Statistical methods for machine learning

Francesco Tomaselli

April 22, 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*:

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

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

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

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

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

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

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.

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

F contains all the binary classifiers like $f : \{x_1, \dots, x_5\} \longrightarrow \{-1, 1\}$ 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:

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

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:

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

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.

4.2 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.

4.3 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 $S = (x_1, y1), \dots, (x_m, y_m)$, 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. $S_l = \{(x_t, y_t), f_i(x_t) \textit{ routed to } l\}$
2. $S_{l'} = \{(x_t, y_t), f_i(x_t) \textit{ routed to } l'\}$

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:

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

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[1]

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

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.

[1] Here we apply the jensen inequality to the previous

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.[2]

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 (split) 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.

Usage Tree classifiers can also be extended to multi-class classification and regression. In the first case, the major class is chosen, while in the second we take the mean of the value.

5 Statistical learning

We talked so far bout data, where examples were obtained through *independent* draws from a *fixed* but unknown *probability distribution* on the data domain cartesian the label set $X \times Y$.

Fixed means that the probability distribution do not change with time, while independent means that every data point is independent from the other.
The first statement is probably not true, as the data distribution could change in time, for instance news or images at certain hours, time of year etc.
The second statement is probably not true either, as maybe some data points are related, let's consider news about a pandemic in time.

The idea is to find a trade-off between simplicity of assumptions and effectivity of algorithms.

Dataset and learning problem From now on every example will be of the form

$$(X, Y) \sim D \text{ } over \text{ } X \times Y$$

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

where the twos are random variables. A dataset, made of training and test set will be a collection of *random statistical samples*

$$D = (X_1, Y_1) \dots (X_m, Y_m)$$

A learning problem will be defined as a pair between the dataset and the loss function (D, l).

5.1 Bayes Optimal Predictor

Statistical risk Given (D, l) and a predictor $h : X \to Y$, the question is how good is h on (D, l)? We can look at *statistical risk*

$$l_D(h) = E[l(Y, h(X))]$$

The expectation is computed with respect to the random draw of (X, Y) from D. The goal is to find a predictor h such that $l_D(h)$ is small as possible.

Optimal Predictor We want to define the predictor with the smallest statistical risk given a problem (D, l), it is defined as

$$f^*(x) = \min_{\hat{y} \in Y} E[l(Y, \hat{y})|X = x]$$

this is the conditional expected value of $l(Y, \hat{y})$, given $X = x$. I have a label distribution given a value for X. The predictor $f^* : X \to Y$ is called the *Bayes Optimal Predictor*.

By definition the following holds

$$\forall h : X \to Y, E[l(y, f^*(x))|X = x] \leq E[l(y, h(x))|X = x]$$

furthermore, by averaging over X, the following holds NOTA 1

$$l_D(f^*) = E[l(y, f^*(x))] \leq E[l(y, h(y))] = l_D(h)$$

Can we compute f^*? No unless I know the distribution of Y given X.

Remark. *That could also mean having to learn the distribution given the dataset, but that requires a lot of data.*

Unless labels are uniquely determined by data points, $l_D(f^*) > 0$, this is also called the *Bayes risk*. The quantity is greater than zero in the case that two same data points have different labels.

Optimal predictor for square loss Let's consider the square loss in a regression task. The optimal predictor now becomes:

$$
\begin{aligned}
f^*(x) &= \min_{\hat{y}\in\mathbb{R}} E[(y-\hat{y})^2|X=x] \\
&= \min_{\hat{y}\in\mathbb{R}} E[(y^2+\hat{y}^2-2y\hat{y})|X=x] \\
&= \min_{\hat{y}\in\mathbb{R}} E[y^2|X=x]+\hat{y}^2-2\hat{y}E[y|X=x] && \text{Expected value linearity} \\
&= \min_{\hat{y}\in\mathbb{R}} \hat{y}^2-2\hat{y}E[y|X=x] && \text{Can ignore the first}
\end{aligned}
$$

