Machine Learning

`or: How I Learned to Stop Worrying and Love Restelli`

A series of extremely verbose and surely overkill notes on the "Machine Learning" course as taught by Marcello Restelli and Francesco Trovò during the second semester of the academic year 2018-2019 at Politecnico di Milano.

`Feel free to modify the document to correct any mistakes / add additional material in order to favour a better understanding of the concepts`

Theory Questions

Here are listed all the theory questions since 06/07/2017, divided in three chapters: Supervised Learning, Reinforcement Learning, and Exercises

Supervised Learning

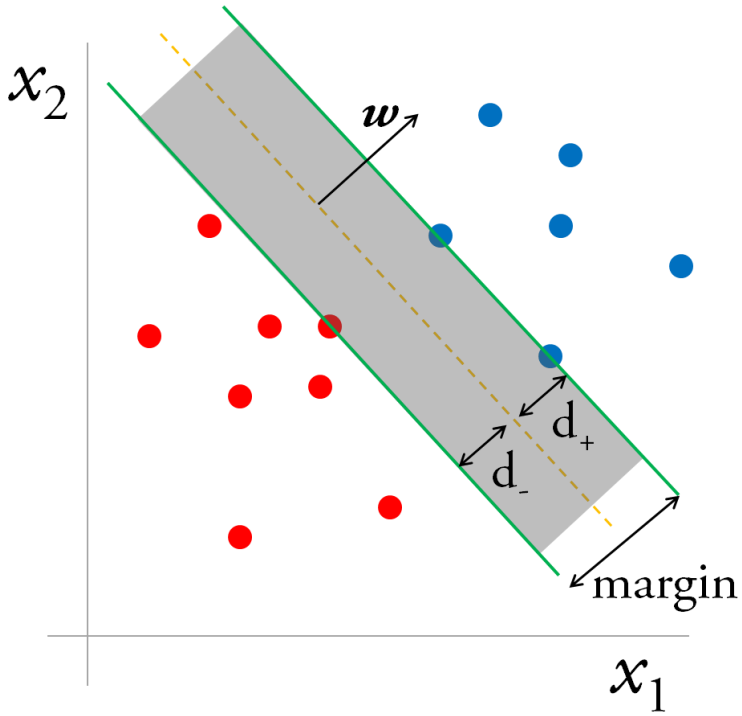
Support Vector Machines

Describe the supervised learning technique denominated Support Vector Machines for classification problems. Which algorithm can we use to train an SVM? Provide an upper bound to the generalization error of an SVM.

(*Andrea Bonvini*)

`TO DO : add upper bound stuff`

Our goal is to build a binary classifier by finding an hyperplane which is able to separate the data with the biggest *margin* possible.



With SVMs we force our *margin* to be at least *something* in order to accept it, by doing that we restrict the number of possible dichotomies, and therefore if we're able to separate the points with a fat dichotomy (*margin*) then that fat dichotomy will have a smaller *VC* dimension then we'd have without any restriction. Let's do that.

Let be $\mathbf{x}_n$ the nearest data point to the *hyperplane* $\mathbf{w}^T\mathbf{x} = 0$ (just image a *line* in a 2-D space for simplicity), before finding the distance we just have to state two observations:

- There's a minor technicality about the *hyperplane* $\mathbf{w}^T\mathbf{x} = 0$ which is annoying , let's say I multiply the vector $\mathbf{w}$ by 1000000 , I get the *same* hyperplane! So any formula that takes $\mathbf{w}$ and produces the margin will have to have built-in *scale-invariance*, we do that by normalizing $\mathbf{w}$, requiring that for the nearest data point $\mathbf{x}_n$:

  $$|\mathbf{w}^T\mathbf{x}_n| = 1$$

  (So I just scale $\mathbf{w}$ up and down in order to fulfill the condition stated above, we just do it because it's *mathematically convenient*! By the way remember that 1 does *not* represent the Euclidean distance)

- When you solve for the margin, the w_1 to w_d will play a completely different role from the role of w_0 , so it is no longer convenient to have them on the same vector. We pull out w_0 from $\mathbf{w}$ and rename w_0 with b (for *bias*).

$$\mathbf{w} = (w_1, \ldots, w_d)$$
$$w_0 = b$$

So now our notation is changed:

