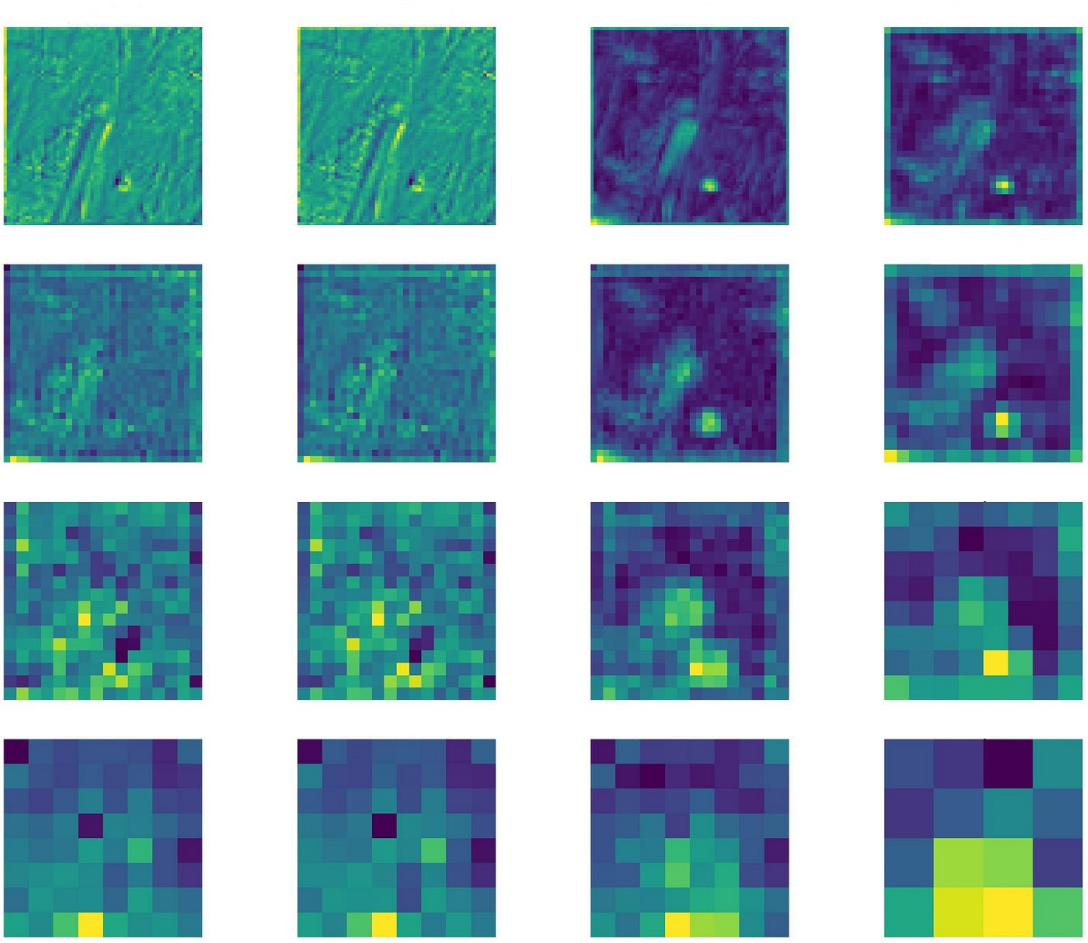
Machine Learning



Held by Prof. D. Loiacono at Politecnico di Milano 2023/2024

Notes by Rayan Emara

Table of contents

1. Disclaimers and preface

These notes were taken during AY 2023/2024 using older material, your mileage may vary. They're meant to accompany the lectures and in no way aim to substitute a professor yapping away at an iPad 30m away.

These notes are in part based on material by **Salvatore Buono**

For any questions/mistakes you can reach me here.

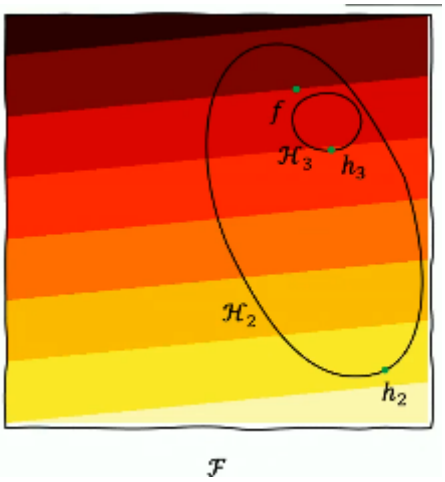
2. Overview of supervised learning

2.1. What kind of problem are we trying to solve ?

We're trying to find/approximate a function f, which is "generating" our dataset D.
The steps are:

- Define a loss function L
- Choose the hypothesis space $\mathcal{H}$
- Find in $\mathcal{H}$ an approximation h of f that minimizes L.

We might be tempted to enlarge the hypothesis space, but if we do that we can have a larger risk of approximating f very poorly as we don't know if we'd be expanding $\mathcal{H}$ in the correct direction.



2.2. The elements of a supervised learning algorithim

- The representation meaning the hypothesis space $\mathcal{H}$ and how it's designed.
- The evaluation, so how the loss function is designed.
- The optimization algorithim, how you're looking for candidate soltutions.

Some examples of representations are:

- Linear models.
- Neural networks.

Evaluation:

- Accuracy, the % of time you're correctly classifying a datapoint.
- If you have a regression problem you could use MSE.

Finally, the optimization technique really depends on your loss function:

- Gradient descent.
- Greedy search.
- Linear programming (in cases where you have some sort of constraint)

3. A supervised learning taxonomy

There are many ways you can try to classify/organize/compare, some more general paradigms other than the previous three.

3.1. Parametric vs Non-parametric

There are some learning algorithims for which the deisgn of the hypothesis space is such that when you decide to apply a learning algorithim to a problem you have to define a set of parameters that are fixed and won't change with the data these are called **Parametric**, while non parametric models scale the number of parameters with the training set.

3.2. Frequentist vs Bayesian

No meaningful definition given by the prof here.