We now compute the derivative to find the optimal predictor

$$
\begin{aligned}
F(\hat{y}) &= \hat{y}^2-2\hat{y}\square \\
\frac{\mathrm{d}F(\hat{y})}{\mathrm{d}\hat{y}} &= 2\hat{y}-2\square = 0 \\
\hat{y} &= \square
\end{aligned}
$$

This means that the optimal predictor can be defined as follows:

$$f^*(x) = E[Y|X=x]$$

If we substitute $\hat{y}$ with $f^*(x)$ in the previous formula, we find that

$$E[(y-f^*(x))^2|X=x] = E[(y-E[Y|x]|X=x)] = \mathit{Var}[Y|X=x]$$

This implies that

$$l_D(f^*) = E[\mathit{Var}[Y|X]] \neq \mathit{Var}[Y|X]$$

Remark. *The idea is to find the right error function that ultimately defines an f^* predictor.*

5.2 Estimating the statistical risk

Let's assume to have a predictor $h: X \to Y$, we want to compute the risk $l_D(h)$. The loss $l(y,\hat{y}) \in [0,1]$ is bounded in that interval.

D is unknown, is that were not the case, we could compute f^*. We used so far the test error to evaluate a predictor performances. So we have the test set and the error:

$$S' = (x'_1,y'_1),\dots,(x'_n,y'_n), \;\; \hat{l}_{S'}(h) = \frac{1}{n}\sum_{t=1}^{n} l(y'_t, h(x'_t))$$

The test set is made of statistical samples, while the second value is the sample mean over S'.

We know that $(X'_t, Y'_t) \sim D$ is a random pair drawn from a distribution. If we compute the expected value of the loss, we find the risk

$$t=1,\dots,n \;\; E[l(y'_t,h(x'_t))] = l_D(h)$$

By the *law of large numbers* we know that $\hat{l}_{S'}(h)$ converges to $l_D(h)$ in probability.

Chernoff-Huffding inequality Given the independent random variables

$$Z_t, P(Z_t \in [0,1]) = 1, 0 \le Z_t \le 1, E[Z_t] = \mu, t = 1, \dots, n$$

The following holds $\forall \epsilon > 0$

$$P\left(\frac{1}{n}\sum_t Z_t > \mu + \epsilon\right) \le e^{-2\epsilon^2 n}, P\left(\frac{1}{n}\sum_t Z_t < \mu - \epsilon\right) \le e^{-2\epsilon^2 n}$$

Note that

$$E\left[\frac{1}{n}\sum_{t=1}^{n} Z_t\right] = \mu$$

Continuing with the previous equation we can write

$$l(y'_t, h(x'_t) = Z_t \in [0,1], E[l(y'_t, h(x'_t)] = l_D(h) = \mu$$

Union bound Given the events A_i,

$$P(\cup_i A_i) \le \sum_i^n P(A_i)$$

Let's not continue with the previous equation, calculating the probability the test error diverges from the risk

$$\begin{aligned} P(|l_D(h) - \hat{l}_{S'}(h)| \ge \epsilon) &= P(l_D(h) - \hat{l}_{S'}(h) > \epsilon \cup \hat{l}_{S'}(h) - l_D(h) > \epsilon) \\ &\le P(l_D(h) - \hat{l}_{S'}(h) > \epsilon) + P(\hat{l}_{S'}(h) - l_D(h) > \epsilon) \\ &= P(\hat{l}_{S'}(h) < l_D(h) - \epsilon) + P(\hat{l}_{S'}(h) > l_D(h) + \epsilon) \end{aligned}$$

We can apply the Chernoff-Huffding inequality to find that

$$P(|l_D(h) - \hat{l}_{S'}(h)| \ge \epsilon) \le 2e^{-2\epsilon^2 n}$$

If we call that probability δ and solve for ϵ

$$\delta = 2e^{-2\epsilon^2 n}, \epsilon = \sqrt{\frac{1}{2n} \ln \frac{2}{\delta}}$$

This means that the loss function diverges from the risk less than ϵ with probability $1 - \delta$. This actually means that the test error is a good proxy for the statistical risk of any given predictor h. It is in fact a probabilistic upper bound.