The *hyperplane* is represented by

$$\mathbf{w}^T \mathbf{x} + b = 0$$

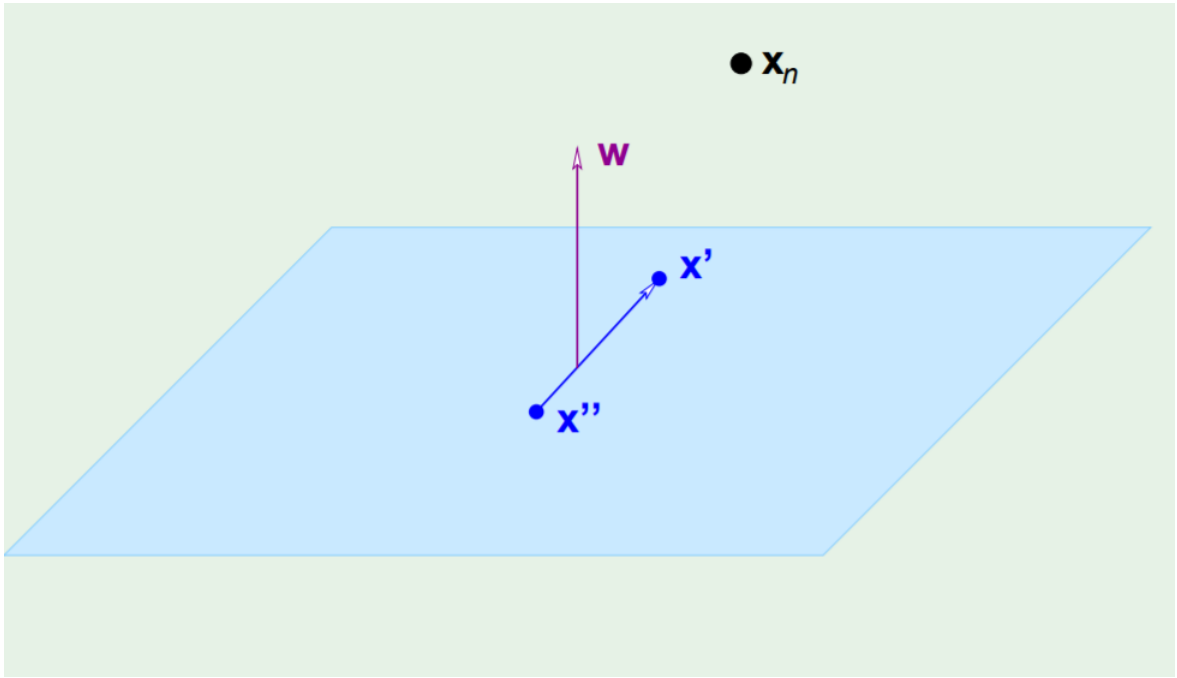
and our constraint becomes

$$|\mathbf{w}^T \mathbf{x}_n + b| = 1$$

It's trivial to demonstrate that the vector $\mathbf{w}$ is orthogonal to the *hyperplane*, just suppose to have two point $\mathbf{x}'$ and $\mathbf{x}''$ belonging to the *hyperplane* , then $\mathbf{w}^T \mathbf{x}' + b = 0$ and $\mathbf{w}^T \mathbf{x}'' + b = 0$.

And of course $\mathbf{w}^T \mathbf{x}'' + b - (\mathbf{w}^T \mathbf{x}' + b) = \mathbf{w}^T (\mathbf{x}'' - \mathbf{x}') = 0$

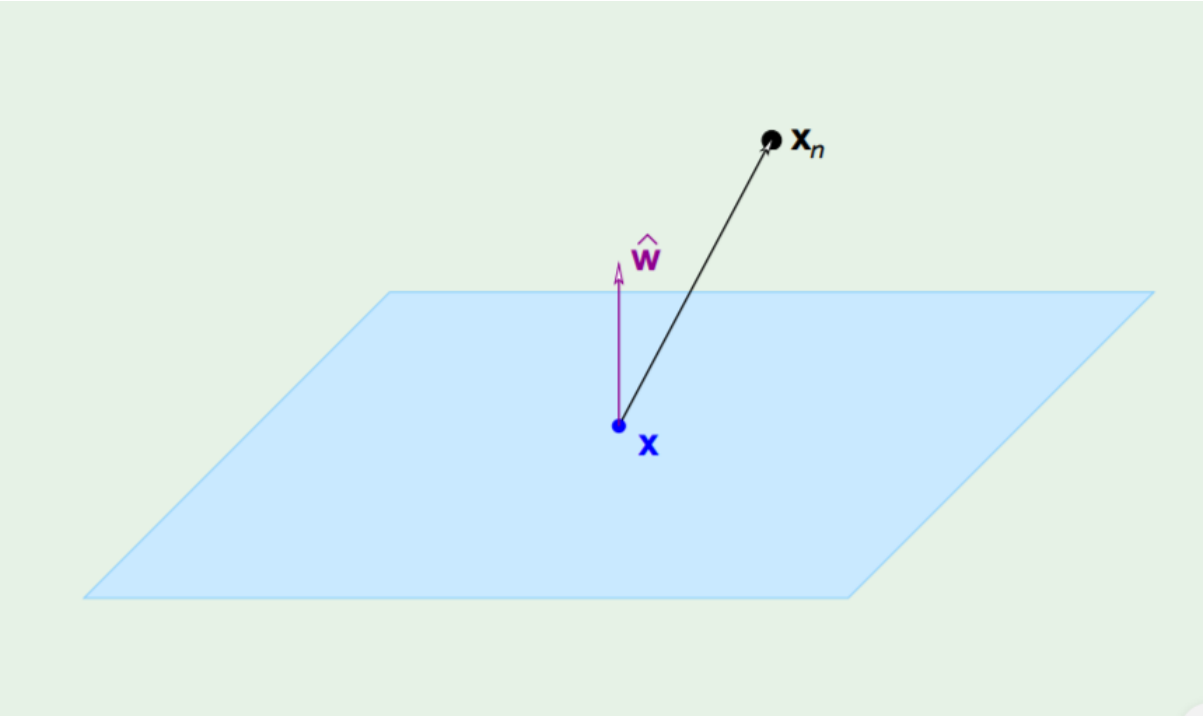
Since $\mathbf{x}'' - \mathbf{x}'$ is a vector which lays on the *hyperplane* , we deduce that $\mathbf{w}$ is orthogonal to the *hyperplane*.



