Derivation of Softmax, k-NN

Milan Straka

November 02, 2019

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Given a function $f(\boldsymbol{x})$, we can find a maximum with respect to a vector $\boldsymbol{x} \in \mathbb{R}^d$, by investigating the critical points $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = 0$.

Consider now finding maximum subject to a constraint $g(\boldsymbol{x}) = 0$.

- Note that $\nabla_{\boldsymbol{x}} g(\boldsymbol{x})$ is orthogonal to the surface of the constraint, because if $\boldsymbol{x}$ and a nearby point $\boldsymbol{x} + \boldsymbol{\varepsilon}$ lie on the surface, from the Taylor expansion $g(\boldsymbol{x} + \boldsymbol{\varepsilon}) \approx g(\boldsymbol{x}) + \boldsymbol{\varepsilon}^T \nabla_{\boldsymbol{x}} g(\boldsymbol{x})$ we get $\boldsymbol{\varepsilon}^T \nabla_{\boldsymbol{x}} g(\boldsymbol{x}) \approx 0$.

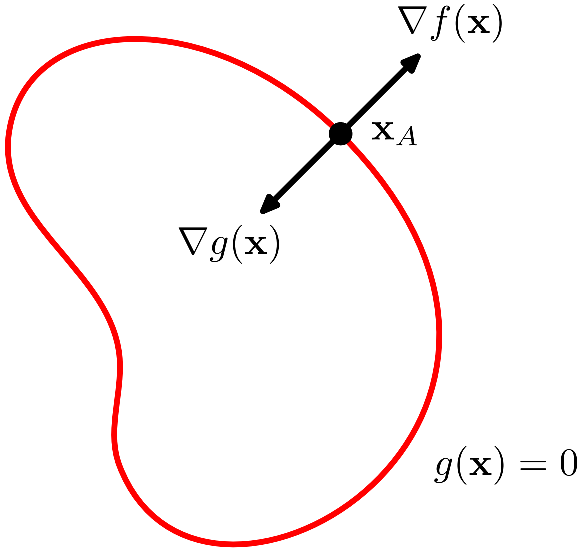


Figure E.1 of Pattern Recognition and Machine Learning.

- In the seeked maximum, $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ must also be orthogonal to the constraint surface (or else moving in the direction of the derivative would increase the value).
- Therefore, there must exist λ such that $\nabla_{\boldsymbol{x}} f + \lambda \nabla_{\boldsymbol{x}} g = 0$.

We therefore introduce the *Lagrangian function*

$$L(\boldsymbol{x}, \lambda) \stackrel{\text{def}}{=} f(\boldsymbol{x}) + \lambda g(\boldsymbol{x}).$$

We can then find the maximum under the constraint by inspecting critical points of $L(\boldsymbol{x}, \lambda)$ with respect to both $\boldsymbol{x}$ and λ:

- $\frac{\partial L}{\partial \lambda} = 0$ leads to $g(\boldsymbol{x}) = 0$;
- $\frac{\partial L}{\partial \boldsymbol{x}} = 0$ is the previously derived $\nabla_{\boldsymbol{x}} f + \lambda \nabla_{\boldsymbol{x}} g = 0$.

If there are multiple equality constraints, we can use induction; therefore, every constraint gets its own λ.

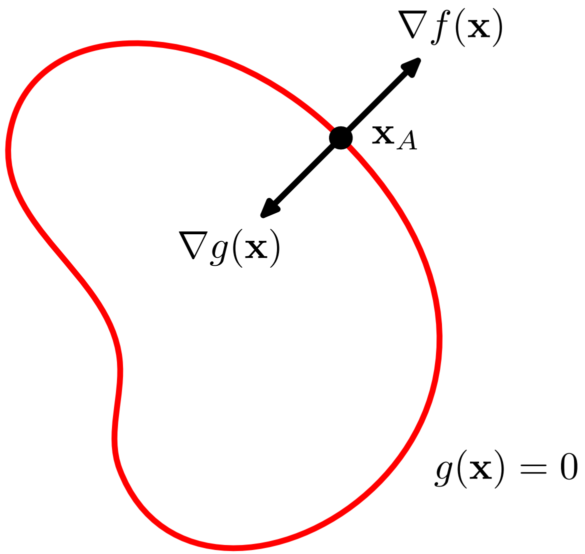


Figure E.1 of Pattern Recognition and Machine Learning.

Calculus of Variations

Many optimization techniques depend on minimizing a function $J(\boldsymbol{w})$ with respect to a vector $\boldsymbol{w} \in \mathbb{R}^d$, by investigating the critical points $\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = 0$.