5.3 Balancing underfitting and overfitting

Bias-variance decomposition Let's assume to have some learning algorithm A, where $A(S) = h_S$ is the predictor output by A on input S, where the latter is a training set.

A is such that $A(S) \in H \; \forall S$, i.e. A always picks a predictor from some class H, for instance A could be the *ERM* for H.

Let h^* be any predictor with minimum risk in H_A, so

$$l_D(h^*) \leq \min_{h \in H_A} l_D(h)$$

If we consider a generic predictor in the set H_A we split the risk in three parts, and this is called bias-variance decomposition:

$$\begin{aligned} l_D(h_S) = l_D(h_S) - l_D(H^*) \quad & \text{estimation or variance error, overfitting} \\ + l_D(h^*) - l_D(f^*) \quad & \text{approximation or bias error, underfitting} \\ + l_D(f^*) \quad & \text{Bayes error} \end{aligned}$$

The estimation error is due to the fact that generally h_S is different from h^*, the approximation error arises because the set H_A might not contain the Bayes optimal predictor, and the Bayes error in unavoidable.

ERM risk Given the bias-variance decomposition we want to study the risk of an ERM. As seen in section 2, the ERM minimizes the training error $\hat{l}$

$$h_s = \min_{h \in H} \hat{l}(h)$$

while the best predictor in the class H is h^* such that

$$l_D(h^*) = \min_{h \in H} l_D(h)$$

Thanks to the law of large numbers, we know that the training error of h^* is close to its risk with high probability, as the training set is obtained by a random draw of the dataset, so it means that the result of section 5.2 on page 13 holds.

However, we can't apply the Chernoff-Hoeffding inequality to show that the training error of h_S is close to its risk, as it is function of the training set, thus a random variable.

The goal here is to rewrite the difference of risk and training error of h_S, so we are able to apply the inequality.

$$\begin{aligned} l_D(h_S) - l_D(h^*) &= l_D(h_S) - \hat{l}(h_S) + \hat{l}(h_S) - l_D(h^*) \\ &\leq l_D(h_S) - \hat{l}(h_S) + \hat{l}(h^*) - l_D(h^*) \\ &\leq |l_D(h_S) - \hat{l}(h_S)| + |\hat{l}(h^*) - l_D(h^*)| \\ &\leq 2 \max_{h \in H} |\hat{l}(h) - l_D(h)| \end{aligned}$$

We used the fact that h_S minimizes the training error, and we provided an upper bound with the max at the end. Therefore, for all $\epsilon > 0$

$$l_D(h_S) - l_D(h^*) > \epsilon \implies \max_{h \in H} |\hat{l}(h) - l_D(h)| > \frac{\epsilon}{2} \implies \exists h \in H : |\hat{l}(h) - l_D(h)| > \frac{\epsilon}{2}$$

The above holds for any training set, so we can write

$$\mathbb{P}(l_D(h_S) - l_D(h^*) > \epsilon) \le \mathbb{P}\left(\exists h \in H : |\hat{l}(h) - l_D(h)| > \frac{\epsilon}{2}\right)$$

Considering $|H| < \infty$, we can rewrite the exists as the union, and then apply the union bound and Cheronff-Hoeffding inequality:

$$\begin{aligned}
\mathbb{P}\left(\exists h \in H : |\hat{l}(h) - l_D(h)| > \frac{\epsilon}{2}\right) &= \mathbb{P}\left(\bigcup_{h\in H}\left(|\hat{l}(h) - l_D(h)| > \frac{\epsilon}{2}\right)\right)\\
&\le \sum_{h\in H}\mathbb{P}\left(|\hat{l}(h) - l_D(h)| > \frac{\epsilon}{2}\right)\\
&\le |H|\max_{h\in H}\mathbb{P}\left(|\hat{l}(h) - l_D(h)| > \frac{\epsilon}{2}\right)\\
&\le |H| 2e^{-m\epsilon^2/2}
\end{aligned}$$

In conclusion, we have that

$$\mathbb{P}(l_D(h_S) - l_D(h^*) > \epsilon) \le |H| 2e^{-m\epsilon^2/2}$$

Setting the left equal to δ and solving for ϵ we find that

$$l_D(h_S) \le l_D(h^*) + \sqrt{\frac{2}{m}\ln\frac{2|H|}{\delta}}$$

holds with probability $1 - \delta$.

