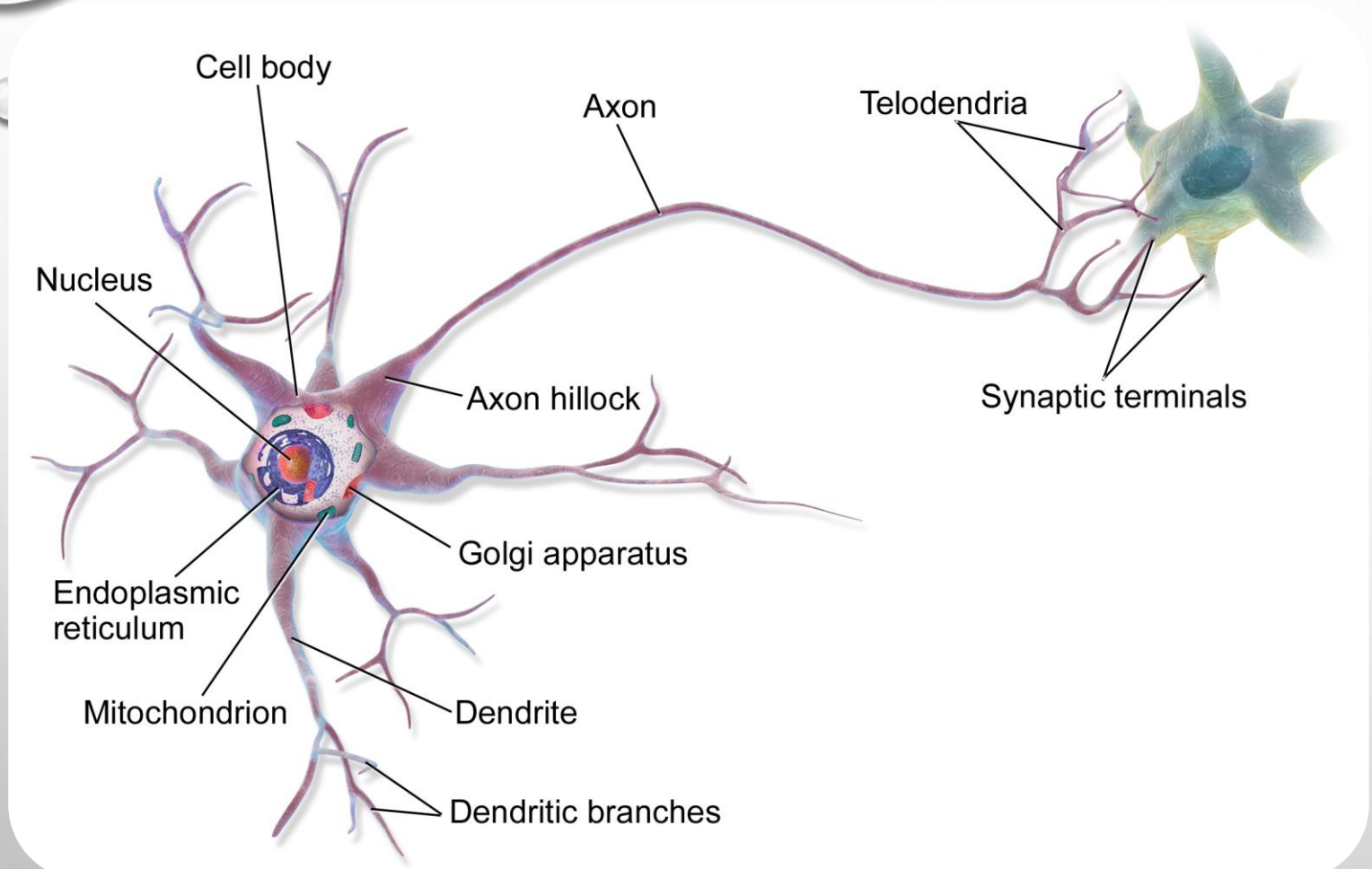
The solvers implemented in the class `LogisticRegression` are “lbfgs”, “liblinear”, “newton-cg”, “newton-cholesky”, “sag” and “saga”:

The following table summarizes the penalties and multinomial multiclass supported by each solver:

	Solvers					
Penalties	**‘lbfgs’**	**‘liblinear’**	**‘newton-cg’**	**‘newton-cholesky’**	**‘sag’**	**‘saga’**
L2 penalty	yes	no	yes	no	yes	yes
L1 penalty	no	yes	no	no	no	yes
Elastic-Net (L1 + L2)	no	no	no	no	no	yes
No penalty (‘none’)	yes	no	yes	yes	yes	yes
Multiclass support						
multinomial multiclass	yes	no	yes	no	yes	yes
Behaviors						
Penalize the intercept (bad)	no	yes	no	no	no	no
Faster for large datasets	no	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	yes	no	no

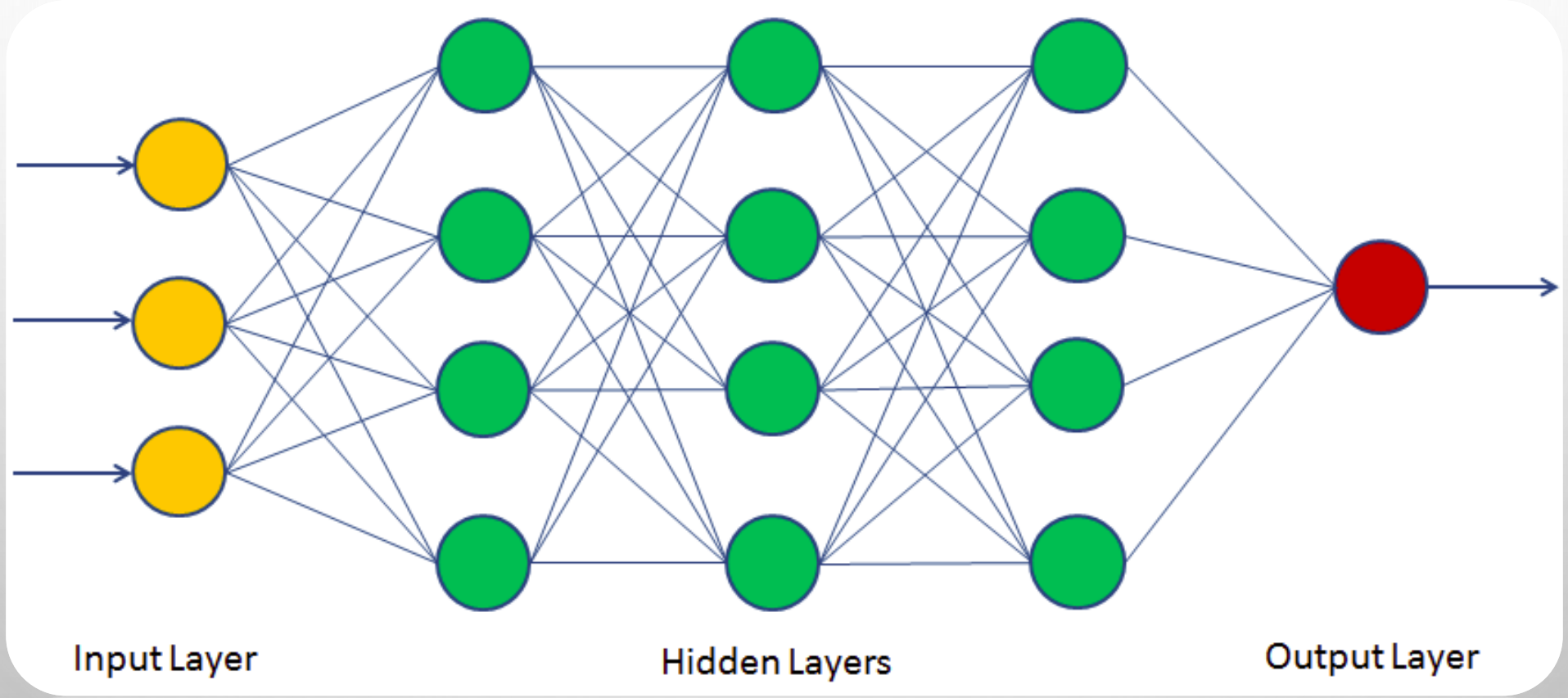
The “lbfgs” solver is used by default for its robustness. For large datasets the “saga” solver is usually faster. For large dataset, you may also consider using `SGDClassifier` with `loss="log_loss"`, which might be even faster but requires more tuning.