A function of a function, $J[f]$, is known as a **functional**, for example entropy $H[\cdot]$.

Similarly to partial derivatives, we can take **functional derivatives** of a functional $J[f]$ with respect to individual values $f(\boldsymbol{x})$ for all points $\boldsymbol{x}$. The functional derivative of J with respect to a function f in a point $\boldsymbol{x}$ is denoted as

$$\frac{\partial}{\partial f(\boldsymbol{x})} J.$$

For this course, we use only the following theorem stating that for all differentiable functions f and differentiable functions $g\big(y = f(\boldsymbol{x}), \boldsymbol{x}\big)$ with continuous derivatives, it holds that

$$\frac{\partial}{\partial f(\boldsymbol{x})} \int g\big(f(\boldsymbol{x}'), \boldsymbol{x}'\big) \,\mathrm{d}\boldsymbol{x}' = \frac{\partial}{\partial y} g(y, \boldsymbol{x}).$$

Calculus of Variations

An intuitive view is to think about $f(\boldsymbol{x})$ as a vector of uncountably many elements (for every value $\boldsymbol{x}$). In this interpretation the result is analogous to computing partial derivatives of a vector $\boldsymbol{w} \in \mathbb{R}^d$:

$$\frac{\partial}{\partial w_i} \sum_j g(w_j, \boldsymbol{x}) = \frac{\partial}{\partial w_i} g(w_i, \boldsymbol{x}).$$

$$\frac{\partial}{\partial f(\boldsymbol{x})} \int g\big(f(\boldsymbol{x}'), \boldsymbol{x}'\big)\,\mathrm{d}\boldsymbol{x}' = \frac{\partial}{\partial y} g(y, \boldsymbol{x}).$$

Function with Maximum Entropy

What distribution over $\mathbb{R}$ maximizes entropy $H[p] = -\mathbb{E}_x[\log p(x)]$?

For continuous values, the entropy is an integral $H[p] = -\int p(x) \log p(x) \,\mathrm{d}x$.

We cannot just maximize H with respect to a function p, because:

- the result might not be a probability distribution – we need to add a constraint that $\int p(x) \,\mathrm{d}x = 1$;
- the problem is underspecified because a distribution can be shifted without changing entropy – we add a constraint $\mathbb{E}[x] = \mu$;
- because entropy increases as variance increases, we ask which distribution with a *fixed* variance σ^2 has maximum entropy – adding a constraint $\mathrm{Var}(x) = \sigma^2$.

Function with Maximum Entropy

Lagrangian $L(p(x), x, \boldsymbol\lambda; \mu, \sigma^2)$ of all the constraints and the entropy function is

$$L = \lambda_1 \Big(\int p(x) \,\mathrm{d}x - 1 \Big) + \lambda_2 \big(\mathbb{E}[x] - \mu \big) + \lambda_3 \big(\operatorname{Var}(x) - \sigma^2 \big) + H[p].$$

By expanding all definitions to integrals, we get

$$\begin{aligned} L(p(x), x, \boldsymbol\lambda; \mu, \sigma^2) = \int \Big(& \lambda_1 p(x) + \lambda_2 p(x) x + \lambda_3 p(x) (x - \mu)^2 - p(x) \log p(x) \Big) \,\mathrm{d}x - \\ & - \lambda_1 - \mu \lambda_2 - \sigma^2 \lambda_3. \end{aligned}$$

The functional derivative of L is:

$$\frac{\partial}{\partial p(x)} L(p(x), x, \boldsymbol\lambda; \mu, \sigma^2) = \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 - \log p(x) = 0.$$

Function with Maximum Entropy

Rearranging the functional derivative of L:

$$\frac{\partial}{\partial p(x)} L(p(x), x, \boldsymbol\lambda; \mu, \sigma^2) = \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 - \log p(x) = 0.$$

we obtain

$$p(x) = \exp\Big(\lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1\Big).$$

We can verify that setting $\lambda_1 = 1 - \log \sigma\sqrt{2\pi}$, $\lambda_2 = 0$ and $\lambda_3 = -1/(2\sigma^2)$ fulfils all the constraints, arriving at

$$p(x) = \mathcal{N}(x; \mu, \sigma^2).$$

Derivation of Softmax using Maximum Entropy

Let $\boldsymbol{X} = \{(\boldsymbol{x}_1, t_1), (\boldsymbol{x}_2, t_2), \ldots, (\boldsymbol{x}_N, t_N)\}$ be training data of a K-class classification, with $\boldsymbol{x}_i \in \mathbb{R}^D$ and $t_i \in \{1, 2, \ldots, K\}$.