3.3. Direct, discriminative or generative

They're different ways to see the same problem from different perspectives, in the **direct** approach you don't really care about the probabilstic interpretation, you're just optinmizing the loss function. In the **discriminative** you interpret your problem as a conditional density $p(t|x)$ you essentially try to learn the distribution and then compute the expected mean. In the **generative** tries to model the *joint* density $p(x,t)$, this allows to then infer the conditional density and generate novel samples by computing the conditional mean.

4. Linear regression

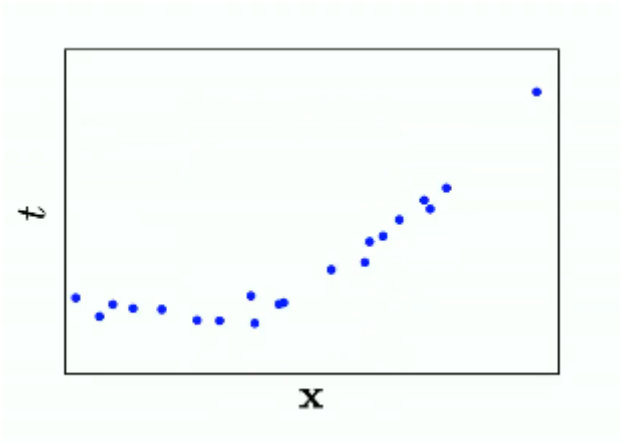
We'll focus on regression as our first type of problem, more specifically on linear models to solve.

References:

- *Pattern recognition and Machine learning*, Bishop

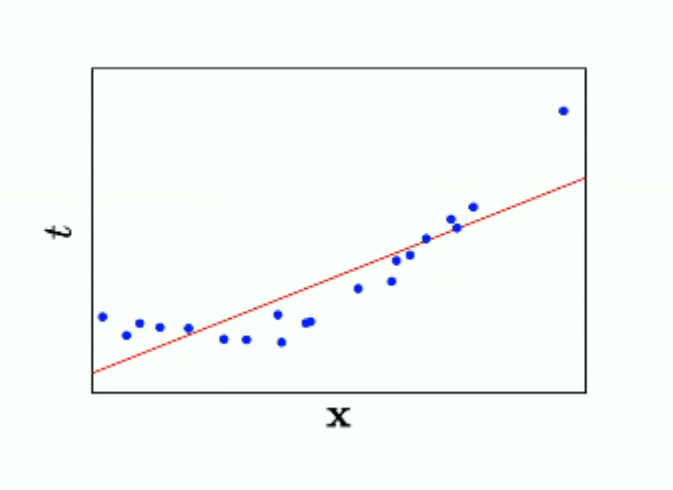
4.1. The model

We want to learn an approximation function $f(x)$:



We start from a dataset
$\mathcal{D} = < x, t > \implies t = f(x)$
How do we model f ? How do we evaluate our approximation ? How do we optimize our approximation ?

In linear regression we model $f(x)$ with linear functions.



A model like this clearly has room for improvement but we can veeery easily explain what the model is doing. Another important property is that we can solve this analitically, this is better since models like NN are hard to "debug", in this case nothing can go wrong, we have algorithims with guaranteed convergence etc...

Another thing is that these models can capture non linear interactions.

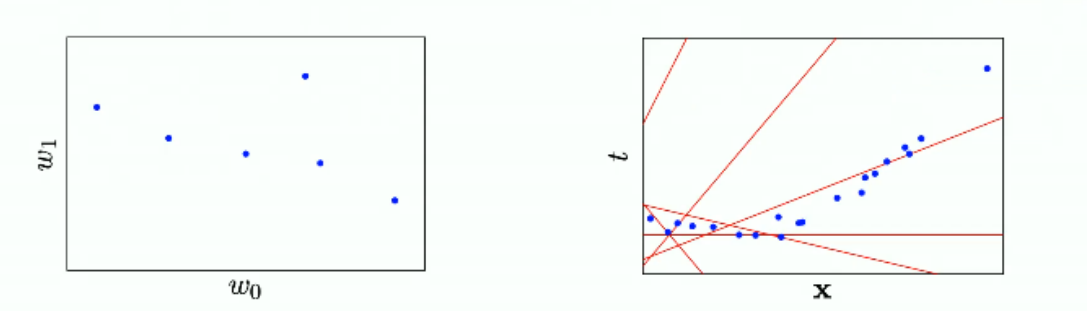
The simplest linear model can be defined as:

$$y(x, w) = w_0 + \sum_{j=1}^{D-1} w_j x_j = w^T x$$

Where:

- $x = (1, x_1, .., x_{D-1})$ is our input variable expressed as a vector to which we prepend 1.
- w_0 is called the bias parameter

My hypothesis space is 2D space with the two weights as dimensions.



Each point in the left image is a "solution" to our problem (a red line). We now need to *evaluate* these solutions, we need to define an error.

4.2. Loss function and optimization

A convenient error loss function is the sum of squared errors (SSE):

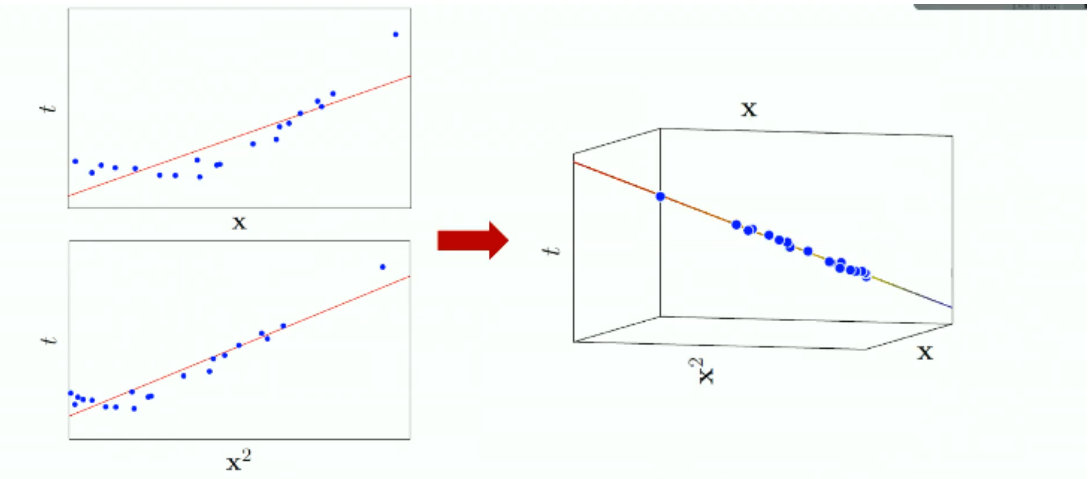
$$L(w) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, w) - t_n)^2$$

The reason we take the square instead of the absolute value is that ???