REDES NEURAIS

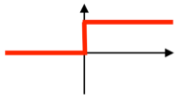
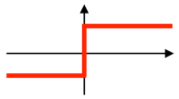
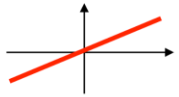
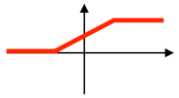
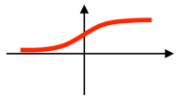
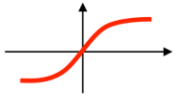
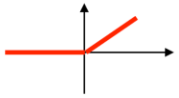
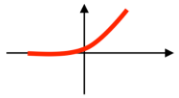


INSPIRAÇÃO BIOLÓGICA

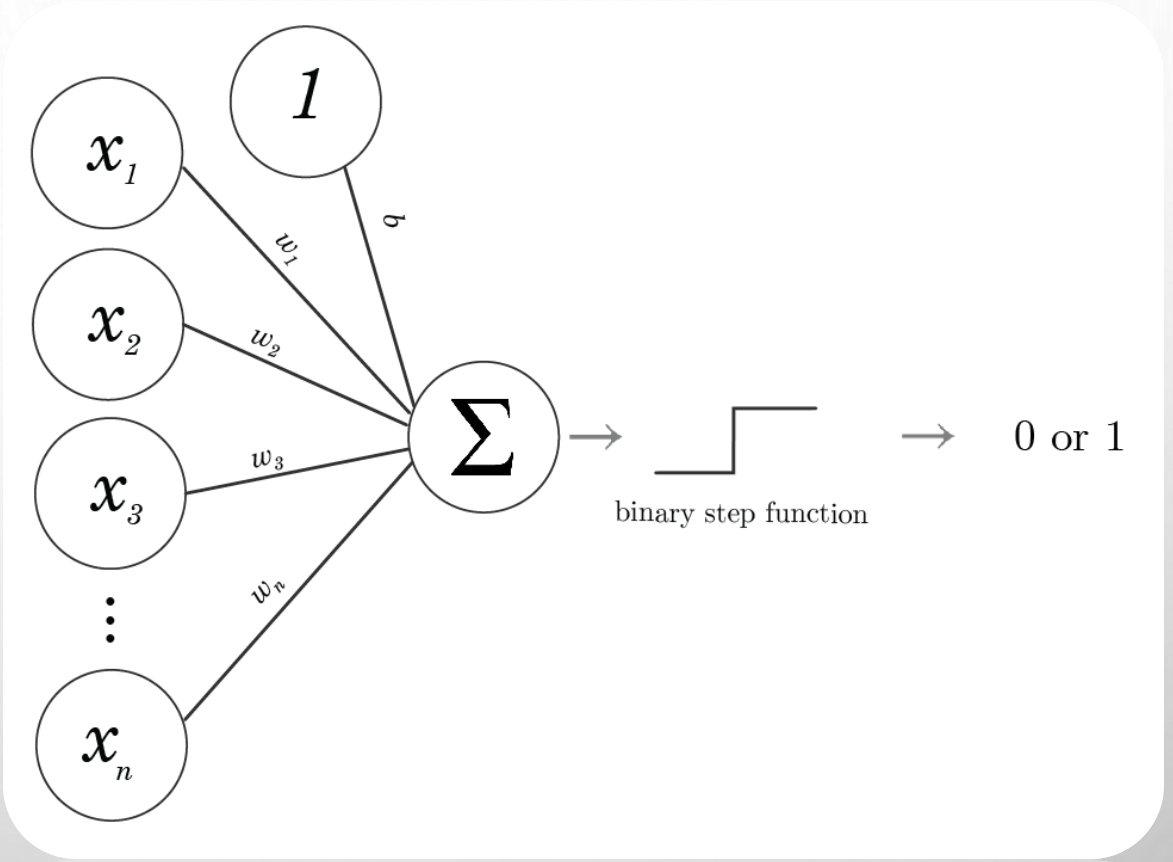
O APROXIMADOR UNIVERSAL

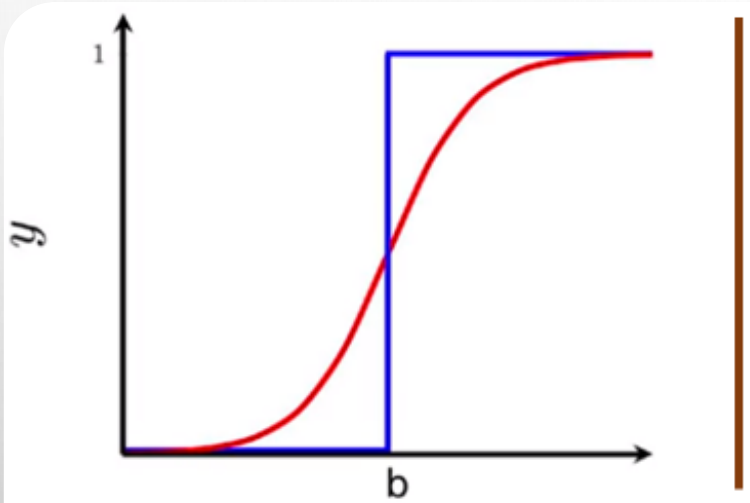


FUNÇÕES DE ATIVAÇÃO

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

PERCEPTRON DE ROSENBLATT





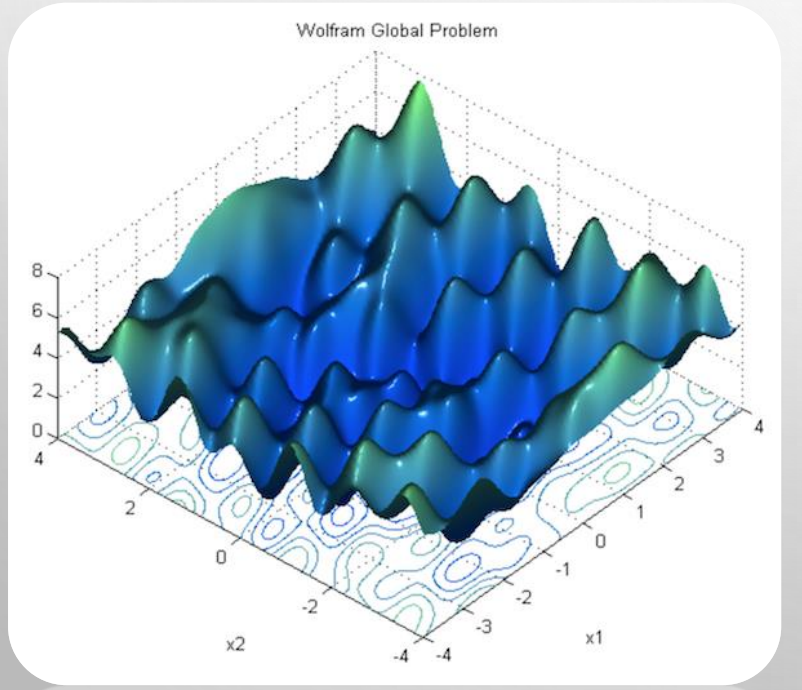
$$y = \frac{1}{1+e^{-(w^T x+b)}}$$

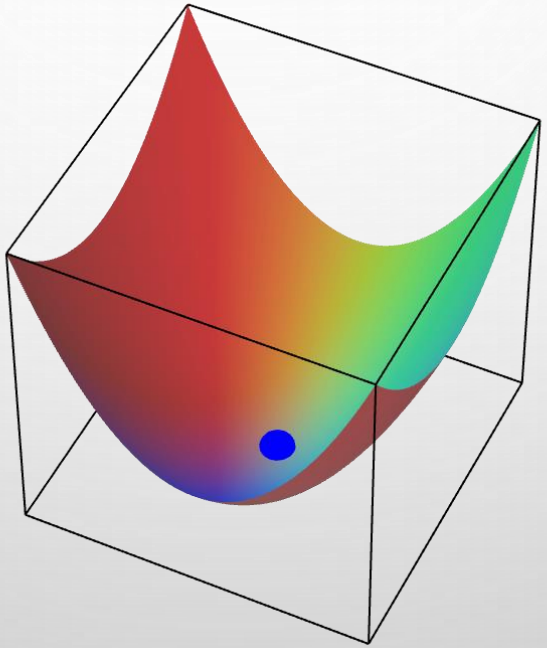
- $\sum_{i=1}^{n} w_i x_i$

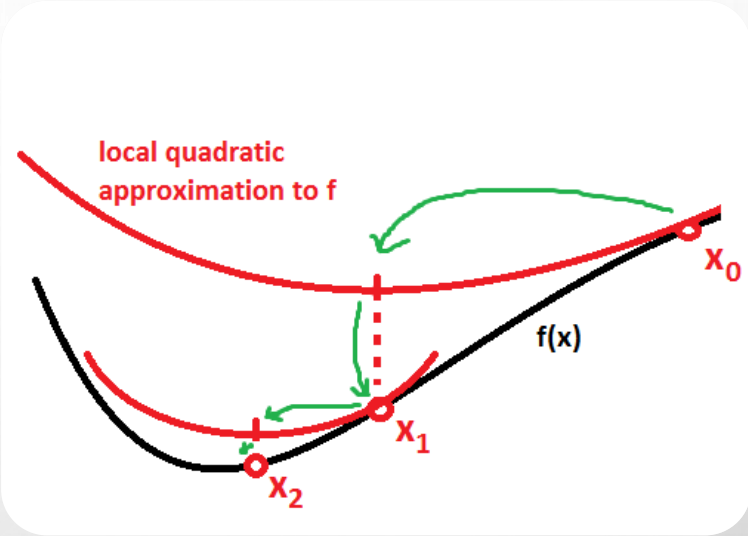
NEURÔNIO SIGMÓIDE