We want to model it using a function $\pi : \mathbb{R}^D \rightarrow \mathbb{R}^K$ so that $\pi(\boldsymbol{x})$ gives a distribution of classes for input $\boldsymbol{x}$.

We impose the following conditions on π:

- $\forall\, 1 \leq k \leq K : \pi(\boldsymbol{x})_k \geq 0,$
- $\displaystyle\sum_{k=1}^{K} \pi(\boldsymbol{x})_k = 1,$
- $\forall\, 1 \leq j \leq D, \forall\, 1 \leq k \leq K : \displaystyle\sum_{i=1}^{N} \pi(\boldsymbol{x}_i)_k x_{i,j} = \sum_{i=1}^{N} \Big[t_i == k\Big] x_{i,j}.$

Derivation of Softmax using Maximum Entropy

There are many such π, one particularly bad is

$$\pi(\boldsymbol{x}) = \begin{cases} \mathbf{1}_{t_i} & \text{if there exists } i : \boldsymbol{x}_i = \boldsymbol{x}, \\ \mathbf{1}_0 & \text{otherwise,} \end{cases}$$

where $\mathbf{1}_i$ is a vector of zeros, except for position i, which is equal to 1.

Therefore, we want to find a more **general** π – consequently, we turn to the principle of maximum entropy and search for π with maximum entropy.

We want to maximize $-\sum_{i=1}^N \sum_{k=1}^K \pi(\boldsymbol{x}_i)_k \log(\pi(\boldsymbol{x}_i)_k)$ given

- $\forall\, 1 \leq i \leq N, \forall\, 1 \leq k \leq K : \pi(\boldsymbol{x}_i)_k \geq 0$,
- $\forall\, 1 \leq i \leq N : \sum_{k=1}^K \pi(\boldsymbol{x}_i)_k = 1$,
- $\forall\, 1 \leq j \leq D, \forall\, 1 \leq k \leq K : \sum_{i=1}^N \pi(\boldsymbol{x}_i)_k x_{i,j} = \sum_{i=1}^N \big[t_i == k\big] x_{i,j}$.

We therefore form a Lagrangian (ignoring the first inequality constraint):

$$
\begin{aligned}
L = & \sum_{j=1}^D \sum_{k=1}^K \lambda_{j,k} \Big(\sum_{i=1}^N \pi(\boldsymbol{x}_i)_k x_{i,j} - \big[t_i == k\big] x_{i,j} \Big) \\
& + \sum_{i=1}^N \beta_i \Big(\sum_{k=1}^K \pi(\boldsymbol{x}_i)_k - 1 \Big) \\
& - \sum_{i=1}^N \sum_{k=1}^K \pi(\boldsymbol{x}_i)_k \log(\pi(\boldsymbol{x}_i)_k).
\end{aligned}
$$

Derivation of Softmax using Maximum Entropy

We now compute partial derivatives of the Lagrangian, notably the values

$$\frac{\partial}{\partial \pi(\boldsymbol{x}_i)_k} L.$$

We arrive at

$$\frac{\partial}{\partial \pi(\boldsymbol{x}_i)_k} L = \boldsymbol{x}_i^T \boldsymbol{\lambda}_{*,k} + \beta_i - \log(\pi(\boldsymbol{x}_i)_k) - 1.$$

Setting the Lagrangian to zero, we get $\boldsymbol{x}_i^T \boldsymbol{\lambda}_{*,k} + \beta_i - \log(\pi(\boldsymbol{x}_i)_k) - 1 = 0$, which we rewrite to

$$\pi(\boldsymbol{x}_i)_k = e^{\boldsymbol{x}_i^T \boldsymbol{\lambda}_{*,k} + \beta_i - 1}.$$

Such a forms guarantees $\pi(\boldsymbol{x}_i)_k > 0$, which we did not include in the conditions.

Derivation of Softmax using Maximum Entropy

In order to find out the β_i values, we turn to the constraint

$$\sum_k \pi(\boldsymbol{x}_i)_k = \sum_k e^{\boldsymbol{x}_i^T \boldsymbol{\lambda}_{*,k} + \beta_i - 1} = 1,$$

from which we get

$$e^{\beta_i} = \frac{1}{\sum_k e^{\boldsymbol{x}_i^T \boldsymbol{\lambda}_{*,k} - 1}},$$

yielding

$$\pi(\boldsymbol{x}_i)_k = e^{\boldsymbol{x}_i^T \boldsymbol{\lambda}_{*,k} + \beta_i - 1} = \frac{e^{\boldsymbol{x}_i^T \boldsymbol{\lambda}_{*,k}}}{\sum_{k'} e^{\boldsymbol{x}_i^T \boldsymbol{\lambda}_{*,k'}}} = \operatorname{softmax}(\boldsymbol{x}_i \boldsymbol{\lambda})_k.$$

When evaluating binary classification, we have used **accuracy** so far.