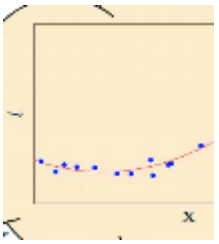
5. Linear models and basis functions

How do we capture non-linear relationships ? We can use basis functions. Instead of using $x_1, x_2, .. x_n$ we can use a set of non-linear functions ϕ_i (as many as we want), my linear model will be *linear* with respect to the basis functions (we're learning the weights).

In this example i generate 2 weights for each sample, one for x and for x^2.



If we now plot this approximation into the original space:



If we have problem specific knowledge we can pinpoint some sort of specific basis functions but generally there's no rule.
There are *families* of basis functions such as :

- Polynomial :

$$\phi_j(x) = x^j$$

- Gaussian :

$$\phi_j(x) = exp\left(-\frac{(x-\mu_j)^2}{2\sigma^2}\right)$$

- Sigmoidal:

$$\phi_j(x) = \frac{1}{1+exp\left(\frac{\mu_j - x}{\sigma}\right)}$$

This can be hyper-parameterezide (idk how to spell it), more on this later in the course.

6. Least squares

From now on we'll assume that our problem is solved in the feature space and not in the input space (in the basis function space in the previous case).

Prof emphasizes how important linear algebra is for this course...

$$L(w) = \frac{1}{2}RSS(w) - \frac{1}{2}(t-\phi w)^T(t-\phi w)$$

Where t is a vector with all the targets and ϕ is a matrix where on each row you have all the samples in the feature space (so on the cols you'd have features).

If you have to optimize this function with respect to a w. Let's assume it's a scalar value, how do we find the optimal value of w ?

Get the derivative and put the derivative equal to 0, right ?

Ordinary Least Squares

- For linear models, a closed-form optimization of the RSS, known as **least squares**, starting from the matrix form of the loss function:

$$L(\mathbf{w}) = \frac{1}{2} RSS(\mathbf{w}) = \frac{1}{2} (\mathbf{t} - \mathbf{\Phi w})^T (\mathbf{t} - \mathbf{\Phi w})$$

  - where $\mathbf{\Phi} = (\boldsymbol{\phi}(\mathbf{x}_1), \ldots, \boldsymbol{\phi}(\mathbf{x}_N))^T$ and $\mathbf{t} = (t_1, \ldots, t_N)^T$

- We can compute first a second derivative of $\mathcal{L}(w)$ to find the optimal w

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = -\mathbf{\Phi}^T (\mathbf{t} - \mathbf{\Phi w}) \qquad \frac{\partial^2 L(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^T} = \mathbf{\Phi}^T \mathbf{\Phi}$$

$$\Rightarrow \quad \hat{\mathbf{w}}_{OLS} = \left(\mathbf{\Phi}^T \mathbf{\Phi} \right)^{-1} \mathbf{\Phi}^T \mathbf{t}$$

Machine Learning — Daniele Loiacono

This is an analytical solution, the caveat is i need the matrix to not be singular cause then i'd have to invert it. It's also slow as fuck for large datasets.

You can instead apply **Gradient descent** (or SGD).

Sequential Learning

- Closed-form optimization (OLS) is not feasible with large dataset
- Instead, a **stochasitcs** (or **sequential**) gradient descent is possible
- **Least Mean Square** (LMS) algorithm:

$$L(\mathbf{x}) = \sum_n L(x_n)$$

$$\Rightarrow \quad \mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \alpha^{(n)} \nabla L(x_n)$$

$$\Rightarrow \quad \mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \alpha^{(n)} \left(\mathbf{w}^{(n)^T} \boldsymbol{\phi}(\mathbf{x}_n) - t_n \right) \boldsymbol{\phi}(\mathbf{x}_n)$$

  - α is called learning rate and to guarantee convergence:

$$\sum_{n=0}^{\infty} \alpha^{(n)} = +\infty \qquad \sum_{n=0}^{\infty} \alpha^{(n)^2} < +\infty$$

Machine Learning — Daniele Loiacono

Prof then talks about geometric interpretation of OLS (Ordinary least squares).

7. Multiple outputs

What if my target is a multitude regression targets ?

In practice it's just like running multiple regression problems in parallel, the main optimization is using the same basis functions.

8. Regularization

What happens if i use too many features ? (p.22 to 24 of 3).

We basically add a regularizing term to the loss function that takes into account how many features we added. We can extend the loss function to take into account the complexity of our model

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

where $L_W(\mathbf{w}$ accounts for model complexity and λ is the **regularization** coefficient and w is the vector of model coefficients (or weights).

