Statistical methods for machine learning

Francesco Tomaselli

March 9, 2021

Contents

1 Supervised classification

1.1 General task

This first section introduces the tasks that are solvable with machine learning.

Machine learning task Typically a machine learning task falls in the three categories below.

1. *Clustering*: group data according to similarity, e.g. group customers by shopping habits
2. *Classification*: predict semantic labels associated with data points, for instance document classification in relation to topics
3. *Planning*: decide which set of actions to be performed to achieve a certain goal, e.g. self driving cars

When it comes to machine learning there are mainly two learning paradigms.

Supervised learning This type of learning relies on semantic tagging of data. This usually solves classification tasks, as we can assign a label to each data point and learn patterns to classify new data.

Unsupervised learning There is no semantic tagging associated with data, this can for instance solve a clustering problem, as the algorithm will consider a form of similarity between data points to cluster them. Similarity can be interpreted as a semantic feature of data, but no explicit label is given.

1.2 Elements

A supervised task is made of many elements, for instance, we need a label definition, actual data points to observe and a measure of the performance and the error to guide the algorithm.

1.2.1 Labels

Along with the definition of a learning problem, there is a label set Y that collects possible labels of data.
For instance, when considering classification of documents, a label set could be defined as follows:

$$Y = \{sport, politics, business, \dots\}$$

Or in the case of stock predictions:

$$Y \subseteq \mathbb{R}$$

Labels definition The definition of labels changes the flavour of the task, in fact:

- if Y contains a finite number of symbols the task is called *classification/categorization*
- if Y is formed by real numbers, the task is called *regression*

Error measures The computation of prediction error differs between classification and regression: in the first we can consider an error if the prediction differs from the label, in the latter we can compute the difference between the prediction and the label.

1.2.2 Loss functions

When learning a map from data to labels, we need a way to tell the machine how good the mapping is, so a loss function is defined on the pair of true label and assigned label.

Classification loss examples In case of classification, we can define:

$$l(y, \hat{y}) = \begin{cases} 0 \ if \ y = \hat{y} \\ 1 \ if \ y \neq \hat{y} \end{cases}$$

It is possible to be a little more precise in the definition of a loss function, for instance:

$$Y = \{spam, notspam\}$$

$$l(y, \hat{y}) = \begin{cases} 2 \ if \ y = notspam \wedge \hat{y} = spam \\ 1 \ if \ y = spam \wedge \hat{y} = notspam \\ 0 \ otherwise \end{cases}$$

The idea is to penalize false positive mistakes, giving them a 2 contribution, and to count false negative mistakes.

Regression loss examples An example of a loss function in the case of regression are the *absolute loss*:

$$l(y, \hat{y}) = |y - \hat{y}|$$

ot the *square loss* that has some better properties:

$$l(y, \hat{y}) = (y - \hat{y})^2$$

Another example, in the case of whether forecast prediction:

$$Y = \{rain, sun\}, Z = [0, 1]$$

We want to define a regression task that outputs the probability of a whether condition. By using the absolute error loss function, we assume a linearity in the error, whereas we can assume that predicting sun while it rains can be a shame.
This means it is preferred to use something like a square loss, to keep track of such a wanted behavior, indeed the error would raise quicker and exponentially.

Another example would be the *logarithmic loss*:

$$l(y, \hat{y}) = \begin{cases} \ln \frac{1}{\hat{y}} \ if \ y = 1 \\ \ln \frac{1}{1-\hat{y}} \ if \ y = 0 \end{cases}$$

$$\lim_{\hat{y} \to 0^+} l(1, \hat{y}) = \lim_{\hat{y} \to 1^-} l(0, \hat{y}) = \infty$$

This means the algorithm will pay an infinite punishment when making wrong predictions.

1.2.3 Data points

The data points encode individual data, such as individual documents, medical records, measurements, and so on. They are digitally stored somewhere.
We define with X the data domain.

Encoding To not loose generality, we would data to *look the same* to a learning algorithm, so it is preferred to have some sort of encoding. Such encoding usually maps X to a vector of numbers.
For instance images can be represented as vectors of pixels, and texts can use something like a frequency vector given a dictionary.

The power of encoding is that after a mapping data points become points in a space, so we can use geometry to face the machine learning task.
Let's consider classification, with a good encoding, usually points related to each other, so points of the same class, are close together.