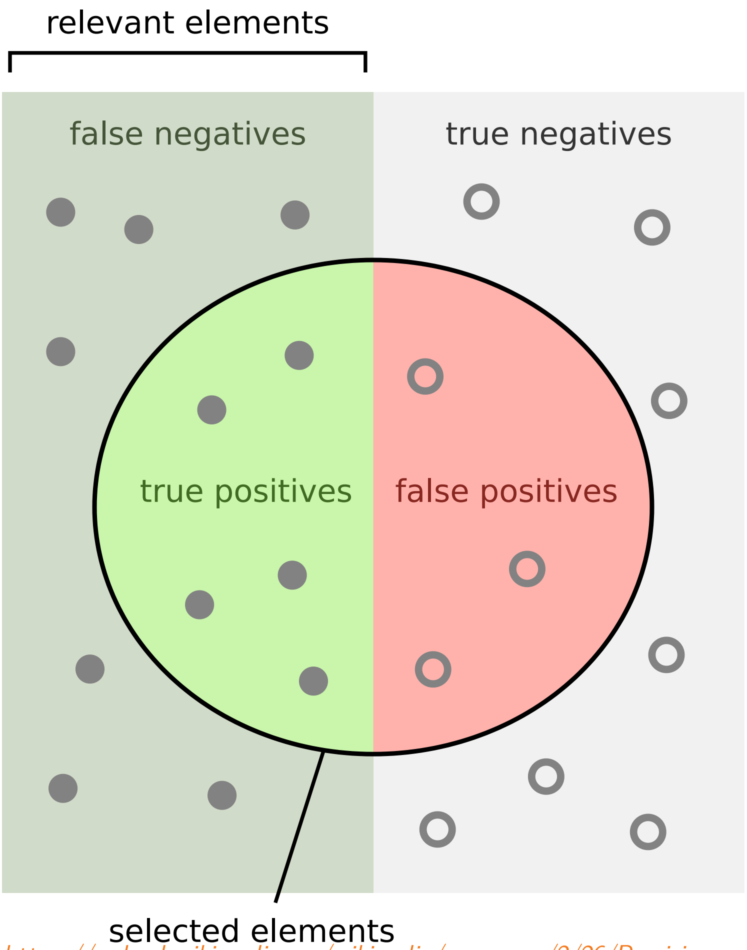
However, there are other metric we might want to consider. One of them is F_1-score.

Consider the following **confusion matrix**:

	Target positive	**Target negative**
Predicted positive	**True Positive (TP)**	**False Positive (FP)**
Predicted negative	**False Negative (FN)**	**True Negative (TN)**

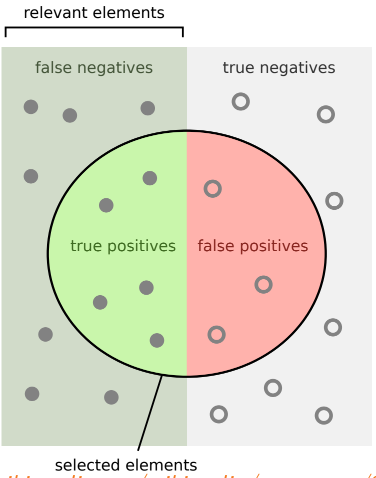
Accuracy can be computed as

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$



https://upload.wikimedia.org/wikipedia/commons/2/26/Precisionrecall.svg

	Target positive	Target negative
Predicted positive	**True Positive (TP)**	**False Positive (FP)**
Predicted negative	**False Negative (FN)**	**True Negative (TN)**