Then the distance from $\mathbf{x}_n$ to the *hyperplane* can be expressed as a dot product between $\mathbf{x}_n - \mathbf{x}$ (where $\mathbf{x}$ is any point belonging to the plane) and the unit vector $\hat{\mathbf{w}}$, where $\hat{\mathbf{w}} = \frac{\mathbf{w}}{||\mathbf{w}||}$ (the distance is just the projection of $\mathbf{x}_n - \mathbf{x}$ in the direction of $\hat{\mathbf{w}}$!)

$$distance = | \hat{\mathbf{w}}^T (\mathbf{x}_n - \mathbf{x}) |$$

(We take the absolute value since we don't know if $\mathbf{w}$ is facing $\mathbf{x}_n$ or is facing the other direction)



We'll now try to simplify our notion of *distance*.

$$distance = \mid \hat{\mathbf{w}}^T(\mathbf{x}_n - \mathbf{x}) \mid = \frac{1}{||\mathbf{w}||} \mid \mathbf{w}^T\mathbf{x}_n - \mathbf{w}^T\mathbf{x} \mid$$

This can be simplified if we add and subtract the missing term b.

$$distance = \frac{1}{||\mathbf{w}||} \mid \mathbf{w}^T\mathbf{x}_n + b - \mathbf{w}^T\mathbf{x} - b \mid = \frac{1}{||\mathbf{w}||} \mid \mathbf{w}^T\mathbf{x}_n + b - (\mathbf{w}^T\mathbf{x} + b) \mid$$

Well, $\mathbf{w}^T\mathbf{x} + b$ is just the value of the equation of the plane...for a point *on* the plane. So without any doubt $\mathbf{w}^T\mathbf{x} + b = 0$, our notion of *distance* becomes

$$distance = \frac{1}{||\mathbf{w}||} \mid \mathbf{w}^T\mathbf{x}_n + b \mid$$

But wait...what is $\mid \mathbf{w}^T\mathbf{x}_n + b \mid$? It is the constraint the we defined at the beginning of our derivation!

$$|\mathbf{w}^T\mathbf{x}_n + b| = 1$$

So we end up with the formula for the distance being just

$$distance = \frac{1}{||\mathbf{w}||}$$

Which is sick AF.

Let's now formulate the optimization problem:

$$\underset{w}{\text{argmax}} \frac{1}{||\mathbf{w}||}$$
$$\text{subject to} \min_{n=1,2,\ldots,N} |\mathbf{w}^T\mathbf{x}_n + b| = 1$$

Since this is not a *friendly* optimization problem (the constraint have a minimum and an absolute value in them, which are annoying) we are going to find an equivalent problem which is easier to solve. Our optimization problem can be rewritten as

$$\underset{w}{\operatorname{argmin}} \frac{1}{2}\mathbf{w}^T\mathbf{w}$$
$$y_n(\mathbf{w}^T\mathbf{x}_n + b) \geq 1 \quad \text{for } n = 1, 2, \ldots, N$$

where y_n is a variable that we introduce that will be equal to either $+1$ or -1 accordingly to the sign of our prediction $(\mathbf{w}^T\mathbf{x}_n + b)$. One could argue that the new constraint is actually different from the former one, since maybe the $\mathbf{w}$ that we'll find will allow the constraint to be *strictly* greater than 1 for every possible point in our dataset [$y_n(\mathbf{w}^T\mathbf{x}_n + b) > 1 \ \ \forall n$] while we'd like it to be *exactly* equal to 1 for *at least* one value of n. But that's actually not true! Since we're trying to minimize $\frac{1}{2}\mathbf{w}^T\mathbf{w}$ our algorithm will try to scale down the right hyperplane $\mathbf{w}^T\mathbf{x}_n + b$ (by "*scaling down*" I simply mean multiplying it by a constant factor e.g. $\gamma < 1$) until it touches 1 for some specific point n of the dataset.

So how can we solve this? This is a constraint optimization problem with inequality constraints, we have to derive the *Lagrangian* and apply the *KKT* (Karush–Kuhn–Tucker) conditions.

Objective Function:

We have to minimize

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{n=1}^{N}\alpha_n(y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1)$$

w.r.t. to $\mathbf{w}$ and b and maximize it *w.r.t.* the *Lagrange Multipliers* α_n

We can easily get the two conditions for the unconstrained part:

$$\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{w} - \sum_{n=1}^{N}\alpha_n y_n \mathbf{x}_n = 0 \qquad \mathbf{w} = \sum_{n=1}^{N}\alpha_n y_n \mathbf{x}_n$$
$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{n=1}^{N}\alpha_n y_n = 0 \qquad \sum_{n=1}^{N}\alpha_n y_n = 0$$

And list the other *KKT* conditions:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 \geq 0 \qquad \forall i$$
$$\alpha_i \geq 0 \qquad \forall i$$
$$\alpha_i(y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1) = 0 \qquad \forall i$$

Alert : the last condition is called the KKT *dual complementary condition* and will be key for showing that the SVM has only a small number of "support vectors", and will also give us our convergence test when we'll talk about the *SMO* algorithm.

Now we can reformulate the *Lagrangian* by applying some substitutions

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{n=1}^{N}\alpha_n(y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1)$$

$$\mathcal{L}(\alpha) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{M} y_n y_m \alpha_n \alpha_m \mathbf{x}_n^T \mathbf{x}_m$$

(if you have doubts just go to minute 36.50 of this lecture by professor Yaser Abu-Mostafa at *Caltech*)

We end up with the *dual* formulation of the problem

$$\underset{\alpha}{\operatorname{argmax}} \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{M} y_n y_m \alpha_n \alpha_m \mathbf{x}_n^T \mathbf{x}_m$$

$$s.t. \qquad \alpha_n \geq 0 \quad \forall n$$

$$\sum_{n=1}^{N}\alpha_n y_n = 0$$

We can notice that the old constraint $\mathbf{w} = \sum_{n=1}^{N}\alpha_n y_n \mathbf{x}_n$ doesn't appear in the new formulation since it is *not* a constraint on α , it was a constraint on $\mathbf{w}$ which is not part of our formulation anymore.

How do we find the solution? we throw this objective (which btw happens to be a *convex* function) to a *quadratic programming* package.

Once the *quadratic programming* package gives us back the solution we find out that a whole bunch of α are just 0 ! All the α which are not 0 are the *support vectors* ! (i.e. the vectors that determines the width of the *margin*) , this can be noted by observing the last *KKT* condition, in fact either a constraint is active , and hence the point is a support vector, or its multiplier is zero.