This means that to reduce the estimation error of the ERM we must decrease $|H|$, but this might increase $l_D(h^*)$, we conclude by saying that the ERM generates predictors with high risk.

In the proof above we also shown that with probability $1 - \delta$

$$\forall h \in H |\hat{l}(h) - l_D(h)| \le \sqrt{\frac{1}{2m}\ln\frac{2|H|}{\delta}}$$

This implies that, when the cardinality of the training set is sufficiently large with respect to $\ln|H|$, then the training error becomes a good estimation of the risk. This tells us that ranking predictors in the training set by their training error, corresponds to ranking them according to their risk.

6 Consistency

Consistent algorithm A learning algorithm is said to be consistent with respect to some loss l, if, for all data distributions D the following holds

$$\lim_{m\to\infty} E[l_d(A(S_m))] = l_d(f^*)$$

Clearly, if A is consistent, then the class of predictors that A can output must be very large. Also, A must be *nonparametric*.

Nonparametric algorithms A learning algorithm A is nonparametric if the structure and consequently the memory footprint of the predictors output by A depends on the training data.

An example of a nonparametric algorithm is *k-NN*, as the Voronoi's cells generated by the algorithm depends only on training data. Another example is found in decision trees, as the perpendicular boundaries learned strongly depends on training data.

Remark. *Without a nonparametric algorithm it's impossible to output arbitrary complex predictors, basically it's impossible to meet the consistency requirement seen in the previous paragraph.*

K-NN consistency K-NN is not consistent fixed a k, but, if we assume the parameter to be function of the training set size m, we meet the consistency criterion.

$$k = K(m), \lim_{m\to\infty} K(m) = \infty$$

No free lunch theorem Let $a_1, a_2, \ldots$ a sequence of positive numbers converging to zero and such that $\frac{1}{16} \geq a_1 \geq a_2 \geq \ldots,$

$$\forall A \ \ \exists D \ s.t. \ l_D(f^*) = 0 \ and \ E[l_D(A(S_m))] \geq a_m \forall m$$

where A is a binary classifier with zero-one loss.

This means it's impossible to guarantee speed of convergence to Bayes risk for any data distribution. There could be some distribution where the algorithm converges fast, but there exists others where it's really slow.

Remark. *We can beat the theorem by making assumptions on the data distribution.*

Convergence recap We can summarize the situation as follows:

1. Under no conditions on D, there is no guaranteed convergence rate to Bayes risk
2. Under Lipschitz assumptions on η, the typical nonparametric convergence rate to Bayes risk is $m^{-1/(d+1)}$, so exponentially slow in d, this is the so called *curse of dimensionality*
3. Giving up consistency, and converging to $l_D(h^*)$ where h^* is the best of your class, e.g. ERM run on finite class H where H is simple, then convergence rate is $\frac{1}{\sqrt{m}}$, exponentially better than the nonparametric case

Remark. *Should I use K-NN for a dataset with* 10^3 *dimensions? Probably not, for the curse of dimensionality.*

7 Linear predictors

This section is about linear predictors, the idea is to find an hyperplane to classify data.

7.1 Structure

Linear predictors are the simplest case of parametric algorithms. We start with $X = \mathbb{R}^d$, so we handle only numerical data. The predictors are defined as follows

$$h : \mathbb{R}^d \to Y, \;\; h(x) = f(w^T x), \;\; w \in \mathbb{R}^d$$

where w is the description of the predictor, so it's what we are learning, and $f : \mathbb{R} \to \mathbb{R}$ is the activation function, and $w^T x = \sum_{i=0}^{d} w_i x_i = ||w||||x|| \cos \theta$ where θ is the angle between the two vectors.