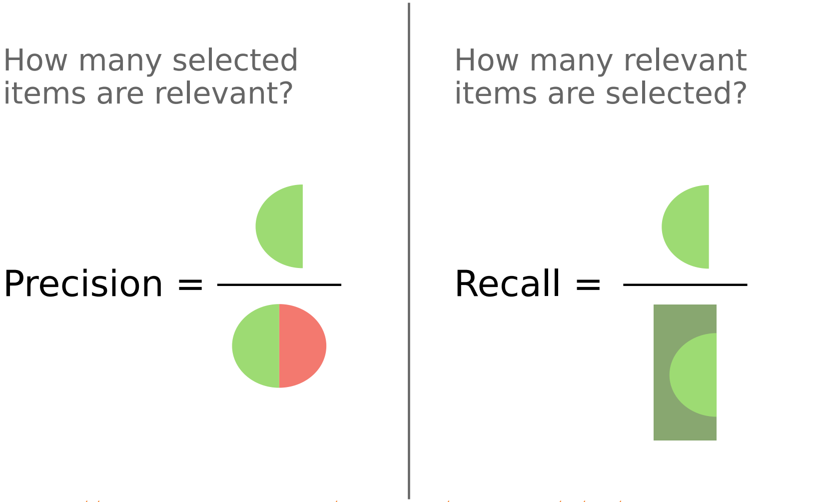
https://upload.wikimedia.org/wikipedia/commons/2/26/Precisionrecall.svg

In some cases, we are mostly interested in positive examples.

We define **precision** (percentage of correct predictions in predicted examples) and **recall** (percentage of correct predictions in the gold examples) as

$$\text{precision} = \frac{TP}{TP + FP},$$

$$\text{recall} = \frac{TP}{TP + FN}.$$

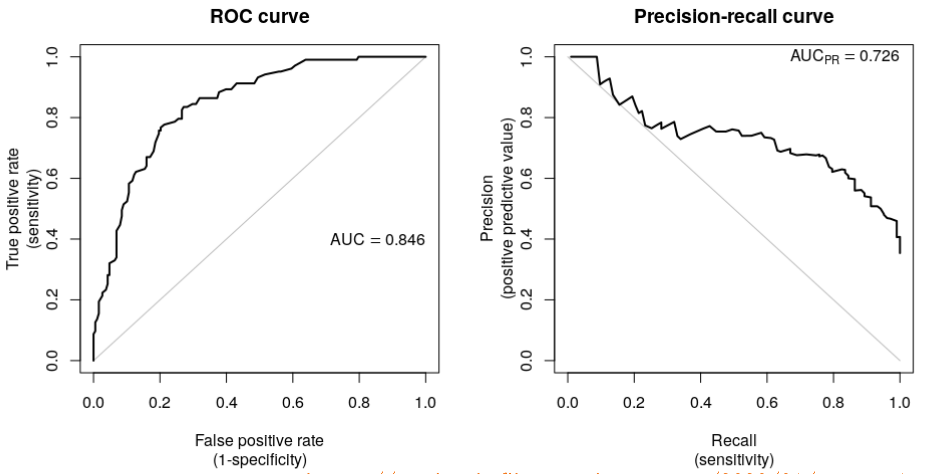


https://upload.wikimedia.org/wikipedia/commons/2/26/Precisionrecall.svg

The precision and recall go "against each other": increasing the classifier threshold usually increases recall and decreases precision, and vice versa.

We therefore define a single $\boldsymbol{F_1}$**-score** as a harmonic mean of precision and recall:

$$F_1 = \frac{2}{\text{precision}^{-1} + \text{recall}^{-1}} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{TP \qquad + \qquad TP}{TP + FP + TP + FN}.$$



https://modtools.files.wordpress.com/2020/01/roc_pr-1.png

General F_β-score

The F_1 score can be generalized to F_β score, which can be used as a metric when recall is β times more important than precision; F_2 favoring recall and $F_{0.5}$ favoring precision are commonly used.

The formula for F_β is

$$\begin{aligned} F_\beta &= \frac{1+\beta^2}{\mathrm{precision}^{-1} + \beta^2 \mathrm{recall}^{-1}} \\ &= \frac{(1+\beta^2)\cdot \mathrm{precision} \cdot \mathrm{recall}}{\beta^2 \cdot \mathrm{precision} + \mathrm{recall}} \\ &= \frac{\mathit{TP} \quad + \quad \beta^2\, \mathit{TP}}{\mathit{TP} + \mathit{FP} + \beta^2 (\mathit{TP} + \mathit{FN})}. \end{aligned}$$

General F_β-score

You may wonder why is β^2 used in the formula

$$F_\beta = \frac{1 + \beta^2}{\mathrm{precision}^{-1} + \beta^2 \mathrm{recall}^{-1}}$$

instead of just β.