Now that we solved the problem we can get both $\mathbf{w}$ and b.

$$\mathbf{w} = \sum_{\mathbf{x}_n \in \text{ SV}} \alpha_n y_n \mathbf{x}_n$$

$$y_n(\mathbf{w}^T\mathbf{x}_{n\in\text{SV}} + b) = 1$$

where $\mathbf{x}_{n\in\text{SV}}$ is any *support vector*. (you'd find the *same* b for every support vector)

But the coolest thing about *SVMs* is that we can rewrite our *objective functions* as follows:

$$\mathcal{L}(\alpha) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{M} y_n y_m \alpha_n \alpha_m k(\mathbf{x}_n \mathbf{x}_m)$$

We can use *kernels* !! (if you don't know what I'm talking about read the *kernel* related question present somewhere in this document)

Finally we end up with the following equation for classifying *new points*:

$$y(\mathbf{x}) = sign\left(\sum_{n=1}^{N}\alpha_n t_n k(\mathbf{x}, \mathbf{x}_n) + b\right)$$

The method described until here is called *hard-margin SVM* since the margin has to be satisfied strictly, it can happens that the points are not *linearly separable* in *any* way, or we just want to handle *noisy data* to avoid overfitting, so now we're going to briefly define another version of it, which is called *soft-margin SVM* that allows for few errors and penalizes for them.

We introduce *slack variables* ξ_i , in this way we allow to *violate* the margin constraint but we add a *penalty*.

We now have to

$$\begin{gathered}\text{Minimize } ||\mathbf{w}||_2^2 + C\sum_i \xi_i \\ \text{s.t.} \\ t_i(\mathbf{w}^T x_i + b) \geq 1 - \xi_i \ , \quad \forall i \\ \xi_i \geq 0 \ , \quad \forall i\end{gathered}$$

C is a coefficient that allows to treadeoff bias-variance and is chosen by *cross-validation*.

And obtain the *Dual Representation*

$$\begin{gathered}\text{Maximize } \quad \mathcal{L}(\alpha) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{M} y_n y_m \alpha_n \alpha_m k(\mathbf{x}_n \mathbf{x}_m) \\ \text{s.t.} \\ 0 \leq \alpha_n \leq C \quad \forall i \\ \sum_{n=1}^{N} \alpha_n t_n = 0\end{gathered}$$

Support vectors are points associated with $\alpha_n > 0$

if $\alpha_n < C$ the points lies *on the margin*

if $\alpha_n = C$ the point lies *inside the margin*, and it can be either *correctly classified* ($\xi_i \leq 1$) or *misclassified* ($\xi_i > 1$)

Fun fact: When C is large, larger slacks penalize the objective function of SVM's more than when C is small. As C approaches infinity, this means that having any slack variable set to non-zero would have infinite penalty. Consequently, as C approaches infinity, all slack variables are set to 0 and we end up with a hard-margin SVM classifier.

And what about generalization? Can we compute an *Error* bound in order to see if our model is overfitting? Kinda.

As *Vapnik* said: "In the support-vectors learning algorithm the complexity of the construction does not depend on the dimensionality of the feature space, but on the number of support vectors." So it's reasonable to define an upper bound of the error as:

$$L_h \leq \frac{\mathbb{E}[\text{number of support vectors}]}{N}$$

This is called *Leave-One-Out Bound* because _____ (https://ocw.mit.edu/courses/mathematics/18-465-topics-in-statistics-statistical-learning-theory-spring-2007/lecture-notes/l4.pdf) check here. The good thing is that it can be easily computed and we don't need to run SVM multiple times.

The other error bound is blabla

Sometimes for computational reasons, when we solve a problem characterized by a huge dataset, it is not possible to compute *all* the support vectors with generic quadratic programming solvers (the number of constraints depends on the number of samples), hence,specialized optimization algorithms are often used. One example is *Sequential Minimal Optimization (SMO)*:

Remember our formulation for the *soft-margin SVM*:

$$\mathcal{L}(\alpha) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{M} y_n y_m \alpha_n \alpha_m k(\mathbf{x}_n \mathbf{x}_m)$$
$$s.t.$$
$$0 \leq \alpha_i \leq C \quad \text{for } i = 1, 2, \ldots, n$$
$$\sum_{i=1}^{n} y_i \alpha_i = 0$$

SMO breaks this problem into a series of smallest possible sub-problems, which are then solved analytically. Because of the linear equality constraint involving the Lagrange multipliers α_i the smallest possible problem involves two such multipliers. Then, for any two multipliers α_1 and α_2 the constraints are reduced to:

$$0 \leq \alpha_1, \alpha_2 \leq C$$
$$y_i \alpha_1 + y_2 \alpha_2 = k$$

and this reduced problem can be solved analytically: one needs to find a minimum of a one-dimensional quadratic function. k is the negative of the sum over the rest of terms in the equality constraint, which is fixed in each iteration.

The algorithm proceeds as follows:

- Find a Lagrange multiplier α_1 that violates the *KKT* conditions for the optimization problem.
- Pick a second multiplier α_2 and optimize the pair (α_1,α_2).
- Repeat steps 1 and 2 until convergence.

When all the Lagrange multipliers satisfy the KKT conditions (within a user-defined tolerance), the problem has been solved. Although this algorithm is guaranteed to converge, heuristics are used to choose the pair of multipliers so as to accelerate the rate of convergence. This is critical for large data sets since there are $\frac{n(n-1)}{2}$ possible choices for α_i and α_j .

PAC & VC Dimension

Write just a very very little definition of PAC Learning, then define the VC dimension and describe the importance and usefulness of VC dimension in machine learning. Define the VC dimension of a hypothesis space. What is the VC dimension of linear classifiers?