SUPERFÍCIE DO ERRO MÉDIO QUADRÁTICO

$$E = \frac{1}{N}\sum (y - f(w, x))^2$$







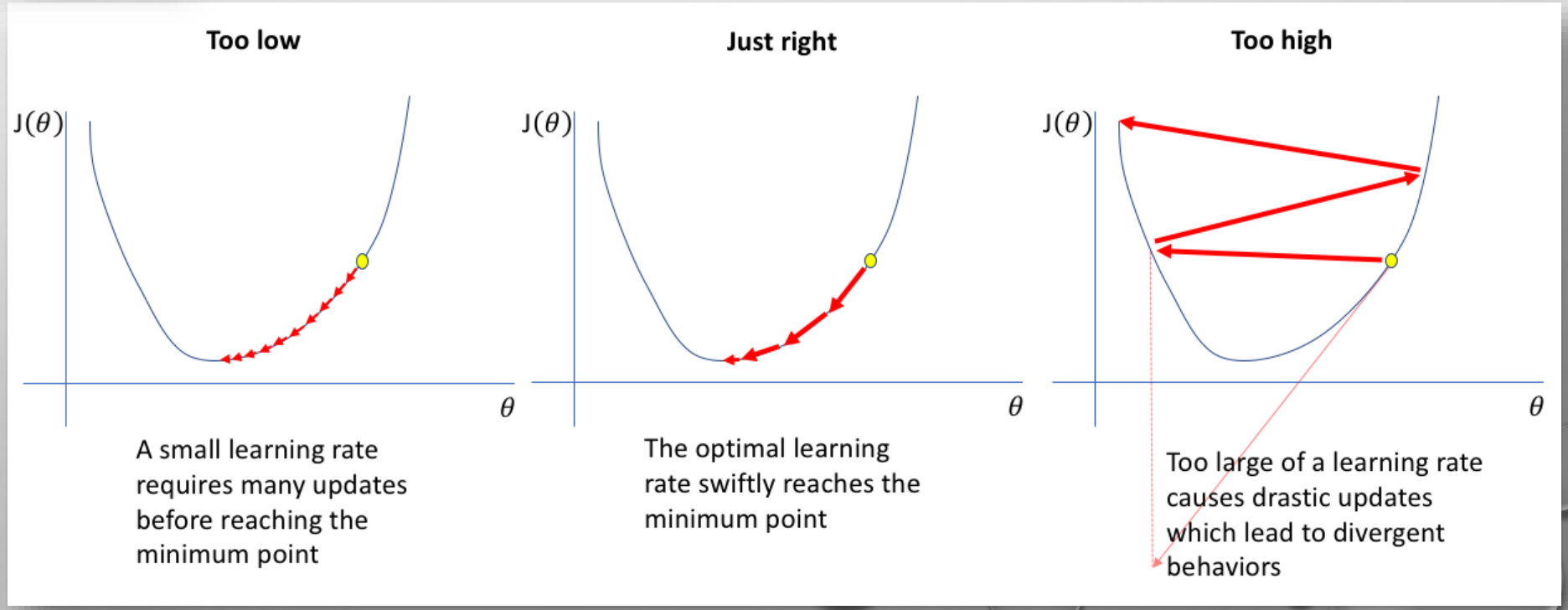
ALGORITMO DO GRADIENTE DESCENDENTE

$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}})$$

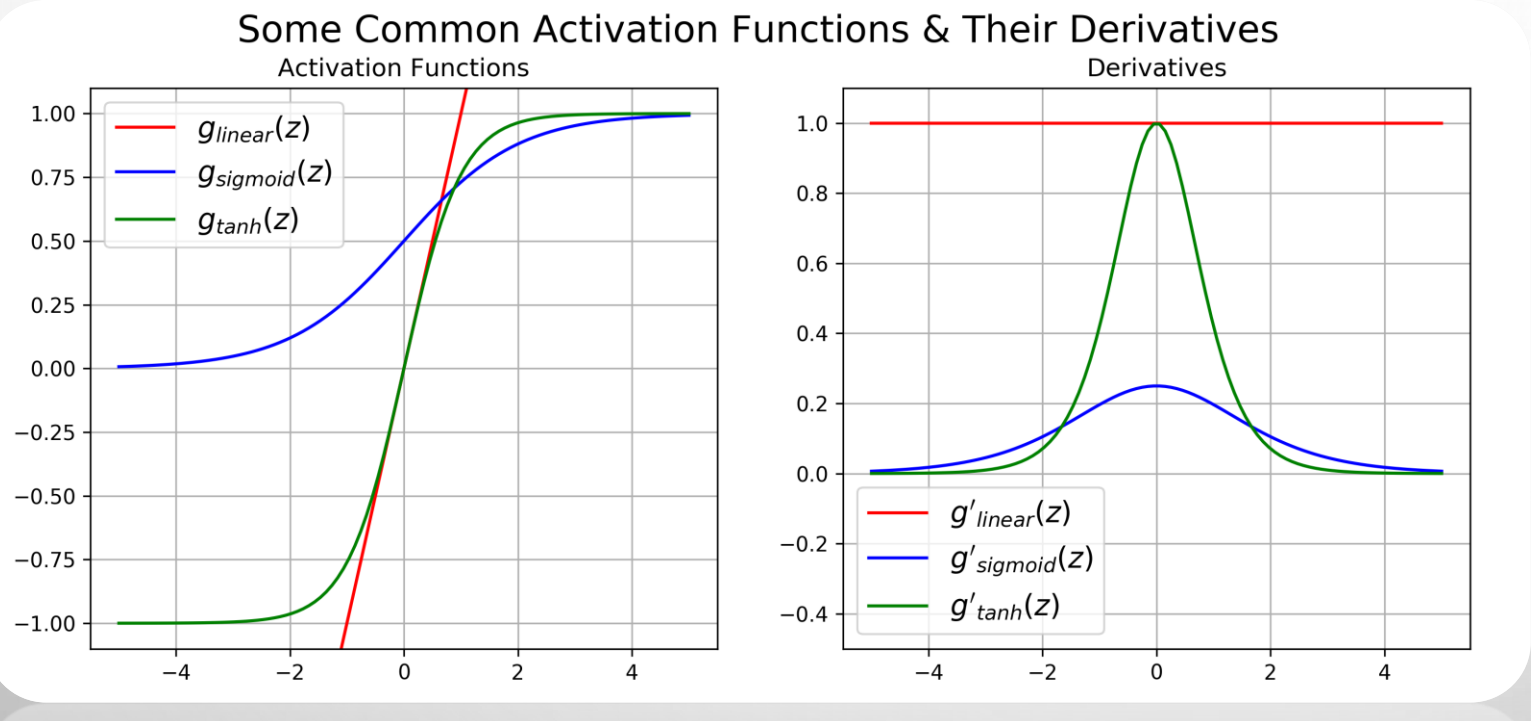
weight increment — learning rate — weight gradient

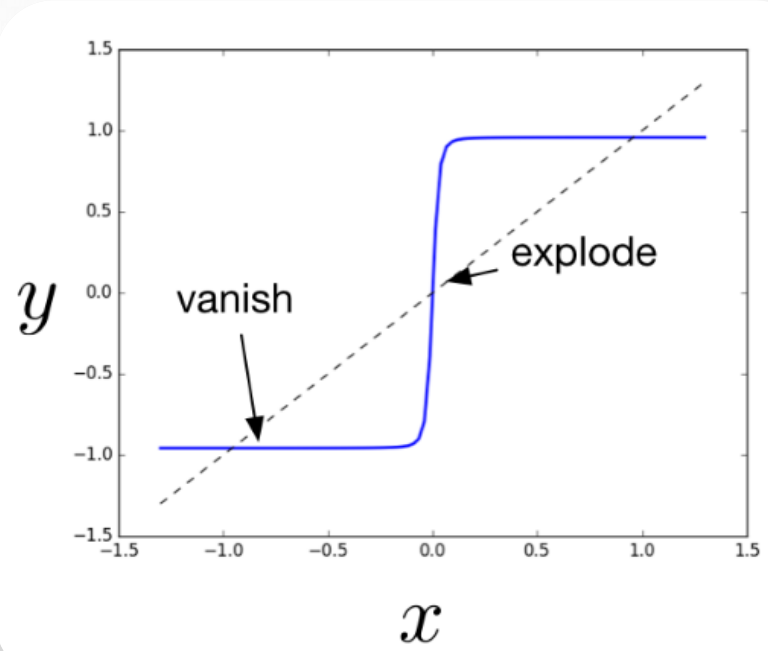
$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}}) + (\gamma * \Delta w_{ij}^{t-1})$$

momentum factor — weight increment, previous iteration



O PROBLEMA DA DISSIPAÇÃO DO GRADIENTE





OTIMIZADORES (REGULARIZADOS)

$$v_t^w = v_{t-1}^w + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t^w + \epsilon}} * \nabla w_t$$

$$v_t^b = v_{t-1}^b + (\nabla b_t)^2$$
$$b_{t+1} = b_t - \frac{\eta}{\sqrt{v_t^b + \epsilon}} * \nabla b_t$$