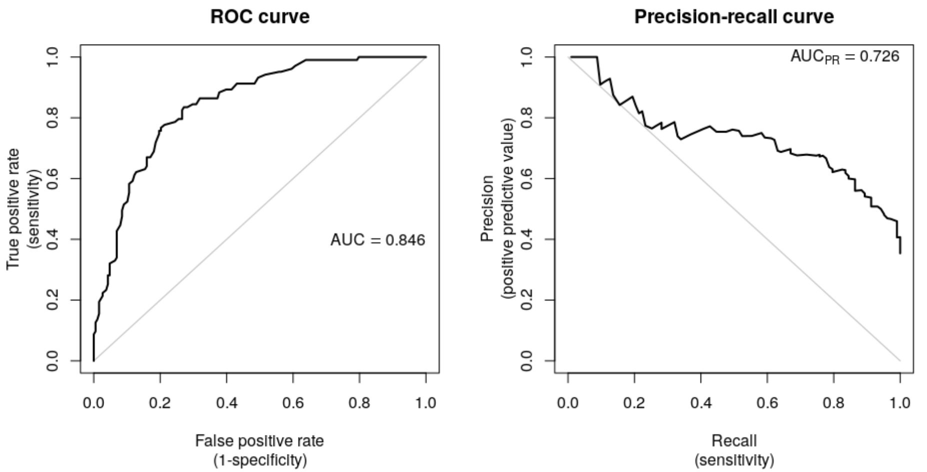
Quoting C. J. van Rijsbergen from his book *Information Retrieval*, 1979:

> *What we want is therefore a parameter β to characterise the measurement function in such a way that we can say: it measures the effectiveness of retrieval with respect to a user who attaches β times as much importance to recall as precision. The simplest way I know of quantifying this is to specify the* $\mathrm{recall}/\mathrm{precision}$ *ratio at which the user is willing to trade an increment in precision for an equal loss in recall.*

It is straightforward to verify that indeed $\frac{\partial F_\beta}{\partial \mathrm{precision}} = \frac{\partial F_\beta}{\partial \mathrm{recall}}$ implies $\frac{\mathrm{recall}}{\mathrm{precision}} = \beta$.

Changing the threshold in logistic regression allows us to trade off precision for recall and vice versa. Therefore, we can tune it on the development set to achieve highest possible F_1 score, if required.

Also, if we want to evaluate F_1-score without considering a specific threshold, the **area under curve** (AUC) is sometimes used as a metric.



https://modtools.files.wordpress.com/2020/01/roc_pr-1.png

F_1-Score in Multiclass Classification

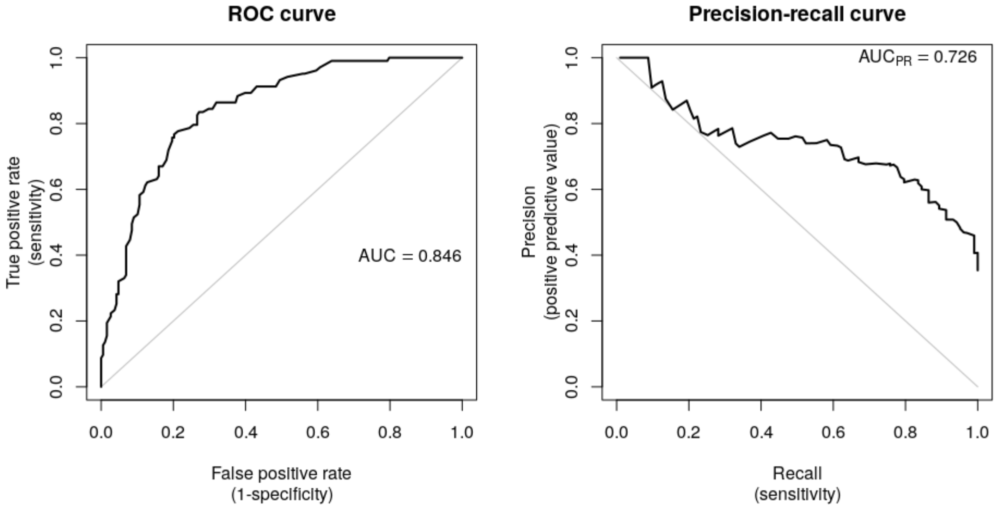
To extend F_1-score to multiclass classification, we expect one of the classes to be *negative* and the others *different kinds of positive*. For each of the positive classes, we compute the same confusion matrix as in the binary case (considering all other labels as negative ones), and then combine the results in one of the following ways:

- **micro-averaged** F_1 (or just **micro** F_1): we first sum all the TP, FP and FN of the individual binary classifications and compute the final F_1-score (this way, the frequency of the individual classes is taken into account);
- **macro-averaged** F_1 (or just **macro** F_1): we first compute the F_1-scores of the individual binary classifications and then compute an unweighted average (therefore, the frequency of the classes is ignored).