We can design $L_W(\mathbf{w}$ in many ways, we'll look at two in particular

8.1. Ridge regression (L2 regularization)

We define $L_W(w)$ as

$$L_W(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2} \parallel \mathbf{w} \parallel_2^2$$

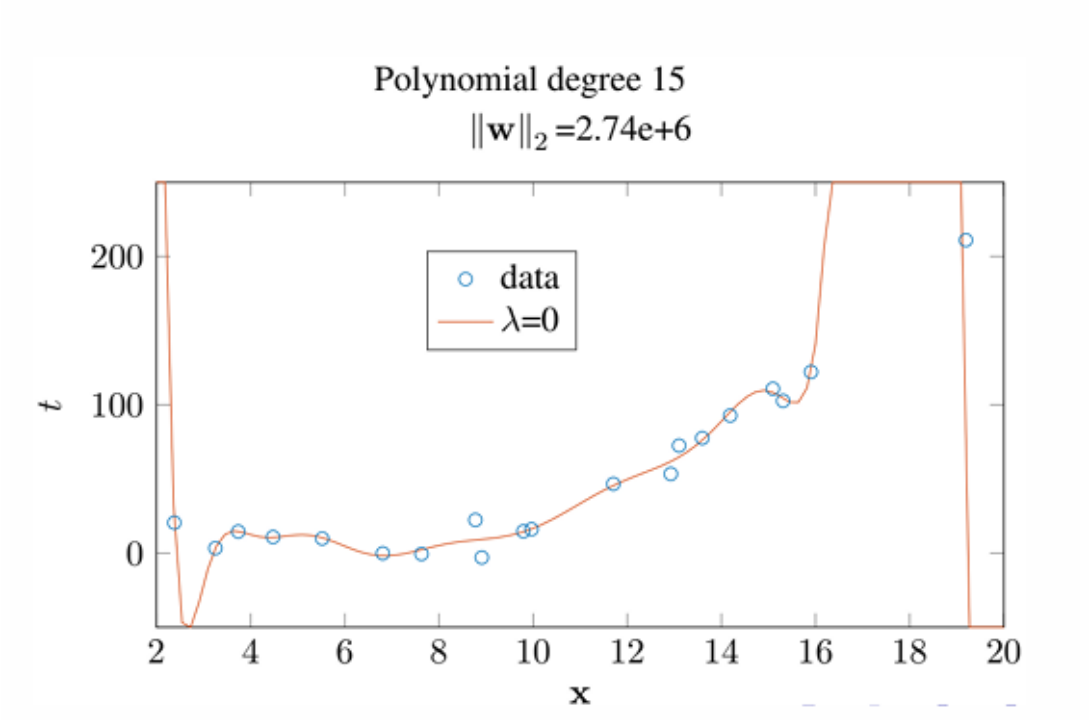
where the loss function $L(w)$ ends up being

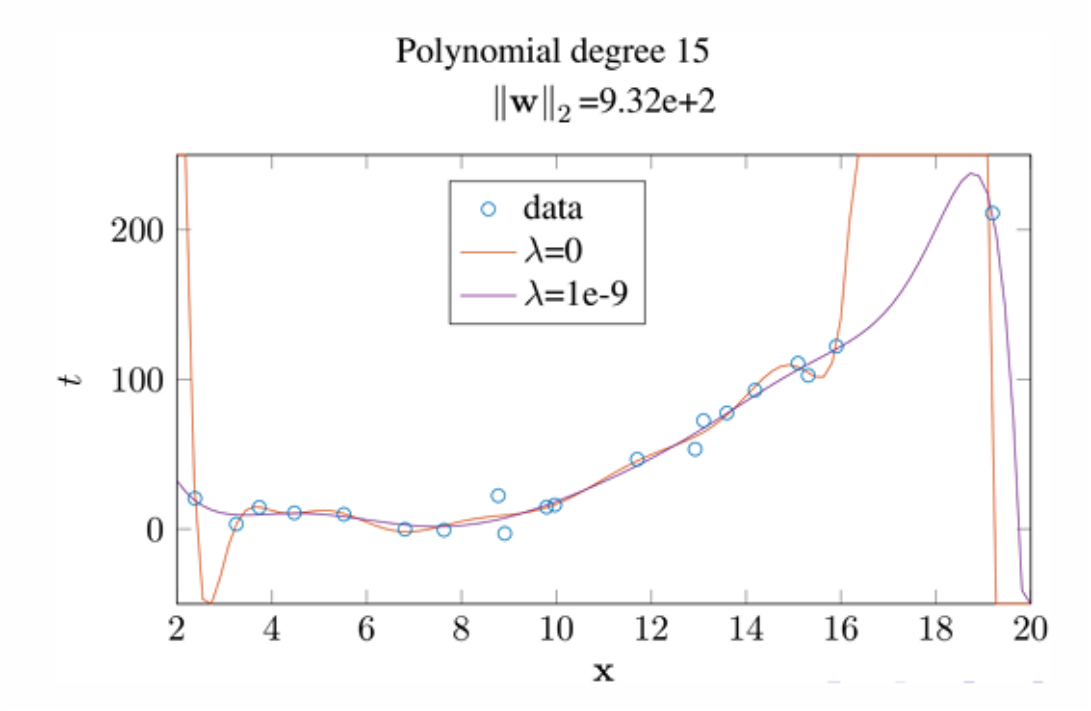
$$L\left(\mathbf{w}\right) = \frac{1}{2}\sum_{i=1}^{N}\left(t_i - \mathbf{w}^T\phi(\mathbf{x}_i)\right)^2 + \frac{\lambda}{2} \parallel \mathbf{w} \parallel_2^2$$