ADAGRAD

$$v_t^w = \beta * v_{t-1}^w + (1-\beta)(\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t^w + \epsilon}} * \nabla w_t$$

$$v_t^b = \beta * v_{t-1}^b + (1-\beta)(\nabla b_t)^2$$
$$b_{t+1} = b_t - \frac{\eta}{\sqrt{v_t^b + \epsilon}} * \nabla b_t$$

RMSPROP

$$m_t = \beta_1 * m_{t-1} + (1-\beta_1) * \nabla w_t$$
$$v_t = \beta_2 * v_{t-1} + (1-\beta_2) * (\nabla w_t)^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

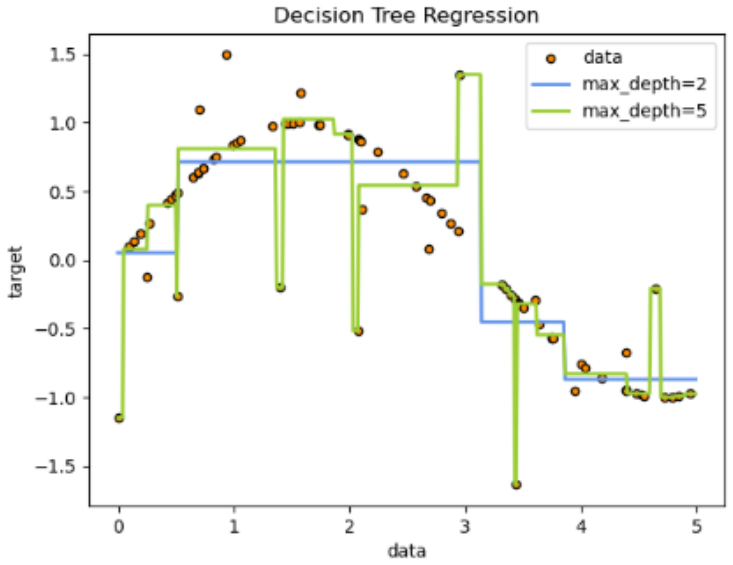
ADAM

ÁRVORES DE DECISÃO

ÁRVORES DE DECISÃO

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.

For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.



PROS AND CONS

Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualized.
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Some tree and algorithm combinations 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, the scikit-learn implementation does not support categorical variables for now. Other techniques are usually specialized in analyzing 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.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalize 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.

CLASSIFICAÇÃO

`DecisionTreeClassifier` is a class capable of performing multi-class classification on a dataset.

As with other classifiers, `DecisionTreeClassifier` takes as input two arrays: an array X, sparse or dense, of shape `(n_samples, n_features)` holding the training samples, and an array Y of integer values, shape `(n_samples,)`, holding the class labels for the training samples:

```
>>> from sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, Y)
```

After being fitted, the model can then be used to predict the class of samples:

```
>>> clf.predict([[2., 2.]])
array([1])
```

In case that there are multiple classes with the same and highest probability, the classifier will predict the class with the lowest index amongst those classes.

As an alternative to outputting a specific class, the probability of each class can be predicted, which is the fraction of training samples of the class in a leaf:

```
>>> clf.predict_proba([[2., 2.]])
array([[0., 1.]])
```

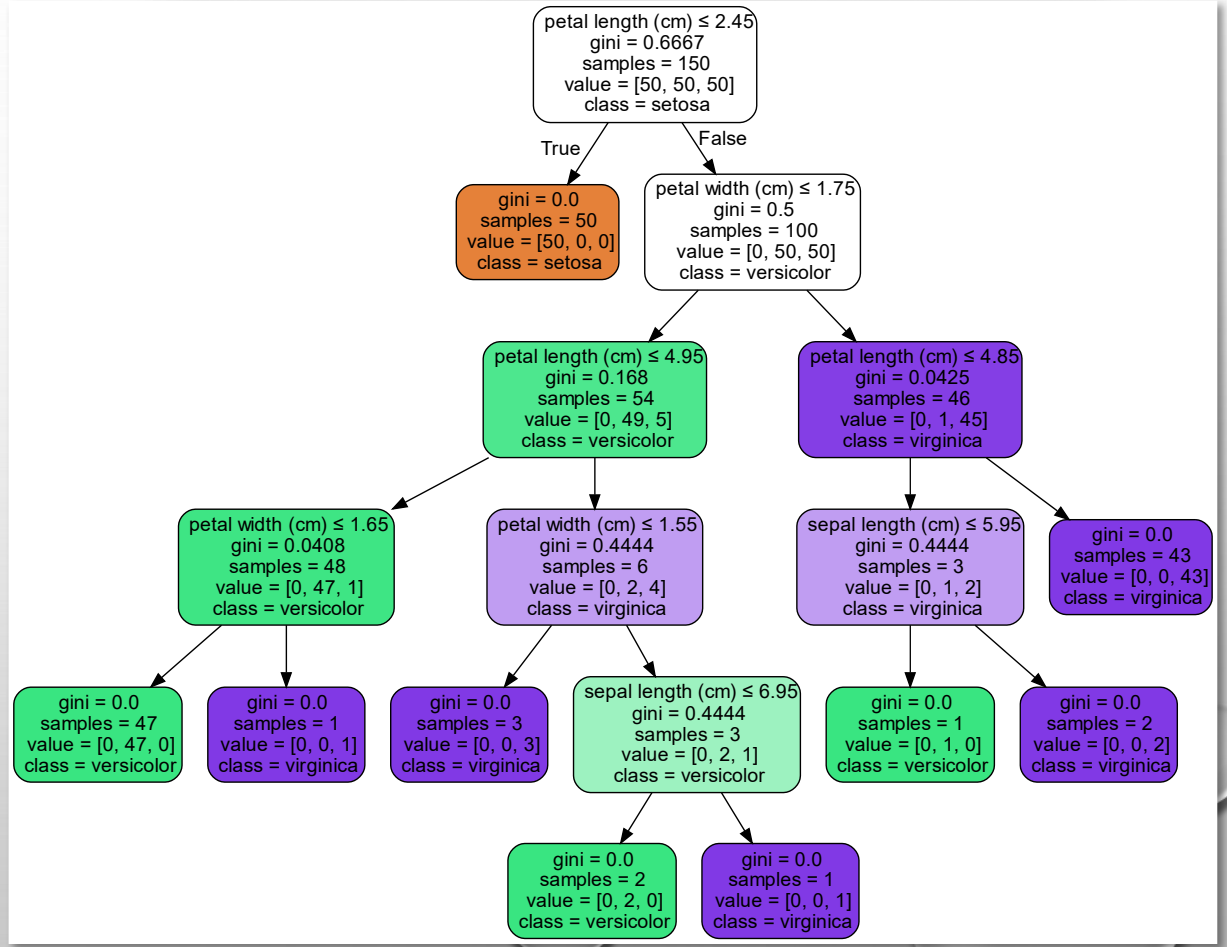
`DecisionTreeClassifier` is capable of both binary (where the labels are [-1, 1]) classification and multiclass (where the labels are [0, ..., K-1]) classification.

Using the Iris dataset, we can construct a tree as follows:

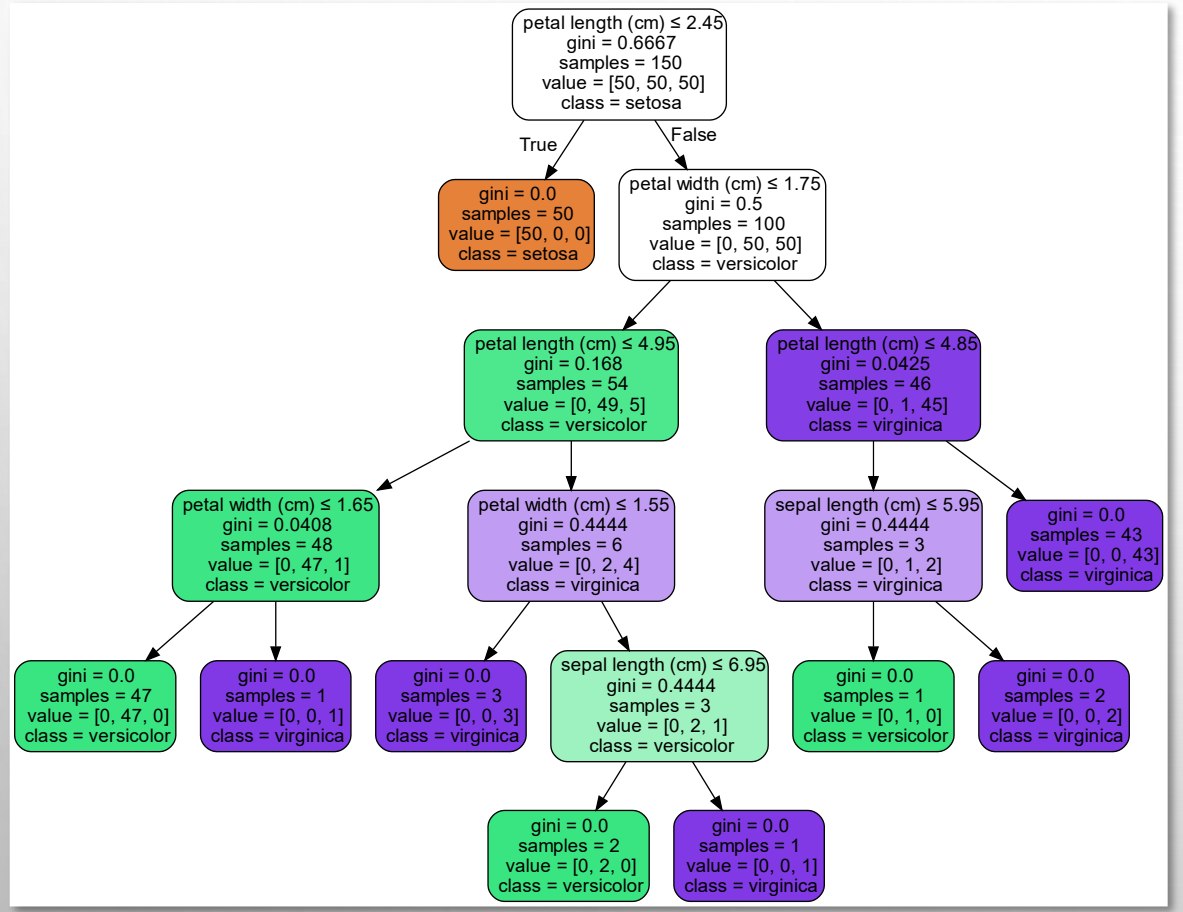
```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, y)
```