Binary classification In a binary classification task, $f = sgn$

$$sgn(z) = \begin{cases} +1 \; if \; x \in H^+ \\ -1 \; if \; x \in H^- \end{cases}$$

The halfspaces H^+ and H^- are defined like this

$$H^+ = \{x : w^T x > c\} \;\; and \;\; H^- = \{x : w^T x \leq c\}$$

The predictor $h : \mathbb{R}^d \to \{-1, +1\}$ is defined as follows

$$h(x) = sgn(w^T x - c)$$

The hyperplane w has d dimensions, and it is called homogeneous if it crosses the origin. Basically if we learn $(w, c), c \neq 0$ we learned a non-homogeneous hyperplane, it holds that $v(w, -c), v \in \mathbb{R}^{d+1}$ is homogeneous.

Thus we can encapsulate the c parameter into the hyperplane by adding dimension and an extra feature to the data points with value 1.

7.2 Training

Note that $||w||$ is irrelevant to determine $sgn(w^T x) = sgn(||w||||x|| \cos \theta)$, thus we can assume $||w|| = 1$.

Let $H_d = \{w \in \mathbb{R}^d : ||w|| = 1\}$ we can rewrite the predictor output as follows

$$h(x) = I\{yw^T x \leq 0\}$$

and we can apply the ERM over H_D, whose output is

$$h_s = \min_{h \in H_d} \frac{1}{m} \sum_{t=1}^{m} I\{y_t w^T x_t \leq 0\}$$

this is nasty to minimize, as it is the sum of non-continuous functions, in fact the problem is in NP. Thus we can not say if a there exist an hyperplane with a bounded error.

Linearly separable data A training set is said to be Linearly separable it there exist a linear classifier with zero training error.

In other terms, we can define a margin function $\gamma(u)$, $u \in \mathbb{R}^d$. Given a separating hyperplane, this quantity is greater than zero:

$$\gamma(u) = \min_{t=1,\ldots,m} y_t u^T x_t > 0$$

If we scale the margin like this $\gamma(u)/||u||$, we obtain the closest point to the separating hyperplane u.

In this case, we can efficiently find a separating hyperplane by using linear solvers. The idea is to express each correct classification the hyperplane makes as a system of linear equations

$$y_t w^T x_t > 0 \quad t = 1, \ldots, m$$

Perceptron algorithm As typically linear solvers are quite complicated we use another way to find linear predictors.

The main idea is to start with an hyperplane and test the prediction on the data points, if an error is made, the parameters of the hyperplane are adjusted to make up for the error. Basically, if $y_t w^T x_t \leq 0$ then $w \leftarrow w + y_t x_t$.

If $y_t = 1$, the algorithm moves w towards x_t, in the opposite case it moves the plane away from the datapoint.

One might think that consequent updates may cancel off the progress that has been made so far but this is not the case, in fact a theorem tells that perceptron finds a solution if the dataset is linearly separable. The algorithm terminates after a number of updates not bigger than

$$\left(\min_{u:\gamma(u)\geq 1} ||u||^2\right)\left(\min_{t=1,\ldots,m} ||x_t||^2\right)$$

Note that although the algorithm terminates, there is no guarantee on the quality of the hyperplane found, in fact the returned hyperplane strongly depends on the order of the data points passed to the perceptron, the goal would be to maximize the margin.

Also, the convergence may not be reached in polynomial time.

7.3 Regression

Linear regression In linear regression the predictors are linear functions

$$h : \mathbb{R}^d \to \mathbb{R} \quad h(x) = w^T x$$

Given a dataset the linear regression predictor is the ERM with respect to the square loss.

$$w_S = \min_{w \in \mathbb{R}^d} \sum_{t=1}^{m} (w^T x_t - y_t)^2$$

The sum of square errors is a convex function. We can define $v = (w^T x_1, \dots, w^T x_m)$ and $y = (y_1, \dots, y_m)$ adn rewrite the previous equation as

$$\sum_{t=1}^{m} (w^T x_t - y_t)^2 = ||v - y||^2$$

If we consider all vectors as column vectors, $v = Sw$ where S is a $m \times d$ matrix containing $x_1^T, \dots, x_m^T$, thus the previous equation becomes