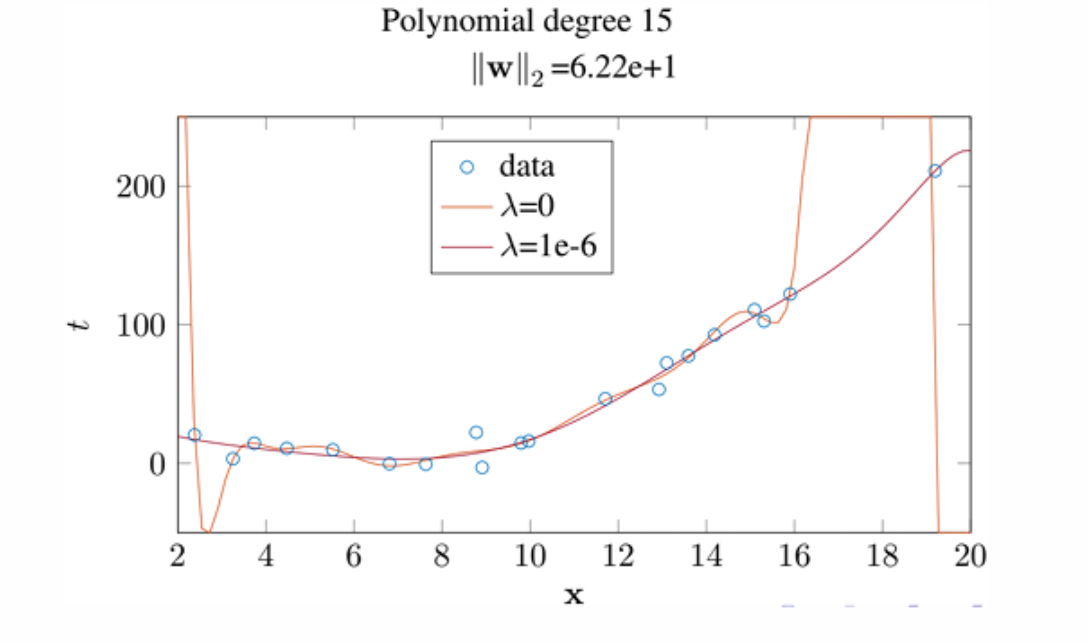
the closed form (estimator) is

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

We're essentially an L^2 penalty







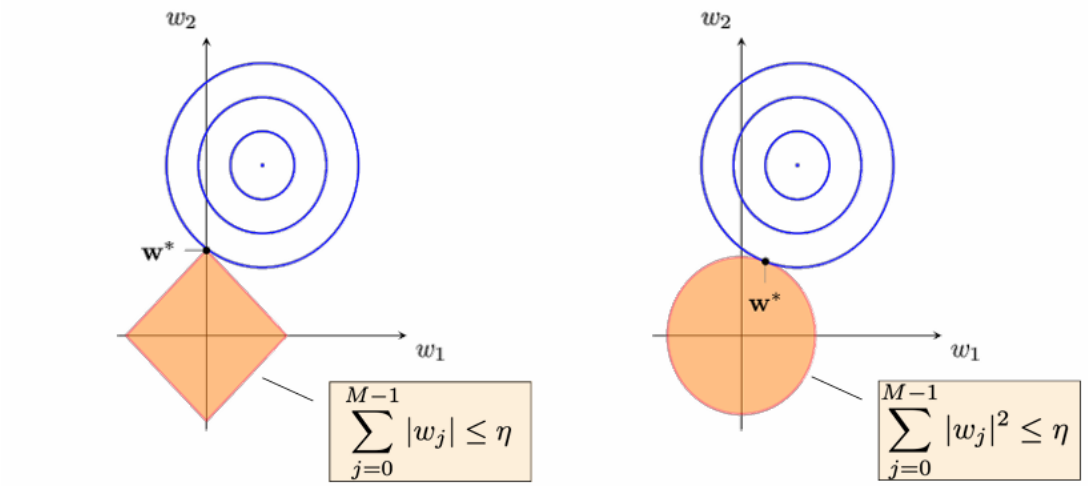
8.2. Lasso regression (L1 regularization)

Lasso stands for *least absolute shrinkage and selection operator*

$$L_W(\mathbf{w}) = \frac{1}{2} \| \mathbf{w} \|_1 = \frac{1}{2} \sum_{j=0}^{M-1} |w_j|$$

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

When λ is large enough some weights might be **equal to zero**. That's because we're basically creating a constraint region in the shape of a diamond (or a cross-polytope) in the weight space. When the loss function's contours (ellipsoids) are minimized under this constraint, the solution often occurs at the corners of the diamond.



9. Least squares and Maximum likelihood

We can also approach regression in a probabilistic way by defining a probabilistic model that maps input x to outputs t using some unknown parameters w

$$y(x, w)$$

we then model the **likelihood** i.e. the probability that observed data $\mathcal{D}$ is generated by a given set of parameters w

$$p(\mathcal{D}|w)$$

we then estimate those parameters by maximizing the likelihood that those w generated our observed sample

$$w_{ML} = \arg\max_w p(\mathcal{D}|w)$$

In the case of linear regression our model can be defined as:

$$t = y(x, w) + \epsilon = w^T\phi(x) + \epsilon$$

where $y(x, w)$ is taken to be a linear model disrupted by Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$
Given a dataset $\mathcal{D}$ of N samples with inputs $X = \{x_1, \ldots, x_N\}$ and outputs $T = \{t_1, \ldots, T_N\}^T$

$$p(\mathcal{D}|w) = p(\mathbf{t}|\mathbf{X}, w, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}\left(t_n|w^T\phi(\mathbf{x}_n), \sigma^2\right)$$

Since we're assuming the datapoints to be i.i.d. the likelihood function is the product of individual Gaussian distributions for each data points.
To find w_{ML} we usually prefer to deal with the **log**-likelihood since it is a strictly increasing map

$$\ell(w) = \ln p(\mathbf{t}|\mathbf{X}, w, \sigma^2) = \sum_{n=1}^{N} \ln p(t_n|\mathbf{x}_n, w, \sigma^2) = -\frac{N}{2}\ln\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}RSS(w)$$

we then set the gradient of this object to zero and find the solution to that equation

$$\nabla\ell(w) = \sum_{n=1}^{N} t_n\phi(\mathbf{x}_n)^T - w^T\left(\sum_{n=1}^{N}\phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^T\right) = 0$$

$$w_{ML} = \left(\mathbf{\Phi}^T\mathbf{\Phi}\right)^{-1}\mathbf{\Phi}^T\mathrm{t}$$

10. Bayesian Linear Regression

We first define the model given our current knowledge, we capture our assumptions by wrapping them into our a priori distribution (in teal) over some unknown parameters before seeing the data, we then observe the data and compute the posterior probability distribution (in red) for the parameters given the observed data.

$$p(parameters|data) = \frac{p(data|parameters)p(parameters)}{p(data)}$$

we then use the posterior distribution to

- Make predictions
- Account for uncertainty over the parameters

Example

The way this works is basically the concept of Bayesian learning, it allows us to update our beliefs after observing data.