Data points can be

$$X = \begin{cases} \mathbb{R}^d \ numerical \ attributes \\ X_1, \ldots, X_d \ categorical \ attributes \end{cases}$$

Problems with numerical attributes A geometric interpretation for such problems is straight forward, as we can use data points as coordinates.

Problems with categorical attributes If a certain attribute does not have a geometrical interpretation, for instance there is no order on the domain of the attribute, it is helpful to use *One-hot encoding*. For instance, to use the sex of a person as a dimension, we must define an order and a mapping to numbers.

Remark. *When using encoding, the created order over categorical attributes can be a problem. There is no golden rule to approach an encoding task, it is a trial and error process, a bad encoding can destroy data meaning.*

1.2.4 Learning predictors

A predictor is a function that maps data points to labels, or predictions if they differ from labels

$$f : X \longrightarrow Y, f : X \longrightarrow Z, Z \neq Y$$

So given a data point x, the prediction is made as follows

$$\hat{y} = f(x)$$

The prediction function must be learned, and the general goal is to have a small loss $l(y, \hat{y})$ over *most* of the data points.

Data points annotation As we are in a supervised task, we can construct a predictor using labeled data. A single data point is a tuple

$$(x, y) \longrightarrow (\textit{data point}, \textit{label})$$

When talking about labels, they can be:

1. *Subjective*: human annotation
2. *Objective*: objective measurements

In the first case, labels may not be consistent and contain noise.

Training set A training set is a set of training examples. The main idea is to pass the training set to a learning algorithm, that considers a certain loss function to finally produce a predictor.

$$\textit{training set} \longrightarrow \textit{learning algorithm} \longrightarrow \textit{predictor}$$

Test set A test set is a set of unseen examples, used to evaluate the predictor performance. Typically a whole dataset is splitted into the training and test sets.

$$\textit{dataset} \longrightarrow (\textit{training set}, \textit{test set})$$

Test error Let's now consider a predictor f learned by a certain learning algorithm A using a loss function l. We can compute the test error as follows

$$\textit{test set} = (x'_1, y'_1), \ldots, (x'_n, y'_n)$$

$$\textit{error} = \frac{1}{n} \sum_{t=1}^{n} l(y'_t, f(x'_t))$$

Test error is a proxy for the behavior of the predictor *in the field*.

Remark. *Our goal is to develop a theory to guide us into the design of learning algorithms that generates predictors with small test error with respect to some loss function.*
Having more data should be convenient to reduce the test error.

Training error Given a training set

$$S = (x_1, y_1), \ldots, (x_m, y_m)$$

The algorithm knows a loss function l, the training error can be computed as follows

$$\hat{l}_s(f) = \frac{1}{m} \sum_{t=1}^{m} l(y_t, f(x_t))$$

The idea is that the learning algorithm can infer the test error from the training error.

2 Empirical risk minimizer

The main idea of the Empirical risk minimizer is to find the best predictor given a training set and a loss function.

Formally, we define F as a set of predictors, and l a loss function.
The ERM works like this:

$$\textit{training set} \longrightarrow \textit{ERM} \longrightarrow \hat{f} \in \min_{f \in F} \hat{l}_S(f)$$

Things can go wrong, for instance if the minimum error

$$\min_{f \in F} \frac{1}{n} \sum_{t=1}^{n} l(y'_t, f(x'_t))$$

is large, we can't find a good predictor. The solution could be increasing the F domain, but that's not always the case.

Example Let's consider the example below, where we want to predict these labels.

$$X = \{x_1, \ldots, x_5\}, Y = \{-1, 1\}, l = \textit{zero one loss}$$

F contains all the binary classifiers like $f : \{x_1, \ldots, x_5\} \longrightarrow \{-1, 1\}$ so we have that $|F| = 2^5 = 32$.

Let's now consider 4 predictors for the 5 data points

	x_1	x_2	x_3	x_4	x_5
Labels	-1	1	1	1	1
f_1	-1	1	1	1	1
f_2	-1	1	1	-1	1
f_3	-1	1	1	1	-1
f_4	-1	1	1	-1	-1

If the training set is made of the first 3 data points, namely

$$S = \{(x_i, y_i)\}, i \in \{1, 2, 3\}$$

All the 4 predictors would have the same, null, training error. In such a case we can't decide what predictor to use, as training error do not give us any relevant information.