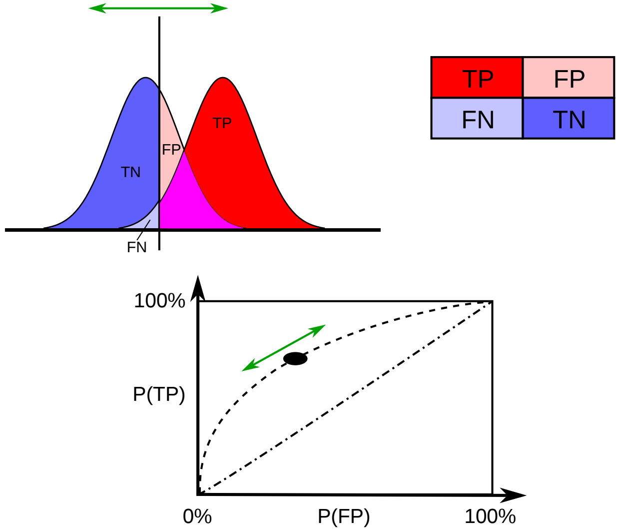
The precision-recall curve is useful when we are interested in the positive examples (i.e., we are ignoring true negative instances). In case we want to consider also the true negatives, we might instead use the **Receiver Operating Characteristic** (ROC) curve.

In the ROC curve, we consider two measures of a binary classifier under changing threshold:

- **true positive rate** or **sensitivity** (probability of detection): $\frac{TP}{\text{target positives}} = \frac{TP}{TP+FN}$;
- **false positive rate** or **1-specificity** (probability of false alarm): $\frac{FP}{\text{target negatives}} = \frac{FP}{FP+TN}$;



https://modtools.files.wordpress.com/2020/01/roc_pr-1.png



https://upload.wikimedia.org/wikipedia/commons/4/4f/ROC_curves.svg

	Target positive	Target negative	
Predicted positive	**True Positive (TP)**	**False Positive (FP)** Type I Error	**precision** $\frac{TP}{TP+FP}$
Predicted negative	**False Negative (FN)** Type II Error	**True Negative (TN)**	
	true positive rate, **recall**, **sensitivity** $\frac{TP}{TP+FN}$	**false positive rate** $\frac{FP}{FP+TN}$ **specificity** $\frac{TN}{TN+FP}$	

- F_1-score $= \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{TP \quad + \quad TP}{TP+FP+TP+FN}$
- accuracy $= \frac{TP+TN}{TP+FP+FN+TN}$

Parametric and Nonparametric Models

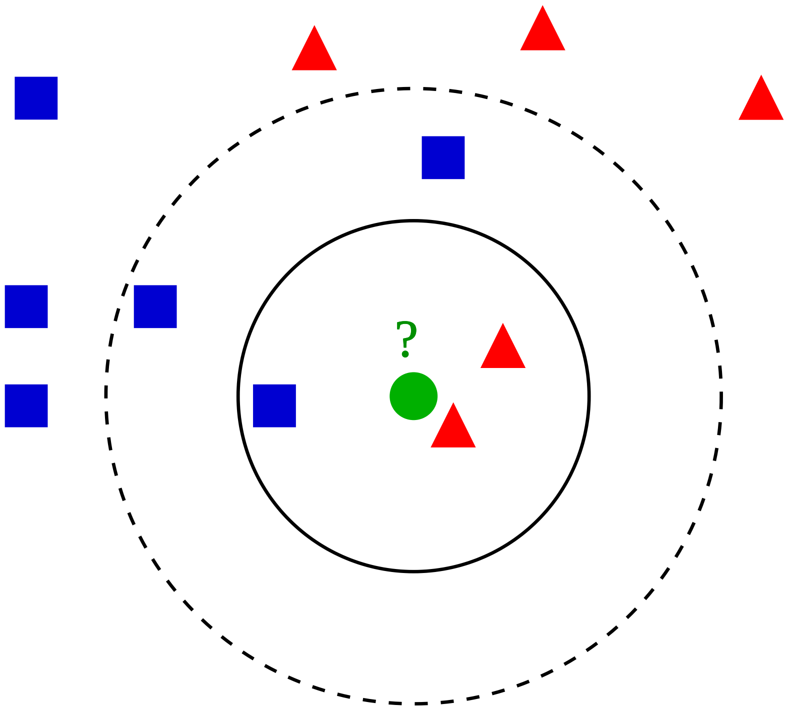
All the machine learning models which we discussed so far are **parametric**, because they use a *fixed* number of parameters (usually depending on the number of features, K for multiclass classification, hidden layer in MLPs, …).