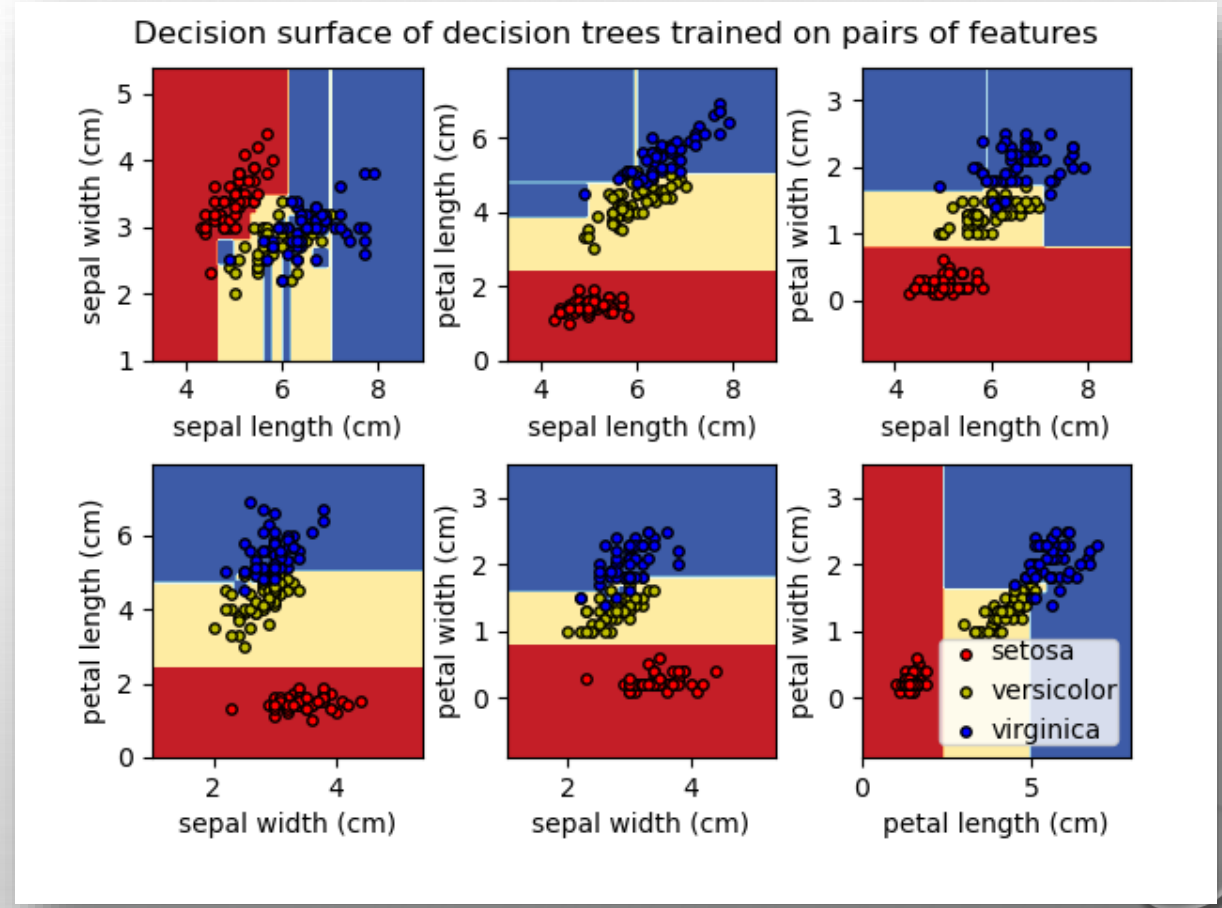
Once trained, you can plot the tree with the `plot_tree` function:

```
>>> tree.plot_tree(clf)
[...]
```



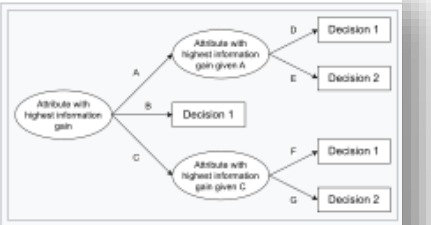
EXEMPLO IRIS





ALGORITMO ID3

In decision tree learning, **ID3** (**Iterative Dichotomiser 3**) is an algorithm invented by Ross Quinlan[1] used to generate a decision tree from a dataset. ID3 is the precursor to the C4.5 algorithm, and is typically used in the machine learning and natural language processing domain



Potential ID3-generated decision tree. Attributes are arranged as nodes by ability to classify examples. Values of attributes are represented by branches.

Algorithm [edit]

The ID3 algorithm begins with the original set S as the root node. On each iteration of the algorithm, it iterates through every unused attribute of the set S and calculates the entropy $\mathrm{H}(S)$ or the information gain $IG(S)$ of that attribute. It then selects the attribute which has the smallest entropy (or largest information gain) value. The set S is then split or partitioned by the selected attribute to produce subsets of the data. (For example, a node can be split into child nodes based upon the subsets of the population whose ages are less than 50, between 50 and 100, and greater than 100.) The algorithm continues to recurse on each subset, considering only attributes never selected before.

Recursion on a subset may stop in one of these cases:

- every element in the subset belongs to the same class; in which case the node is turned into a leaf node and labelled with the class of the examples.
- there are no more attributes to be selected, but the examples still do not belong to the same class. In this case, the node is made a leaf node and labelled with the most common class of the examples in the subset.
- there are no examples in the subset, which happens when no example in the parent set was found to match a specific value of the selected attribute. An example could be the absence of a person among the population with age over 100 years. Then a leaf node is created and labelled with the most common class of the examples in the parent node's set.

Throughout the algorithm, the decision tree is constructed with each non-terminal node (internal node) representing the selected attribute on which the data was split, and terminal nodes (leaf nodes) representing the class label of the final subset of this branch.

Summary [edit]

1. Calculate the entropy of every attribute a of the data set S.
2. Partition ("split") the set S into subsets using the attribute for which the resulting entropy after splitting is minimized; or, equivalently, information gain is maximum.
3. Make a decision tree node containing that attribute.
4. Recurse on subsets using the remaining attributes.

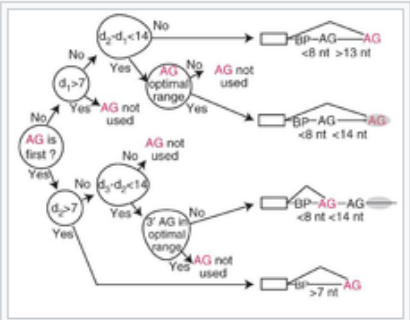
ALGORITMO ID3

Properties [edit]

ID3 does not guarantee an optimal solution. It can converge upon local optima. It uses a greedy strategy by selecting the locally best attribute to split the dataset on each iteration. The algorithm's optimality can be improved by using backtracking during the search for the optimal decision tree at the cost of possibly taking longer.

ID3 can overfit the training data. To avoid overfitting, smaller decision trees should be preferred over larger ones. [*further explanation needed*] This algorithm usually produces small trees, but it does not always produce the smallest possible decision tree.

ID3 is harder to use on continuous data than on factored data (factored data has a discrete number of possible values, thus reducing the possible branch points). If the values of any given attribute are continuous, then there are many more places to split the data on this attribute, and searching for the best value to split by can be time-consuming.