$$w_S = \min_{w \in \mathbb{R}^d} ||Sw - y||^2$$

As the function to minimize is a convex function we must find a zero gradient to minimize. USing matrix calculus, assuming $S^T S$ is non singular, i.e. invertible, we find that the ERM for this case is

$$w_s = (S^T S)^{-1} S^T y$$

Ridge regression When $S^T S$ is nearly singular we need a regularizer term to make the predictor less sensitive to training set perturbations. Redefining the function to minimize we gets

$$w_\alpha = \min_{w \in \mathbb{R}^d} ||Sw - y||^2 + \alpha ||w||^2$$

When α tends to zero we have linear regression, and when is goes to infinity we get the zero vector as solution. It can control the bias of the algorithm.

We have a close form solution by zeroing the gradient of the function, and the optimal one is:

$$w_\alpha = (\alpha I + S^T S)^{-1} S^T y$$

We don't need to worry about the $S^T S$ singularity anymore, in fact $\alpha I + S^T S$ is invertible, and the eigenvalues of the sum are simply $\alpha + \lambda_i$ where $\lambda_{1,\dots,d}$ are the eigenvalues of $S^T S$.

Remark. *In all the cases where ERM is unstable, we might benefit by adding a regularizer term in the objective function. Usually the risk is lower.*

8 Learning paradigms

Online learning Recalling how a perceptron works and focusing on training complexity, we have that an epoch in that learning algorithm is linear in the number of data points. This is a good result and it makes the perceptron really competitive on big datasets.

Also, perceptrons are good on data streams, as they can simply continue the training phase without recomputing everything.

This is called online learning, so we analyze a single data point and adapt the model, this means performing a local minimization.

Batch learning The opposite of online learning is batch learning, where we train a model on an entire dataset rather than on a single point. This means finding a global optimization, such as in ERM.

8.1 Sequential risk

Online learning protocol We start with parameters, that is a class H of predictors and a loss function l. The process is simple, a default predictor $h_1 \in H$ is given as output, then, for $t = 1, 2, \dots$:

1. The next example is observed (x_t, y_t)
2. The loss $l(h_t(x_t), y_t)$ is computed
3. The predictor is updated $h_t \leftarrow h_{t+1} \in H$

In general the algorithm does not store past examples.

Sequential risk The evaluation of the algorithm performances is made via the sequential risk, it somehow counts the evolution of the loss.

$$\frac{1}{T}\sum_{t=1}^{T} l(h_t(x_t), y_t)$$

this decreases overtime as the predictor improves.

Regret To have an analogy to the variance error, or the approximation error, we introduce the regret measure

$$\frac{1}{T}\sum_{t=1}^{T} l(h_t(x_t), y_t) - \min_{h\in H}\frac{1}{T}\sum_{t=1}^{T} l(h(x_t), y_t)$$

8.2 Gradient descent

Let's fix a stream of data, we introduce $l_t(h) = l(h_t(x_t), y_t), h \in H$.
The question now is, are there any other learning algorithms besides perceptron?

In convex optimization the goal is to find a minimum in a convex function $f : \mathbb{R}^d \to \mathbb{R}$ One easy way to minimize such a function is to use gradient descent.

Basically we compute the gradient at a point and move in the opposite direction, so towards the minimum gradient.

Projected OGD We now focus on using gradient descent in an online fashion, with the square loss for a linear predictor. The main idea is to compute the gradient of the loss for each data point and update the model accordingly.

For $t = 1, 2, \ldots, \eta > 0, U > 0$:

1. $w'_{t+1} = w_t - \frac{\eta}{\sqrt{t}} \nabla l_t(w_t)$
2. $w_{t+1} = \min_{w:||w|| \leq U} ||w - w'_{t+1}||$

Our goal is to bound the regret, so

$$\frac{1}{T} \sum_{t=1}^{T} l_t(w_t) - \min_{h \in H} \frac{1}{T} \sum_{t=1}^{T} l_t(u_T^*) \quad \text{where} \quad u_T^* = \min_{u:||u|| \leq U} \frac{1}{T} \sum_{t=1}^{T} l_t(u)$$

. . .