Say we have a prior belief that a certain coins bias p follows a β distribution with parameters $\alpha = 2$ and $\beta = 2$, this means we think the coin is fairly balanced with a bias centered around $p = 0.5$.
Now let's say we flip the coin 10 times and observe that it lands heads 6 times and tails 4 times, using Bayes' theorem we compute the posterior distribution of the coin bias p given the observed data which will also be a β distribution (prove this hehe) with different parameters.

We've successfully employed Bayes' theorem to update our prior belief system, we can now use this posterior distribution to

- Predict the outcome of future coin flips by averaging over the posterior distribution of p
- Quantify our uncertainty about the coin bias p using summary statistics of the posterior distribution, such as the mean, median, or credible intervals
- U se the posterior distribution to calculate the expected loss under different decisions and choose the decision with the lowest expected loss.

The general formula is:

$$p(w|\mathcal{D}) = \frac{p(\mathcal{D}|w)p(w)}{p(\mathcal{D})}$$

where $p(w)$ is the prior probability over the parameter or what we know before observing the data, $p(\mathcal{D}|w)$ is the **likelihood** or the probability of observing the data ($\mathcal{D}$) given some value of the parameters (w).
We're essentially computing a probabilistic mean so we have to normalize the product on the numerator, that's where $p(\mathcal{D})$ comes

$$p(\mathcal{D}) = \int p(\mathcal{D}|w)p(w)\mathrm{d}w$$

We usually model the prior distribution using a Gaussian likelihood

$$p(w) = \mathcal{N}\left(w|w_0, \mathbf{S}_0\right)$$

this means that the posterior distribution result from the Bayes theorem is still a Gaussian distribution (known property of Gaussian distributions)

$$p(\mathbf{w}|\mathbf{t}, \mathbf{\Phi}, \sigma^2) \propto \mathcal{N}\left(\mathbf{w}|\mathbf{w}_0, \mathbf{S}_0\right)\mathcal{N}\left(\mathbf{t}|\mathbf{\Phi}\mathbf{w}, \sigma^2\mathbf{I}\right)$$

$$\begin{aligned} p(\mathbf{w}|\mathbf{t}, \mathbf{\Phi}, \sigma^2) &= \mathcal{N}\left(\mathbf{w}|\mathbf{w}_N, \mathbf{S}_N\right) \\ \mathbf{w}_N &= \mathbf{S}_N\left(\mathbf{S}_0^{-1}\mathbf{w}_0 + \frac{\mathbf{\Phi}^T\mathbf{t}}{\sigma^2}\right) \\ \mathbf{S}_N^{-1} &= \mathbf{S}_0^{-1} + \frac{\mathbf{\Phi}^T\mathbf{\Phi}}{\sigma^2} \end{aligned}$$

Note that:

- The posterior acts as prior for the next iteration in the case of sequential data.
- If the prior has infinite variance w_N converges to the **maximum likelihood** estimator
- If $\mathcal{S} \to \infty$ it converges to **ordinary least squares** instead.

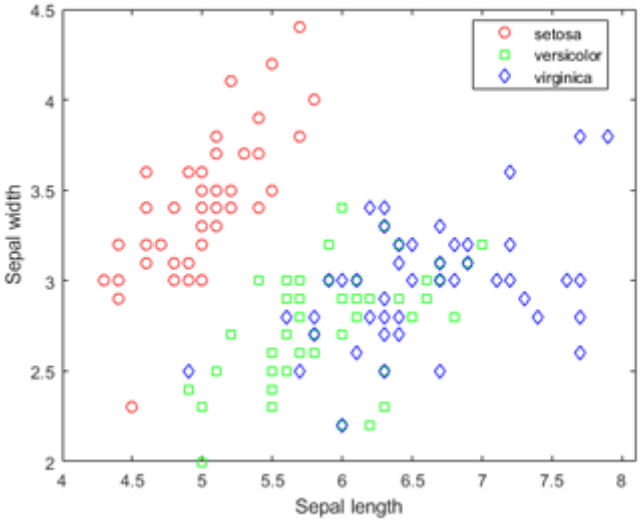
MISSING SOME STUFF HERE p.6 of Salvatore Buono study the part where OLS and ML

11. Linear classification

Given a dataset $\mathcal{D}$ we want to learn an approximation function $f(x)$ that maps input x to a discrete class C_k where $K = 1, \ldots, k$.

$$\mathcal{D} = \langle x, C_k \rangle \quad \longrightarrow \quad C_k = f(x)$$

We need to predict discrete class labels, more specifically the (linear) decision boundaries that divide the different class labels.
This type of classification is linear despite employing non-linear activation functions, the name comes from the linear decision boundaries



There are 3 main approaches:

- **Discriminant function:** we model a parametric *function* that directly maps the input to a class.
- **Probabilistic discriminative approach:** where we model $p(C_k|x)$ with respect to certain parameters and learn them from the training dataset (logistic regression).
- **Probabilistic generative approach (Bayesian):** where we model $p(C_k|x)$ and the prior $p(C_k)$ and infer the posterior distribution using Bayes' theorem

In linear classification we'll use **generalized linear models**:

$$f(\mathbf{x}, \mathbf{w}) = f\left(w_0 + \sum_{j=1}^{D-1} w_j x_j\right) = f(\mathbf{x}^T \mathbf{w} + w_0)$$

where $f(\cdot)$ is **not** linear in $\mathbf{w}$ because its output is a discrete value or alternatively a probability value. The $f(\cdot)$ effectively partitions the input