(*Andrea Bonvini*)

`TODO: { add *Agnostic Learning* and finish *VC DIMENSION* }`

First, some concepts you need to know:

- We are talking about *Classification*.
- Overfitting happens:
  - Because with a large hypothesis space the training error is a bad estimate of the prediction error, hence we would like to infer something about the generalization error from the training samples.
  - When the learner doesn't have access to enough samples, hence we would like to estimate how many samples are enough.

  This cannot be performed by measuring the bias and the variance, but we can bound them.

  Given:

  - Set of instances $\mathcal{X}$
  - Set of hypotheses $\mathcal{H}$ (finite)
  - Set of possible target concepts C. Each concept c corresponds to a boolean function $c : \mathcal{X} \rightarrow \{0, 1\}$ which can be viewed as belonging to a certain class or not
  - Training instances generated by a fixed, unknown probability distribution P over X.

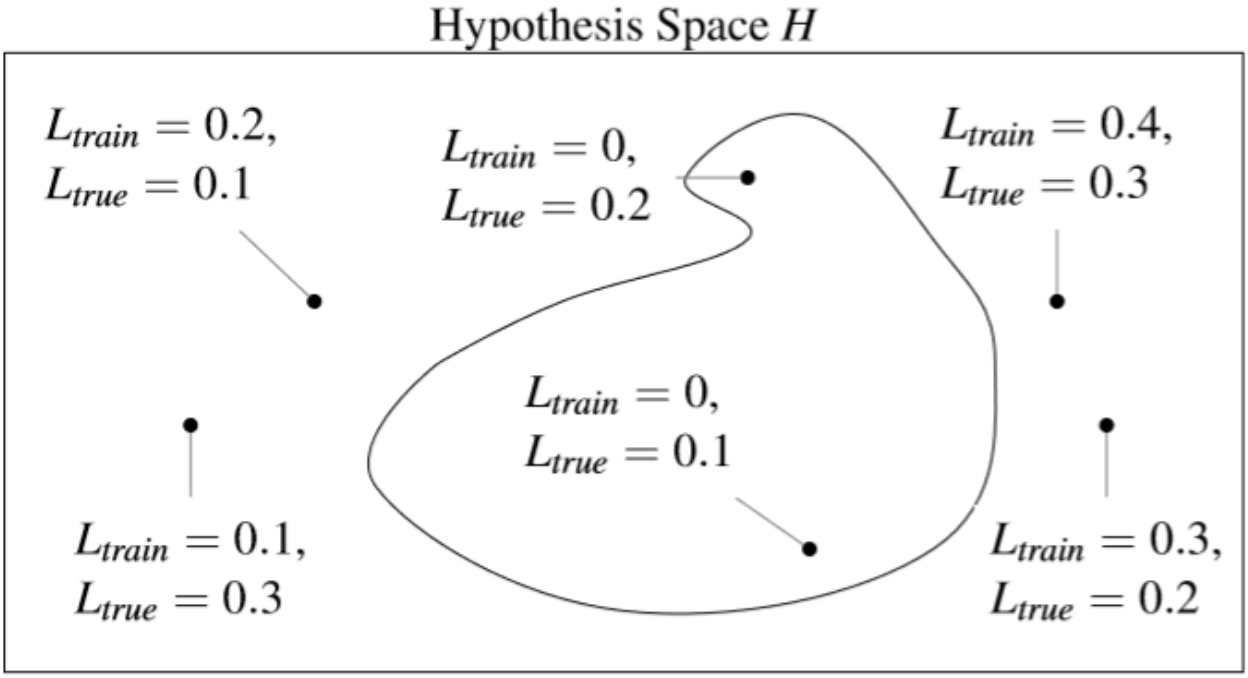
  The learner observes a sequence D of training examples $\langle x, c(x)\rangle$, for some target concept $c \in C$ and it must output a hypothesis h estimating c.

  h is evaluated by its performance on subsequent instances drawn according to P

  $$L_{true} = Pr_{x \in P}[c(x) \neq h(x)]$$

  We want to bound L_{true} given L_{train}, which is the percentage of misclassified training instances.

  Let's talk now about *Version Spaces* : The version space $VS_{\mathcal{H},\mathcal{D}}$ is the subset of hypothesis in H consistent with the training data D (in other words is the subset of H where $L_{train} = 0$).



How likely is the learner to pick a *bad hypothesis* ?

Theorem
If the hypothesis space H is finite and $\mathcal{D}$ is a sequence of $N \geq 1$ independent random examples of some target concept c, then for any $0 \leq \epsilon \leq 1$, the probability that $VS_{H,\mathcal{D}}$ contains a hypothesis error greater that ϵ is less than $|H|e^{-\epsilon N}$:

$$Pr(\exists h \in H : L_{train} = 0 \wedge L_{true} \geq \epsilon) \leq |H|e^{-\epsilon N}$$

If you're interested in the proof:

$$\begin{aligned}
&Pr((L_{train}(h_1) = 0 \wedge L_{true}(h_1) \geq \epsilon) \vee \cdots \vee (L_{train}(h_{|H|}) = 0 \wedge L_{true}(h_{|H|}) \geq \epsilon)) \\
&\quad \leq \sum_{h \in H} Pr(L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) && \text{(Union bound)} \\
&\quad \leq \sum_{h \in H} Pr(L_{train}(h) = 0 | L_{true}(h) \geq \epsilon) && \text{(Bound using Bayes' rule)} \\
&\quad \leq \sum_{h \in H} (1 - \epsilon)^N && \text{(Bound on individual } h_i s\text{)} \\
&\quad \leq |H|(1 - \epsilon)^N && (k \leq |H|) \\
&\quad \leq |H|e^{-\epsilon N} && (1 - \epsilon \leq e^{-\epsilon}, \text{ for } 0 \leq \epsilon \leq 1)
\end{aligned}$$

where k is probably the number of hypothesis $h \in VS_{\mathcal{H},\mathcal{D}}$