As a rule of thumb, given m the size of the training set S, if F is bound, the following should hold

$$m \geq \log_2 F$$

Underfitting and overfitting The first means having too few predictors to predict well, so the error on the test set is too large.
In the second case there are too many predictors, and training error do not gives us any information, as happened in the previous example.

Noisy labels In practice there is no $f^* : X \to Y$ such that $f^*(x) = y$, for all examples (x, y). This means y is noisy given x, for instance, the same data point may be observed with different labels.

This can happen in two scenarios:

1. when there are *humans in the loop*, as human decision can be ambiguous
2. lack of information, if we base our predictions on vectors and they do not contain sufficient information, a prediction can't be made precisely, as multiple labels could be linked to the same vector

3 NN classifiers

Let's consider the case where data is made of coordinates and labels are integers:

$$x = (x_1, x_2) \in \mathbb{R},\ S = (x_1, y_1), \ldots, (x_m, y_m),\ y_t \in \{-1, +1\}$$

A really simple approach to classification is to consider the *nearest neighbor rule*.

3.1 Nearest neighbor predictor

A new data point is classified looking at the class of the nearest training point. This approach induces a partition of the space in Voronoi cells. A distance function is needed, for instance the Euclidean distance.
If there's a tie we can specify a rule to break it.

The nearest neighbor rule generates a predictor with zero training error, on the other end, a predictor is made of the entire training set, as all the data points are needed to compute a prediction.

3.2 K-nearest neighbor

A generalization of the nearest neighbor rule can be using the majority of the labels of the closest K training points. K is odd to avoid ties.

Varying the K parameter induces different classification boundaries, for instance, if K equals to the number of training points, the prediction is always the major label. The number of *switches* in the classification decreases by increasing K.

Usually, with a small parameter the predictor is overfitting the data, while with a large parameter underfitting occurs.

Extensions K-NN can be extended to multi-class classification and regression, in the second we can compute the average of the closest points.

4 Tree predictors

K-nn works only for numerical attributes, we want now to generalize to heterogeneous data. The main idea is to divide the points with orthogonal lines in the space, for instance, people that are younger than 30 and smokes, etc.

4.1 Structure

A tree predictor is a ordered rooted tree where each node has many children. Each children represent a decision made considering the value of on a certain categorical parameter or a function over a numerical value.

Each node has at least two children or is a leaf, i.e. a node where a prediction is made.

Split function Each internal node is associated with a test, so, given an attribute i and a node u with k children, there is a function

$$f : X_i \to \{1, \dots, k\}$$

that partitions the values of the attribute among the children. For instance:

$$f(x_1) = \begin{cases} 1 \text{ if } x_1 = c \\ 2 \text{ if } x_1 = d \\ 3 \text{ otherwise} \end{cases}$$

Remark. *The function typically considers only one attribute to avoid comparing attributes of different domains.*

Leaves labels Leaves nodes are tagged by labels $y \in Y$, so when the visit reaches a leaf a prediction is made.

4.1.1 Prediction

We take a specific data point. This point is passed to the root, that will returns a specific child index, based on the split function evaluation performed on the data point.

This continue recursively, until a leaf node is reached, at this point a prediction in made, according to the label stored in the reached leaf node.

The prediction $h_T(x)$ is always a label stored in some leaf, in particular, the one reached by the path followed by the data point.

4.2 Training

Given a training set S, how can we build a tree classifier?

Let's simplify the task by considering binary classification, $Y = \{+1, -1\}$ and complete binary trees, so each node has either zero or two children.

The training set is of the form $S = (x_1, y1), \ldots, (x_m, y_m)$, where each x_i is made of multiple categorical and numerical attributes. The loss taken into consideration is the zero-one loss.

Single starting node We start as a single node, the root, and the label associated with it is the majority of labels in S. We made a constant classifier.

Splitting the node We now consider a certain split function f_i, that split the dataset into two leaves, l and l'. We obtain:

1. $S_l = \{(x_t, y_t), f_i(x_t)$ *routed to* $l\}$
2. $S_{l'} = \{(x_t, y_t), f_i(x_t)$ *routed to* $l'\}$

There's a need to decide which label to associate to the two leaves. We can pick the majority of labels contained in each one of them. this minimize the training error.