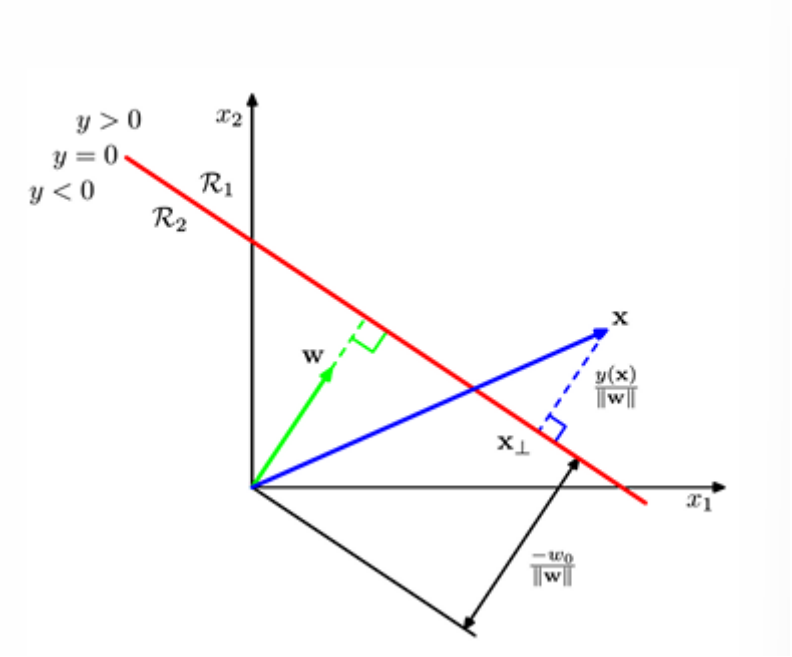
space into **decision boundaries** (**decision surfaces** in higher dimensions) which are linear functions of $\mathbf{x}$ and $\mathbf{w}$ as they satisfy

$$\mathbf{x}^T\mathbf{w} + w_0 = \text{const}$$

The most common choice for encoding for the general $k-$classes problem is **1-of-k** (or 1 hot encoding) encoding where $t \in B^k$ with B representing the set of 0 and 1.
With this encoding t and $f(\cdot)$ represent the density probability over the classes.
In the case of two-class problems we might opt for the natural $t \in B^0$, $t \in \{-1, 1\}$ is sometimes used for some algorithms.

The discriminant linear function for a two-class problem would be

$$f(\mathbf{x}, \mathbf{w}) = \begin{cases} C_1, & \text{if } \mathbf{x}^T\mathbf{w} + w_0 \geq 0 \\ C_2, & \text{otherwise} \end{cases}$$



where the the decision surface has the following properties:

- It's equal to

$$\mathbf{y}(\cdot) = \mathbf{x}^T\mathbf{w} + w_0 = 0$$

- It's orthogonal to $\mathbf{w}$
- The distance between the decision surface and the origin is

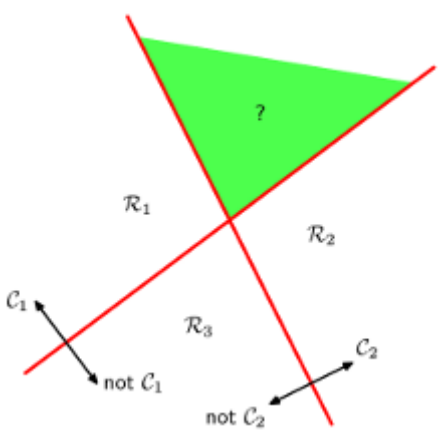
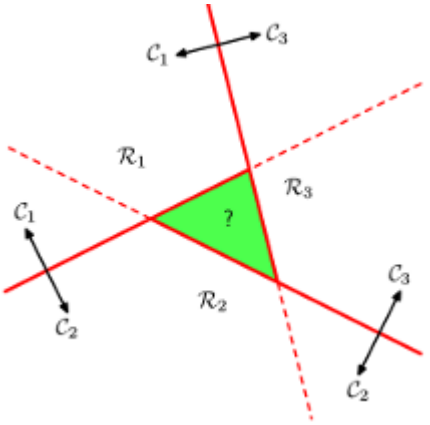
$$-\frac{w_0}{\|\mathbf{w}\|_2}$$

- And the distance between the **ds** and x is

$$\frac{y(\mathbf{x})}{\|\mathbf{w}\|_2}$$

But what if we have multiple classes ? We generally denote the number of classes with K.
We have two main approaches

- **One-versus-the-rest**: where we use $K-1$ binary classifiers where each classifier i discriminates between C_i and not C_i.
- **One-versus-one**: where we use $\frac{K(K-1)}{2}$ binary classifiers to discriminate between C_i and C_j

One-versus-the-rest	One-versus-one
	

These approaches are not a good idea if one region is mapped to more than one class, in that case we might elect to use K linear discriminant functions:

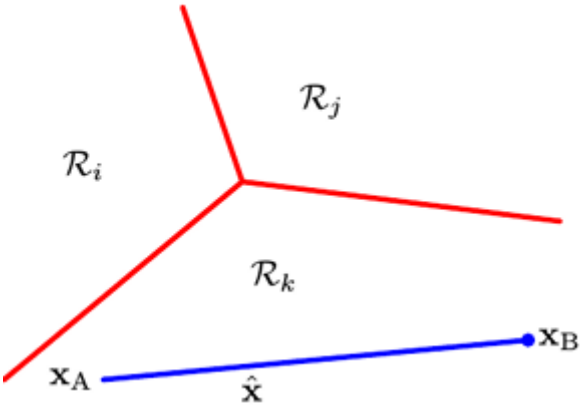
$$y_k(\mathbf{x}) = \mathbf{x}^T\mathbf{w}_k + w_{k0}, \text{ where } k = 1, \dots, K$$

where we're basically mapping $\mathbf{x}$ to C_k if $y_k > y_j \quad \forall j \neq k$.
This approach presents no ambiguity for classes belonging to the same regions.