Now, we use a *Probably Approximately Correct (PAC) bound*:

If we want this probability to be at most δ we can write

$$|H|e^{-\epsilon N} \leq \delta$$

which means

$$N \geq \frac{1}{\epsilon}\left(\ln|H| + \ln\left(\frac{1}{\delta}\right)\right)$$

and

$$\epsilon \geq \frac{1}{N}\left(\ln|H| + \ln\left(\frac{1}{\delta}\right)\right)$$

Note that if, *for example*, we consider M boolean features, there are $|C| = 2^M$ distinct concepts and hence $|H| = 2^{2^M}$ (which is huuuge)

If you wonder why let's suppose we have just 2 boolean features (A and B) , then we have $|H| = 2^{2^2} = 16$ distinct boolean functions :

```
A   B|  F0  F1  F2  F3  F4  F5  F6  F7
0   0|  0   0   0   0   0   0   0   0
0   1|  0   0   0   0   1   1   1   1
1   0|  0   0   1   1   0   0   1   1
1   1|  0   1   0   1   0   1   0   1

A   B|  F8  F9  F10 F11 F12 F13 F14 F15
0   0|  1   1   1   1   1   1   1   1
0   1|  0   0   0   0   1   1   1   1
1   0|  0   0   1   1   0   0   1   1
1   1|  0   1   0   1   0   1   0   1

function            symbol          name
F0                  0               FALSE
F1                  A ^ B           AND
F2                  A ^ !B          A AND NOT B
F3                  A               A
F4                  !A ^ B          NOT A AND B
F5                  B               B
F6                  A xor B         XOR
F7                  A v B           OR
F8                  A nor B         NOR
F9                  A XNOR B        XNOR
F10                 !B              NOT B
F11                 A v !B          A OR NOT B
F12                 !A              NOT A
F13                 !A v B          NOT A OR B
F14                 A nand B        NAND
F15                 1               TRUE
```

and so the bounds have an *exponential* dependency on the number of features M !

$$N \geq \frac{1}{\epsilon}\left(\ln|H| + \ln\left(\frac{1}{\delta}\right)\right)$$
$$N \geq \frac{1}{\epsilon}\left(\ln 2^{2^M} + \ln\left(\frac{1}{\delta}\right)\right)$$
$$N \geq \frac{1}{\epsilon}\left(\underline{\underline{2^M}} \ln 2 + \ln\left(\frac{1}{\delta}\right)\right)$$
$$\epsilon \geq \frac{1}{N}\left(\ln|H| + \ln\left(\frac{1}{\delta}\right)\right)$$
$$\epsilon \geq \frac{1}{N}\left(\ln 2^{2^M} + \ln\left(\frac{1}{\delta}\right)\right)$$
$$\epsilon \geq \frac{1}{N}\left(\underline{\underline{2^M}} \ln 2 + \ln\left(\frac{1}{\delta}\right)\right)$$

which is bad news.

Instead of having an *exponential* dependency on M we'd like to have a, *guess what?* , *polynomial* dependency!

Consider a class C of possible target concepts defined over a set of instances X and a learner L using hypothesis space H.

Definition :

C is ***PAC-learnable*** it there exists an algorithm L such that for every $c \in C$, for any distribution P , for any ϵ such that $0 \leq \epsilon \leq \frac{1}{2}$ and δ such that $0 \leq \delta \leq 1$, with probability at least $1 - \delta$, outputs an hypothesis $h \in H$, such that $L_{true}(h) \leq \epsilon$, using a number of samples that is polynomial of $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$

C is ***efficiently PAC-learnable*** by a learner L using H if and only if every $c \in C$, for any distribution P , for any ϵ such that $0 \leq \epsilon \leq \frac{1}{2}$ and δ such that $0 \leq \delta \leq \frac{1}{2}$, with probability at least $1 - \delta$, outputs an hypothesis $h \in H$, such that $L_{true}(h) \leq \epsilon$, using a number of samples that is polynomial of $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$, M and $size(c)$.

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

- When counting the number of hypotheses, the entire input space is taken into consideration. In the case of a perceptron, each perceptron differs from another if they differ in at least one input point, and since the input is continuous, there are an infinite number of different perceptrons. (e.g. in a $2 - D$ space you can draw an infinite number of different lines)

  Instead of counting the number of hypotheses in the entire input space, we are going to restrict the count only to the sample: a *finite* set of input points. Then, simply count the number of the possible *dichotomies*. A dichotomy is like a mini-hypothesis, it's a *configuration of labels* on the sample's input points.

  A *hypothesis* is a function that maps an input from the entire *input space* to a result:

  $$h : \mathcal{X} \to \{-1, +1\}$$

  The number of hypotheses $|\mathcal{H}|$ can be infinite.

  A *dichotomy* is a hypothesis that maps from an input from the *sample size* to a result:

  $$h : \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\} \to \{-1, +1\}$$

  The number of *dichotomies* $|\mathcal{H}\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}|$ is at most 2^N, where N is the sample size.

e.g. for a sample size $N = 3$ we have at most 8 possible dichotomies:

```
    x1 x2 x3
1   -1 -1 -1
2   -1 -1 +1
3   -1 +1 -1
4   -1 +1 +1
5   +1 -1 -1
6   +1 -1 +1
7   +1 +1 -1
8   +1 +1 +1
```

- The *growth function* is a function that counts the *most* dichotomies on any N points.