Training error Let T be a certain tree classifier, S_l is the training points routed to the leaf l, $N_l = |S_l|$, y_l the majority of labels in S_l.

$$S_l^+ = \{(x_t, y_t) \in S_l, y_t = +1\}, \quad S_l^- = \{(x_t, y_t) \in S_l, y_t = -1\}$$

$$y_l = \begin{cases} +1 \textit{ if } N_l^+ \geq N_l^- \\ -1 \textit{ otherwise} \end{cases}$$

Let's compute the error, where I is one if the condition is true

$$\begin{aligned}\hat{l}_j(h_t) &= \frac{1}{m}\sum_{t=1}^{m} I(h_t(x_t) \neq y_t) \\ &= \frac{1}{m}\min\{N_l^+, N_l^-\}\end{aligned}$$

If we take the sum of the positive and negative sets over all leaves, we obtain the cardinality of $|S|$

$$\sum_l (N_l^+ + N_l^-) = m = |S|$$

We can keep on going with the previous equation:

$$\begin{aligned}\hat{l}_j(h_t) &= \frac{1}{m}\min\{N_l^+, N_l^-\} \\ \hat{l}_j(h_t) &= \frac{1}{m}\min\left\{\frac{N_l^+}{N_l}, \frac{N_l^-}{N_l}\right\}N_l \\ \hat{l}_j(h_t) &= \Psi\left(\frac{N_l^+}{N_l}\right)N_l \quad \text{Where } \Psi(a) = \min\{a, 1-a\}\end{aligned}$$

Given a certain tree, we can consider the training error

$$\Psi\left(\frac{N_l^+}{N_l}\right)N_l = \Psi\left(\frac{N_{l'}^+}{N_{l'}}\frac{N_{l'}}{N_l} + \frac{N_{l''}^+}{N_{l''}}\frac{N_{l''}}{N_l}\right)N_l$$

If we plot Ψ we can see it is concave, so we can apply the *Jenses inequiality*:

$$\forall a, b \in [0,1], \forall \alpha \in [0,1], \Psi(\alpha a + (1-\alpha)b) \geq \alpha\Psi(a) + (1-\alpha)\Psi(b)$$

So we keep going with the previous equation

$$\begin{aligned}\Psi\left(\frac{N_{l'}^+}{N_{l'}}\frac{N_{l'}}{N_l} + \frac{N_{l''}^+}{N_{l''}}\frac{N_{l''}}{N_l}\right)N_l &\geq \left(\frac{N_{l'}}{N_l}\Psi\left(\frac{N_{l'}^+}{N_{l'}}\right) + \frac{N_{l''}}{N_l}\Psi\left(\frac{N_{l''}^+}{N_{l''}}\right)\right)N_l \\ &= N_{l'}\Psi\left(\frac{N_{l'}^+}{N_{l'}}\right) + N_{l''}\Psi\left(\frac{N_{l''}^+}{N_{l''}}\right)\end{aligned}$$

So the error before splitting is bigger or equal to the error after splitting, so by splitting recursively we obtain a reduction of the training error.

$$\Psi\left(\frac{N_l^+}{N_l}\right)N_l \geq \Psi\left(\frac{N_{l'}^+}{N_{l'}}\right)N_{l'} + \Psi\left(\frac{N_{l''}^+}{N_{l''}}\right)N_{l''}$$

This means replacing the leaf node S_l obtained by the previous split with two new leaves $S_{l'}$, $S_{l''}$. If $\frac{N_l^+}{N_l} \approx \frac{1}{2}$, then the split is good whenever $\frac{N_{l'}^+}{N_{l'}}$ and $\frac{N_{l''}^+}{N_{l''}}$ are close to either 0 or 1.

Stopping criterion If the tree keeps splitting, the training error will go to 0, as all leaves will be pure, i.e. all labels are uniform inside a leaf. This means overfitting.[1]

So the objective is to grow the tree minimizing the training error, by using the minimum number of nodes. Every time we split we would like to fix as many mistakes as possible. For the sake of computation efficiency, we perform greedy cuts.

Other error functions In practice we do not use $\Psi(a) = \min(a, 1-a)$, but some other ones:

- *Gini function* $\Psi_2(p) = 2p(1-p)$
- *Scaled entropy* $\Psi_3(p) = -\frac{p}{2}\log_2 p - \frac{1-p}{2}\log_2(1-p)$

Those are upper-bounds of the training error, this means considering splits that would be excluded by the original Ψ function. This surrogate functions are better because they can move things around and create possible new splits, they help to not get stuck in local minima where growing two nodes seems to not help the situation, thus they help grow the tree more efficiently.

[1] Its the same principle of small K in K-NN