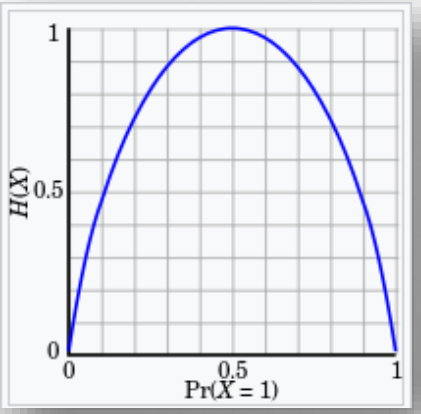


ID3-generated decision tree used to determine whether a particular nucleotide pair within a pre-mRNA sequence corresponds to an mRNA splice site. This tree has been shown to have a 95% correct prediction rate.[2]

Usage [edit]

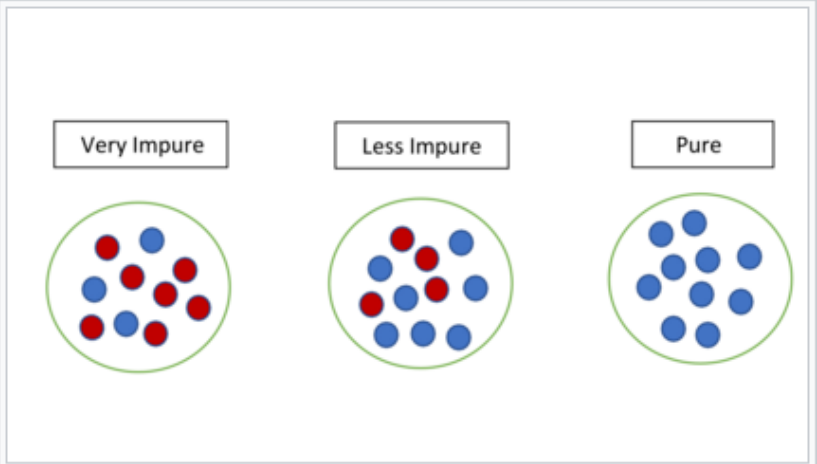
The ID3 algorithm is used by training on a data set S to produce a decision tree which is stored in memory. At runtime, this decision tree is used to classify new test cases (feature vectors) by traversing the decision tree using the features of the datum to arrive at a leaf node.

ENTROPIA E GANHO DE INFORMAÇÃO



Entropy $\mathrm{H}(X)$ (i.e. the expected surprisal) of a coin flip, measured in bits, graphed versus the bias of the coin $\Pr(X = 1)$, where $X = 1$ represents a result of heads.[10]:14–15

Here, the entropy is at most 1 bit, and to communicate the outcome of a coin flip (2 possible values) will require an average of at most 1 bit (exactly 1 bit for a fair coin). The result of a fair die (6 possible values) would have entropy $\log_2 6$ bits.



Entropy diagram[2]

Entropy [edit]

Entropy $\mathrm{H}(S)$ is a measure of the amount of uncertainty in the (data) set S (i.e. entropy characterizes the (data) set S).

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

Where,

- S – The current dataset for which entropy is being calculated
  - This changes at each step of the ID3 algorithm, either to a subset of the previous set in the case of splitting on an attribute or to a "sibling" partition of the parent in case the recursion terminated previously.
- X – The set of classes in S
- $p(x)$ – The proportion of the number of elements in class x to the number of elements in set S

When $\mathrm{H}(S) = 0$, the set S is perfectly classified (i.e. all elements in S are of the same class).

In ID3, entropy is calculated for each remaining attribute. The attribute with the **smallest** entropy is used to split the set S on this iteration. Entropy in information theory measures how much information is expected to be gained upon measuring a random variable; as such, it can also be used to quantify the amount to which the distribution of the quantity's values is unknown. A constant quantity has zero entropy, as its distribution is perfectly known. In contrast, a uniformly distributed random variable (discretely or continuously uniform) maximizes entropy. Therefore, the greater the entropy at a node, the less information is known about the classification of data at this stage of the tree; and therefore, the greater the potential to improve the classification here.

As such, ID3 is a greedy heuristic performing a best-first search for locally optimal entropy values. Its accuracy can be improved by preprocessing the data.

Information gain [edit]

Information gain $IG(A)$ is the measure of the difference in entropy from before to after the set S is split on an attribute A. In other words, how much uncertainty in S was reduced after splitting set S on attribute A.

$$IG(S, A) = \mathrm{H}(S) - \sum_{t \in T} p(t)\mathrm{H}(t) = \mathrm{H}(S) - \mathrm{H}(S|A).$$

Where,

- $\mathrm{H}(S)$ – Entropy of set S
- T – The subsets created from splitting set S by attribute A such that $S = \bigcup_{t \in T} t$
- $p(t)$ – The proportion of the number of elements in t to the number of elements in set S
- $\mathrm{H}(t)$ – Entropy of subset t

In ID3, information gain can be calculated (instead of entropy) for each remaining attribute. The attribute with the **largest** information gain is used to split the set S on this iteration.

ALGORITMO C4.5

C4.5 is an algorithm used to generate a decision tree developed by Ross Quinlan.[1] C4.5 is an extension of Quinlan's earlier ID3 algorithm. The decision trees generated by C4.5 can be used for classification, and for this reason, C4.5 is often referred to as a statistical classifier. In 2011, authors of the Weka machine learning software described the C4.5 algorithm as "a landmark decision tree program that is probably the machine learning workhorse most widely used in practice to date".[2]

It became quite popular after ranking #1 in the *Top 10 Algorithms in Data Mining* pre-eminent paper published by Springer LNCS in 2008.[3]

Algorithm [edit]

C4.5 builds decision trees from a set of training data in the same way as ID3, using the concept of information entropy. The training data is a set $S = s_1, s_2, \ldots$ of already classified samples. Each sample s_i consists of a p-dimensional vector $(x_{1,i}, x_{2,i}, \ldots, x_{p,i})$, where the x_j represent attribute values or features of the sample, as well as the class in which s_i falls.

At each node of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. The splitting criterion is the normalized information gain (difference in entropy). The attribute with the highest normalized information gain is chosen to make the decision. The C4.5 algorithm then recurses on the partitioned sublists.

This algorithm has a few base cases.

- All the samples in the list belong to the same class. When this happens, it simply creates a leaf node for the decision tree saying to choose that class.
- None of the features provide any information gain. In this case, C4.5 creates a decision node higher up the tree using the expected value of the class.
- Instance of previously unseen class encountered. Again, C4.5 creates a decision node higher up the tree using the expected value.

Pseudocode [edit]

In pseudocode, the general algorithm for building decision trees is:[4]

1. Check for the above base cases.
2. For each attribute *a*, find the normalized information gain ratio from splitting on *a*.
3. Let *a_best* be the attribute with the highest normalized information gain.
4. Create a decision *node* that splits on *a_best*.
5. Recurse on the sublists obtained by splitting on *a_best*, and add those nodes as children of *node*.

ALGORITMO C4.5

Improvements from ID3 algorithm [edit]

C4.5 made a number of improvements to ID3. Some of these are:

- Handling both continuous and discrete attributes - In order to handle continuous attributes, C4.5 creates a threshold and then splits the list into those whose attribute value is above the threshold and those that are less than or equal to it.[5]
- Handling training data with missing attribute values - C4.5 allows attribute values to be marked as ? for missing. Missing attribute values are simply not used in gain and entropy calculations.
- Handling attributes with differing costs.
- Pruning trees after creation - C4.5 goes back through the tree once it's been created and attempts to remove branches that do not help by replacing them with leaf nodes.

Improvements in C5.0/See5 algorithm [edit]

Quinlan went on to create C5.0 and See5 (C5.0 for Unix/Linux, See5 for Windows) which he markets commercially. C5.0 offers a number of improvements on C4.5. Some of these are:[6][7]

- Speed - C5.0 is significantly faster than C4.5 (several orders of magnitude)
- Memory usage - C5.0 is more memory efficient than C4.5
- Smaller decision trees - C5.0 gets similar results to C4.5 with considerably smaller decision trees.
- Support for boosting - Boosting improves the trees and gives them more accuracy.
- Weighting - C5.0 allows you to weight different cases and misclassification types.
- Winnowing - a C5.0 option automatically winnows the attributes to remove those that may be unhelpful.

Source for a single-threaded Linux version of C5.0 is available under the GNU General Public License (GPL).

ALGORITMO CART

1.10.7. Mathematical formulation

Given training vectors $x_i \in R^n$, i=1,..., l and a label vector $y \in R^l$, a decision tree recursively partitions the feature space such that the samples with the same labels or similar target values are grouped together.

Let the data at node m be represented by Q_m with n_m samples. For each candidate split $\theta = (j, t_m)$ consisting of a feature j and threshold t_m, partition the data into $Q_m^{left}(\theta)$ and $Q_m^{right}(\theta)$ subsets

$$Q_m^{left}(\theta) = \{(x, y) | x_j \leq t_m\}$$
$$Q_m^{right}(\theta) = Q_m \setminus Q_m^{left}(\theta)$$

The quality of a candidate split of node m is then computed using an impurity function or loss function $H()$, the choice of which depends on the task being solved (classification or regression)

$$G(Q_m, \theta) = \frac{n_m^{left}}{n_m} H(Q_m^{left}(\theta)) + \frac{n_m^{right}}{n_m} H(Q_m^{right}(\theta))$$

Select the parameters that minimises the impurity

$$\theta^* = \operatorname{argmin}_\theta G(Q_m, \theta)$$

Recurse for subsets $Q_m^{left}(\theta^*)$ and $Q_m^{right}(\theta^*)$ until the maximum allowable depth is reached, $n_m < \min_{samples}$ or $n_m = 1$.