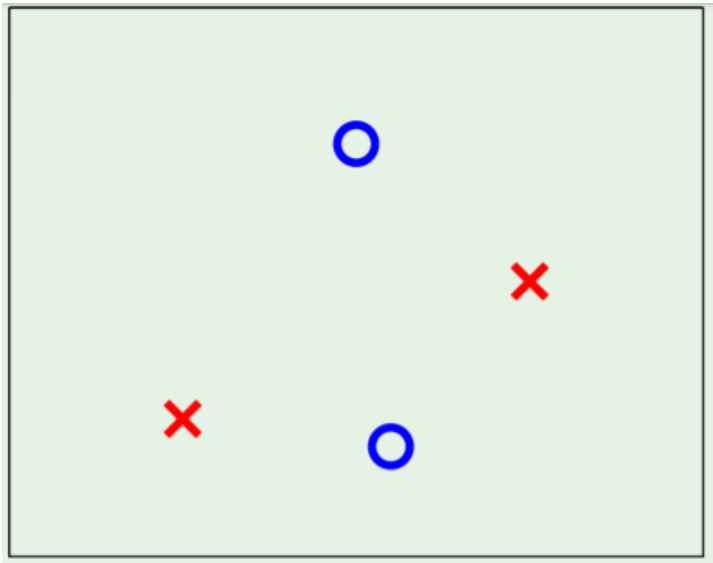
$$m_{\mathcal{H}}(N) = \max_{\mathbf{x}_1,\ldots,\mathbf{x}_N \in \mathcal{X}} |\mathcal{H}(\mathbf{x}_1, \ldots, \mathbf{x}_N)|$$

This translates to choosing any N points and laying them out in *any* fashion in the input space. Determining m is equivalent to looking for such a layout of the N points that yields the *most* dichotomies.

The growth function satisfies:

$$m_{\mathcal{H}}(N) \leq 2^N$$

This can be applied to the perceptron. For example, when $N = 4$, we can lay out the points so that they are easily separated. However, given a layout, we must then consider all possible configurations of labels on the points, one of which is the following:



This is where the perceptron breaks down because it *cannot* separate that configuration, and so $m_{\mathcal{H}}(4) = 14$ because two configurations—this one and the one in which the left/right points are blue and top/bottom are red—cannot be represented. For this reason, we have to expect that that for perceptrons, m can't be the maximum possible because it would imply that perceptrons are as strong as can possibly be.

The *VC (Vapnik-Chervonenkis) dimension* of a hypothesis set $\mathcal{H}$, denoted by $d_{VC}(\mathcal{H})$ is the largest value of N for which $m_{\mathcal{H}}(N) = 2^N$, in other words is "*the most points $\mathcal{H}$ can shatter* "

We can say that the *VC* dimension is one of many measures that characterize the expressive power, or capacity, of a hypothesis class.

You can think of the VC dimension as "how many points can this model class memorize?" (a ton? $\rightarrow$ BAD! not so many? $\rightarrow$ GOOD!)

With respect to learning, the effect of the VC dimension is that if the VC dimension is finite, then the hypothesis will generalize:

$$d_{vc}(\mathcal{H}) \implies g \in \mathcal{H} \text{ will generalize}$$

The key observation here is that this statement is independent of:

- The learning algorithm
- The input distribution
- The target function

The only things that factor into this are the training examples, the hypothesis set, and the final hypothesis.

Ridge Regression

Describe the supervised learning technique called ridge regression for regression problems.
(*William Bonvini*)
Ridge Regression is a regularization technique that aims to reduce model complexity and prevent over-fitting which may result from simple linear regression.
In ridge regression, the cost function is altered by adding a penalty equivalent to the square of the magnitude of the coefficients.

$$cost\ function = \sum_{i=1}^{M}(y_i - \sum_{j=0}^{p}(w_j \times x_{ij})\ )^2 + \lambda \sum_{j=0}^{p} w_j^2$$

where M is the number of samples and p is the number of features.

The penalty term λ regularizes the coefficients of the features such that if they take large values the optimization function is penalized.

When $\lambda \to 0$, the cost function becomes similar to the linear regression cost function. So lowering λ, the model will resemble the linear regression model.

It is always principled to standardize the features before applying the ridge regression algorithm. Why is this? The coefficients that are produced by the standard least squares method are scale equivariant, i.e. if we multiply each input by c then the corresponding coefficients are scaled by a factor of $\frac{1}{c}$. Therefore, regardless of how the predictor is scaled, the multiplication of the coefficient and the predictor $(w_j x_j)$ remains the same. However, this is not the case with ridge regression, and therefore, we need to standardize the predictors or bring the predictors to the same scale before performing ridge regression. the formula used to do this is given below.

$$\hat{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \bar{x}_j)^2}}$$

(Sources: tds - Ridge And Lasso Regression - tds - Regularization in ML)

Since λ is not defined a priori, we need a method to select a good value for it.
We use Cross-Validation for solving this problem: we choose a grid of λ values, and compute the cross-validation error rate for each value of λ.
We then select the value for λ for which the cross-validation error is the smallest.
Finally, the model is re-fit using all of the available observations and the selected value of λ.
Restelli offers the following cost function notation:

$L(w) = L_D(\mathbf{w}) + \lambda L_W(\mathbf{w})$

where $L_D(\mathbf{w})$ is the error on data terms (e.g. RSS) and $L_W(\mathbf{w})$ is the model complexity term.

By taking $L(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2}||\mathbf{w}||_2^2$

we obtain:

$$L(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(t_i - \mathbf{w}^T\phi(\mathbf{x}_i))^2 + \frac{\lambda}{2}||\mathbf{w}||_2^2$$

We observe that the loss function is still quadratic in **w**:

$$\hat{\mathbf{w}}_{ridge} = (\lambda \mathbf{I} + \mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{t}$$

(Source: Restelli's Slides)

Ridge Regression is, for example, used when the number of samples is relatively small wrt the number of features.
Ridge Regression can improve predictions made from new data (i.e. reducing variance) by making predictions less sensitive to the Training Data.