However, there also exist **nonparametric** models. Even if the name seems to suggest they do not have any parameters, they have a non-fixed number of parameters, because the number of parameters usually depend on the size of the training data – therefore, the model size usually grows with the size of the training data.

A simple but sometimes effective nonparametric method for both classification and regression is $\boldsymbol{k}$**-nearest neighbors** algorithm.

The training phase of the k-nearest neighbors algorithm is trivial, it consists of only storing the whole train set (the so-called **lazy learning**).

For a given test example, the main idea is to use the targets of the most similar training data to perform the prediction.



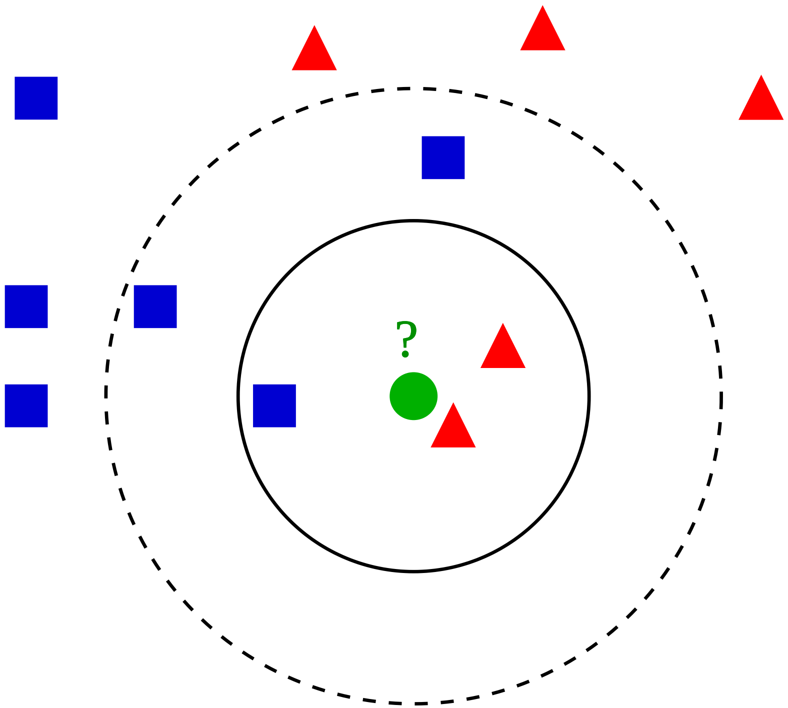
https://upload.wikimedia.org/wikipedia/commons/e/e7/KnnClassification.svg

k-Nearest Neighbors

Several hyperparameters influence the behaviour of the prediction phase:

- **k**: consider k most similar training examples (higher k usually decrease variance, but increase bias);
- **metric**: a function used to find the nearest neighbors; common choices are metrics based on L_p norms (with usual values of p being 1, 2, 3, ∞). For $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^D$, the distance is measured as $\|\boldsymbol{x}-\boldsymbol{y}\|_p$, where

$$\|\boldsymbol{x}\|_p = \Big(\sum\nolimits_i |x_i|^p\Big)^{1/p};$$



https://upload.wikimedia.org/wikipedia/commons/e/e7/KnnClassification.svg

- **weights**: optionally, more similar examples can be considered with bigger weights:
  - *uniform*: all k nearest neighbors are considered equally;
  - *inverse*: the weight of an example is proportional to the inverse of distance;
  - *softmax*: the weights are proportional to $\operatorname{softmax}$ of negative distances.

Regression

To perform regression when k nearest neighbors have values t_i and weights w_i, we predict

$$t = \sum_i \frac{w_i}{\sum_j w_j} \cdot t_i.$$

Classification

For uniform weights, we can use **voting** during prediction – the most frequent class is predicted (with ties broken arbitrarily).

Otherwise, we weight the categorical distributions $\boldsymbol{t}_i \in R^K$ (with the training target classes represented using one-hot encoding), predicting a distribution

$$\boldsymbol{t} = \sum_i \frac{w_i}{\sum_j w_j} \cdot \boldsymbol{t}_i.$$

The predicted class is then the one with largest probability, i.e., $\arg\max_k \sum_i w_i t_{i,k}$.

k-Nearest Neighbors

A trivial implementation of the k-nearest neighbors algorithm is extremely demanding during the inference, requiring to measure distances of a given example to all training instances.

However, several data structures capable of speeding up the k-nearest neighbor search exist, like

- k-d trees, which allow both a static or dynamic construction and can perform nearest neighbor queries of uniformly random points in logarithmic time on average, but which become inefficient for high-dimensional data;
- ball trees, R-trees, …