1.10.7.1. Classification criteria

If a target is a classification outcome taking on values 0,1,...,K-1, for node m, let

$$p_{mk} = \frac{1}{n_m} \sum_{y \in Q_m} I(y = k)$$

be the proportion of class k observations in node m. If m is a terminal node, `predict_proba` for this region is set to p_{mk}. Common measures of impurity are the following.

Gini:

$$H(Q_m) = \sum_k p_{mk}(1 - p_{mk})$$

Log Loss or Entropy:

$$H(Q_m) = -\sum_k p_{mk} \log(p_{mk})$$

API SKLEARN

DecisionTreeClassifier

```
class sklearn.tree.DecisionTreeClassifier(*, criterion='gini', splitter='best',
max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features=None, random_state=None,
max_leaf_nodes=None, min_impurity_decrease=0.0, class_weight=None, ccp_alpha=0.0,
monotonic_cst=None)
```

[source]

A decision tree classifier.

Read more in the User Guide.

DICAS DE USO PRÁTICO

- Decision trees tend to overfit on data with a large number of features. Getting the right ratio of samples to number of features is important, since a tree with few samples in high dimensional space is very likely to overfit.
- Consider performing dimensionality reduction (PCA, ICA, or Feature selection) beforehand to give your tree a better chance of finding features that are discriminative.
- Understanding the decision tree structure will help in gaining more insights about how the decision tree makes predictions, which is important for understanding the important features in the data.
- Visualize your tree as you are training by using the `export` function. Use `max_depth=3` as an initial tree depth to get a feel for how the tree is fitting to your data, and then increase the depth.
- Remember that the number of samples required to populate the tree doubles for each additional level the tree grows to. Use `max_depth` to control the size of the tree to prevent overfitting.
- Use `min_samples_split` or `min_samples_leaf` to ensure that multiple samples inform every decision in the tree, by controlling which splits will be considered. A very small number will usually mean the tree will overfit, whereas a large number will prevent the tree from learning the data. Try `min_samples_leaf=5` as an initial value. If the sample size varies greatly, a float number can be used as percentage in these two parameters. While `min_samples_split` can create arbitrarily small leaves, `min_samples_leaf` guarantees that each leaf has a minimum size, avoiding low-variance, over-fit leaf nodes in regression problems. For classification with few classes, `min_samples_leaf=1` is often the best choice.

  Note that `min_samples_split` considers samples directly and independent of `sample_weight`, if provided (e.g. a node with m weighted samples is still treated as having exactly m samples). Consider `min_weight_fraction_leaf` or `min_impurity_decrease` if accounting for sample weights is required at splits.

- Balance your dataset before training to prevent the tree from being biased toward the classes that are dominant. Class balancing can be done by sampling an equal number of samples from each class, or preferably by normalizing the sum of the sample weights (`sample_weight`) for each class to the same value. Also note that weight-based pre-pruning criteria, such as `min_weight_fraction_leaf`, will then be less biased toward dominant classes than criteria that are not aware of the sample weights, like `min_samples_leaf`.
- If the samples are weighted, it will be easier to optimize the tree structure using weight-based pre-pruning criterion such as `min_weight_fraction_leaf`, which ensure that leaf nodes contain at least a fraction of the overall sum of the sample weights.
- All decision trees use `np.float32` arrays internally. If training data is not in this format, a copy of the dataset will be made.
- If the input matrix X is very sparse, it is recommended to convert to sparse `csc_matrix` before calling fit and sparse `csr_matrix` before calling predict. Training time can be orders of magnitude faster for a sparse matrix input compared to a dense matrix when features have zero values in most of the samples.

RANDOM FOREST

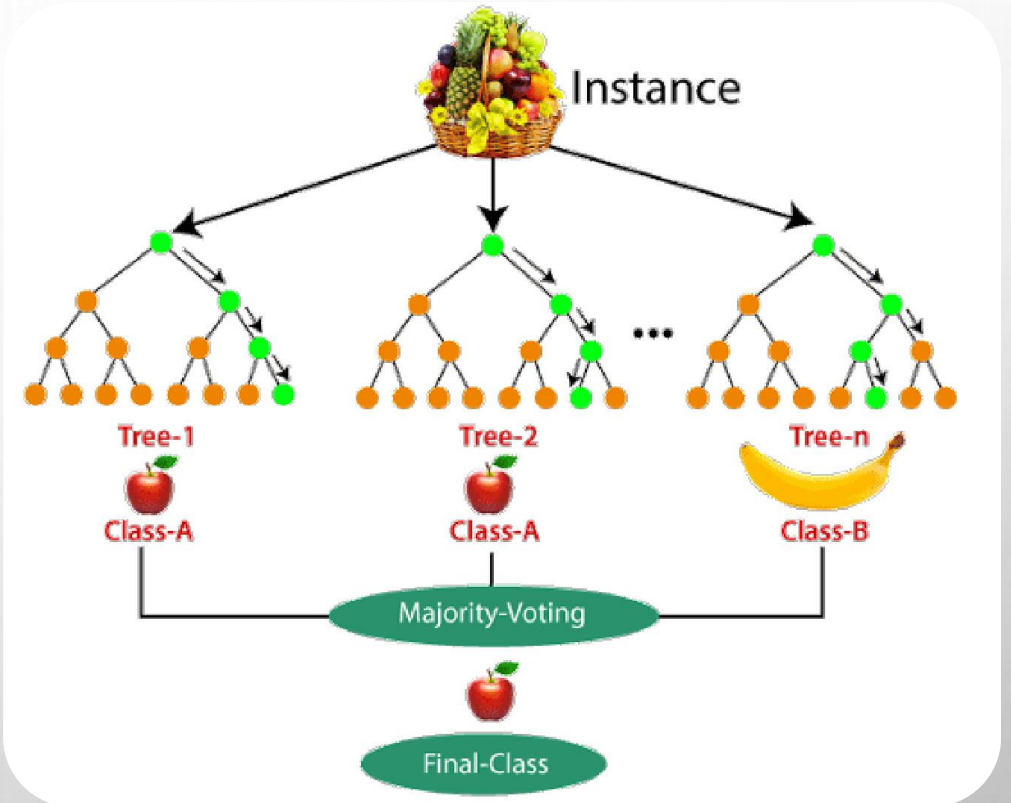
RANDOM FORESTS I

The `sklearn.ensemble` module includes two averaging algorithms based on randomized decision trees: the RandomForest algorithm and the Extra-Trees method. Both algorithms are perturb-and-combine techniques [B1998] specifically designed for trees. This means a diverse set of classifiers is created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers.