(Source: statquests explanation)

Ridge vs Lasso

Describe and compare the ridge regression and the LASSO algorithms.
(William Bonvini)

Before diving into the definitions, let's define what is Regularization: it's a technique which makes slight modifications to the learning algorithm such that the model avoids overfitting, so performing better on unseen data.

Ridge Regression is a Regularization Technique which consists in adding to the Linear Regression Loss Function a penalty term called L2 regularization element:

$L(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(t_i - \mathbf{w}^T\phi(\mathbf{x}_i))^2 + \frac{\lambda}{2}||\mathbf{w}||_2^2$

Lasso Regression is a Regularization Technique very similar to Ridge Regression, but instead of adding a L2 regularization element, it adds the so called L1 regularization element:

$L(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(t_i - \mathbf{w}^T\phi(\mathbf{x}_i))^2 + \frac{\lambda}{2}||\mathbf{w}||_1$

The main difference between Ridge and Lasso Regression is that Ridge Regression can only shrink to weights of the features close to 0 while Lasso Regression can shrink them all the way to 0.
This is due to the fact that Ridge Regression squares the features weights, while Lasso Regression considers the absolute value of such weights.

This means that Lasso Regression can exclude useless features from the lost function, so being better than Ridge Regression at reducing the variance in models that contain a lot of useless features. In contrast, Ridge Regression tends to do a little better when most features are useful.

You may ask yourself why Lasso is able to shrink some weights exactly to zero, while Ridge doesn't.
The following example may be explanatory:

Consider a model with only one feature x_1. This model learns the following output: $\hat{f}(x_1) = 4x_1$. Now let's add a new feature to the model: x_2. Suppose that such second feature does not tell anything new to the model, which means that it depends linearly from x_1. Actually, $x_2 = x_1$.
This means that any of the following weights will do the job:
$\hat{f}(x_1, x_2) = 4x_1$

$\hat{f}(x_1, x_2) = 2x_1 + 2x_2$

$\hat{f}(x_1, x_2) = x_1 + 3x_2$

$\hat{f}(x_1, x_2) = 4x_2$

We can generalize saying that $\hat{f}(x_1, x_2) = w_1x_1 + w_2x_2 \quad with \;\; w_1 + w_2 = 4$.

Now consider the l_1 and l_2 norms of various solutions, remembering that $l_1 = |w_1 + w_2|$ and that $l_2 = (w_1^2 + w_2^2)$.

w_1	w_2	l_1	l_2
4	0	4	16
2	2	4	8
1	3	4	10
-1	5	6	26

we can see that minimizing l_2 we obtain $w_1 = w_2 = 2$, which means that it, in this case, tends to spread equally the weights.
While l_1 can choose arbitrarily between the first three options, as long as the weights have the same sign it's ok.

Now suppose $x_2 = 2x_1$, which means that x_2 does not add new information to the model, but such features have different scale now. We can say that all functions with $w_1 + 2w_2 = k$ (in the example above $k = 4$) give the same predictions and have same empirical risk.

For clarity I will show you some of the possible values we can assign to the weights.

w_1	w_2	l_1	l_2
4	0	4	16
3	0.5	3.5	9.25
2	1	3	5
1	1.5	2.5	3.25
0	2	2	4

l_1 (which translates into Lasso Regression) chooses $w_1 = 0\ ;\ w_2 = 2$
l_2 (which translates into Ridge Regression) chooses $w_1 = 1\ ;\ w_2 = 1.5$
Obviously I'm oversimplifying, these won't be the actual chosen values for l_2. Ridge will choose similar values that will better minimize l_2, I just didn't list all the possible combinations for w_1 and w_2, but the important thing is that Lasso will actually go for $w_1 = 0; w_2 = 2$.

What have we noticed then?

- For Identical Features
  - l_1 regularization spreads weight arbitrarily (all weights same sign)
  - l_2 regularization spreads weight evenly
- For Linearly Related Features
  - l_1 regularization chooses the variable with the largest scale, 0 weight to the others
  - l_2 prefers variables with larger scale, it spreads the weight proportional to scale

(Sources: PoliMi Data Scientists Notes - Machine Learning ; Bloomberg - Lasso, Ridge, and Elastic Net)

Ridge Regression vs Bayesian Linear Regression

Describe the ridge regression algorithm and compare it with the Bayesian linear regression approach.
(William Bonvini) I've already described Ridge Regression previously.

Comparison:

Ridge Regression is a frequentist approach:
the model assumes that the response variable (y) is a linear combination of weights multiplied by a set of predictor variables (x). The full formula also includes an error term to account for random sampling noise.

What we obtain from frequentist linear regression is a single estimate for the model parameters based only on the training data. Our model is completely informed by the data: in this view, everything that we need to know for our model is encoded in the training data we have available.

Ridge Regression gives us a single point estimate for the output. However, if we have a small dataset we might like to express our estimate as a distribution of possible values. This is where Bayesian Linear Regression comes in.

The aim of Bayesian Linear Regression is not to find the single "best" value of the model parameters, but rather to determine the *posterior distribution* (*a probability distribution that represents your updated beliefs about the parameter after having seen the data*) for the model parameters.
Not only is the response generated from a probability distribution, but the model parameters are assumed to come from a distribution as well. The posterior probability of the model parameters is conditional upon the training inputs and outputs:

$$P(\beta|y,X) = \frac{P(y|B,X)P(\beta|X)}{P(y|X)}$$