As other classifiers, forest classifiers have to be fitted with two arrays: a sparse or dense array X of shape `(n_samples, n_features)` holding the training samples, and an array Y of shape `(n_samples,)` holding the target values (class labels) for the training samples:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = RandomForestClassifier(n_estimators=10)
>>> clf = clf.fit(X, Y)
```

Like decision trees, forests of trees also extend to multi-output problems (if Y is an array of shape `(n_samples, n_outputs)`).



RANDOM FORESTS II

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 through an exhaustive search of the features values of either 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.

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.

A competitive alternative to random forests are Histogram-Based Gradient Boosting (HGBT) models:

- Building trees: Random forests typically rely on deep trees (that overfit individually) which uses much computational resources, as they require several splittings and evaluations of candidate splits. Boosting models build shallow trees (that underfit individually) which are faster to fit and predict.
- Sequential boosting: In HGBT, the decision trees are built sequentially, where each tree is trained to correct the errors made by the previous ones. This allows them to iteratively improve the model's performance using relatively few trees. In contrast, random forests use a majority vote to predict the outcome, which can require a larger number of trees to achieve the same level of accuracy.
- Efficient binning: HGBT uses an efficient binning algorithm that can handle large datasets with a high number of features. The binning algorithm can pre-process the data to speed up the subsequent tree construction (see Why it's faster). In contrast, the scikit-learn implementation of random forests does not use binning and relies on exact splitting, which can be computationally expensive.

Overall, the computational cost of HGBT versus RF depends on the specific characteristics of the dataset and the modeling task. It's a good idea to try both models and compare their performance and computational efficiency on your specific problem to determine which model is the best fit.

EXTREMELY RANDOMIZED TREES

In extremely randomized trees (see `ExtraTreesClassifier` and `ExtraTreesRegressor` classes), randomness goes one step further in the way splits are computed. As in random forests, a random subset of candidate features is used, but instead of looking for the most discriminative thresholds, thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule. This usually allows to reduce the variance of the model a bit more, at the expense of a slightly greater increase in bias:

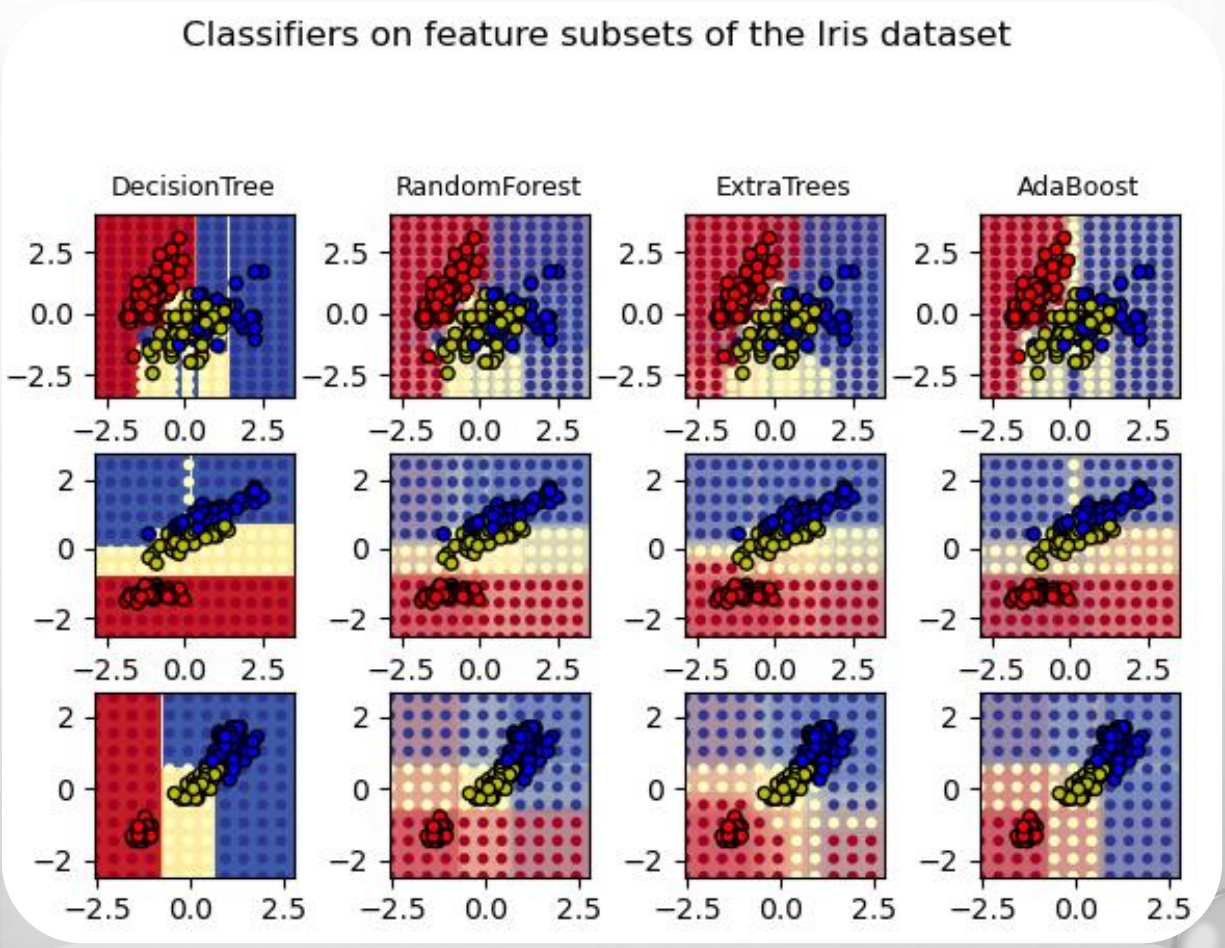
```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import make_blobs
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.tree import DecisionTreeClassifier

>>> X, y = make_blobs(n_samples=10000, n_features=10, centers=100,
...     random_state=0)

>>> clf = DecisionTreeClassifier(max_depth=None, min_samples_split=2,
...     random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.98...

>>> clf = RandomForestClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.999...

>>> clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean() > 0.999
True
```



TREE ENSEMBLES: PARÂMETROS

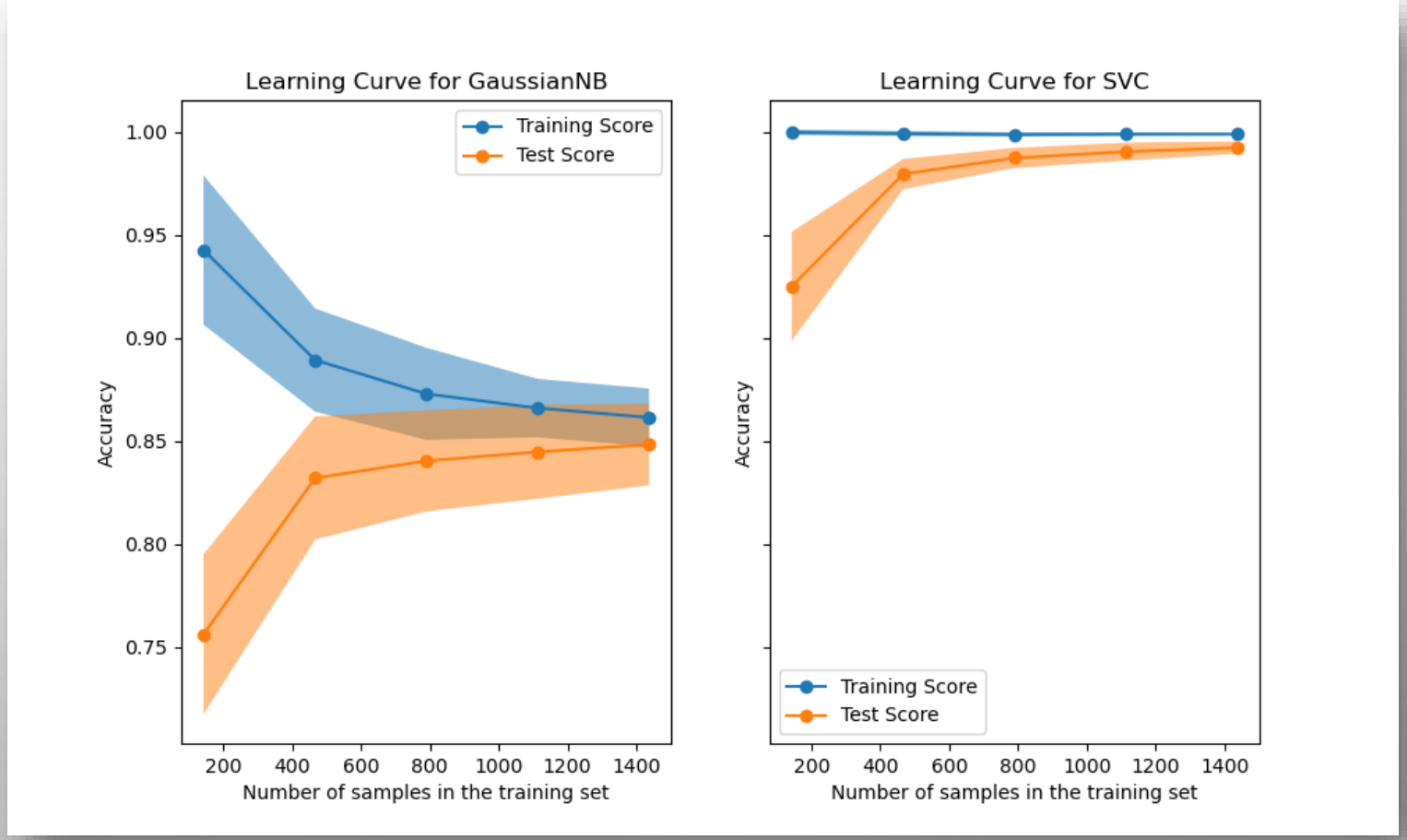
The main parameters to adjust when using these methods is `n_estimators` and `max_features`. The former is the number of trees in the forest. The larger the better, but also the longer it will take to compute. In addition, note that results will stop getting significantly better beyond a critical number of trees. The latter is the size of the random subsets of features to consider when splitting a node. The lower the greater the reduction of variance, but also the greater the increase in bias. Empirical good default values are `max_features=1.0` or equivalently `max_features=None` (always considering all features instead of a random subset) for regression problems, and `max_features="sqrt"` (using a random subset of size `sqrt(n_features)`) for classification tasks (where `n_features` is the number of features in the data). The default value of `max_features=1.0` is equivalent to bagged trees and more randomness can be achieved by setting smaller values (e.g. 0.3 is a typical default in the literature). Good results are often achieved when setting `max_depth=None` in combination with `min_samples_split=2` (i.e., when fully developing the trees). Bear in mind though that these values are usually not optimal, and might result in models that consume a lot of RAM. The best parameter values should always be cross-validated. In addition, note that in random forests, bootstrap samples are used by default (`bootstrap=True`) while the default strategy for extra-trees is to use the whole dataset (`bootstrap=False`). When using bootstrap sampling the generalization error can be estimated on the left out or out-of-bag samples. This can be enabled by setting `oob_score=True`.

Note

The size of the model with the default parameters is $O(M * N * log(N))$, where M is the number of trees and N is the number of samples. In order to reduce the size of the model, you can change these parameters: `min_samples_split`, `max_leaf_nodes`, `max_depth` and `min_samples_leaf`.

EVALUATION

CURVA DE APRENDIZADO



PRODUÇÃO