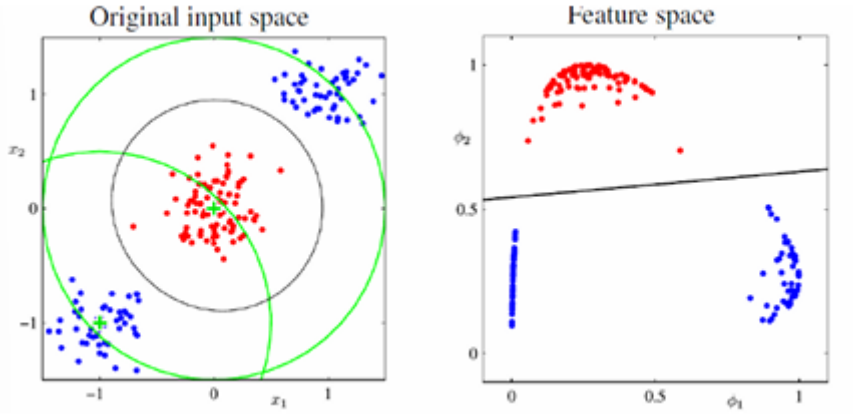
Theorem

Let $\mathbf{x}_A, \mathbf{x}_b \in \mathcal{R}_k$
Thus $y_k(\mathbf{x}_A) > y_j(\mathbf{x}_A)$ and $y_k(\mathbf{x}_B) > y_j(\mathbf{x}_B)$, therefore $\forall\alpha$ such that $0 < \alpha < 1$:

$$y_k(\alpha\mathbf{x}_A + (1-\alpha)\mathbf{x}_B) > y_j(\alpha\mathbf{x}_A + (1-\alpha)\mathbf{x}_B)$$



Keep in mind that all of these models are applied to the problem input space, we can extend these models by using basis functions $\varphi(\mathbf{x})$, this has the effect of potentially admitting decision boundaries that are linear in the **feature space** and non linear in the input space.



11.1. Least Squares for Classification

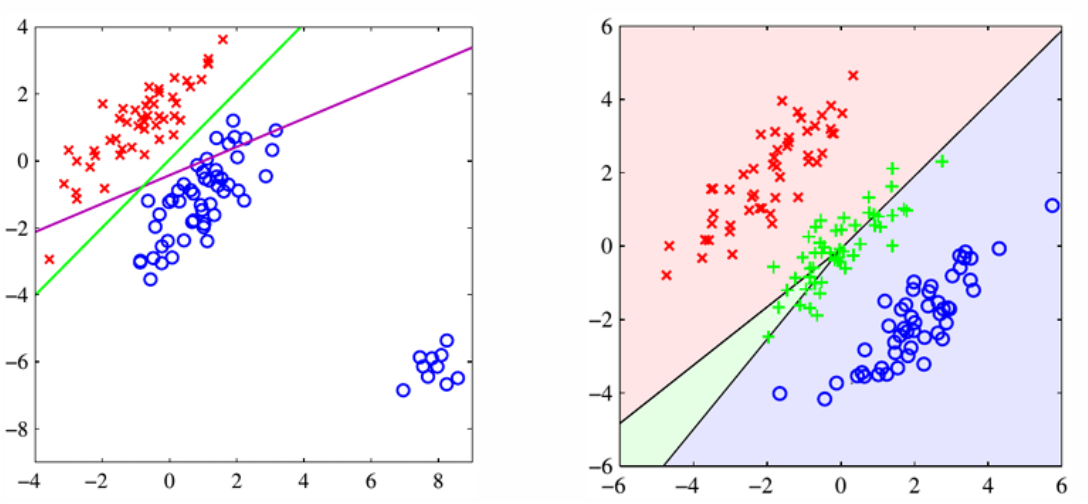
We consider a K-class problem with a 1-of-K encoding where each class is modeled with a linear function

$$y_k(\mathbf{x}) = \mathbf{x}^T\mathbf{w}_k + w_{k0}, \text{ where } k = 1, \ldots, K$$

or

$$\mathbf{y}(\mathbf{x}) = \tilde{\mathbf{W}}^T\tilde{\mathbf{x}}$$

where $\tilde{\mathbf{W}}$ has size $(D+1)\cdot K$ and the k-th column of $\tilde{\mathbf{W}}$ is $\widetilde{\mathbf{w}_k} = \left(w_{k0}, \mathbf{w}_k^T\right)^T$.
The main problem with this approach is its sensibility to outliers, as the error for these elements will move the decision boundaries far more than other elements.



11.2. Perceptron

The perceptron is an **online** (computes and updates one sample at a time), **linear** discriminant model. The algorithm tries to find the decision boundary by minimizing the distance of misclassified points to the decision boundary:

- Correct classifications get 0 error
- Misclassified points x_n get error $w^T\phi(x_n)t_n$

The algorithm effectively optimizes this function to be minimal:

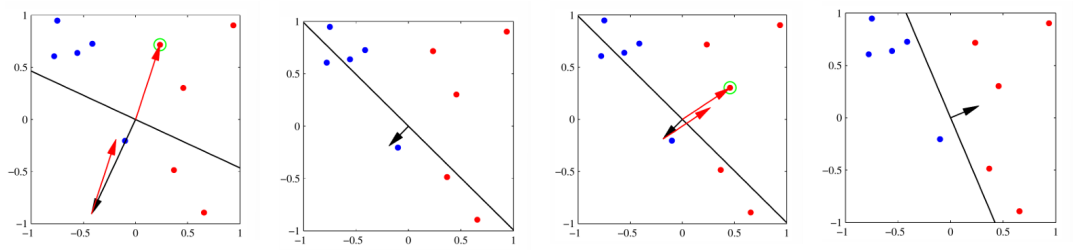
$$L_P(\mathbf{w}) = -\sum_{n\in\mathcal{M}} \mathbf{w}^T\phi\left(\mathbf{x}_n\right)t_n$$

the optimization (minimization) can be performed by using stochastic gradient descent, note how correctly classified samples do not contribute to the loss. We can update one sample at a time using the following where the learning rate α is usually set to 1

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha\nabla L_P(\mathbf{w}) = \mathbf{w}^{(k)} + \alpha\phi(\mathbf{x}_n)t_n$$

```
k = 0
repeat
    k = k+1
    n = k mod N
    if model_output != target then
        w(k+1) = w(k) + phi(x_n)*target
    end if
until convergence
```

where the dataset $\mathcal{D} = \{x_i, t_i\}\ \forall i = 1, \ldots, N$.



1	2	3	4

it's important to note that while a single iteration will reduce the error for the single misclassified sample, this does **not** guarantee that the entire loss is reduced after each update.

Perceptron convergence theorem

If the training dataset is linearly separable in the feature space φ, then the perceptron learning algorithm is guaranteed to find an **exact solution** in a finite number of step.
There's no guarantee on the number of steps or the uniqueness of the solution.

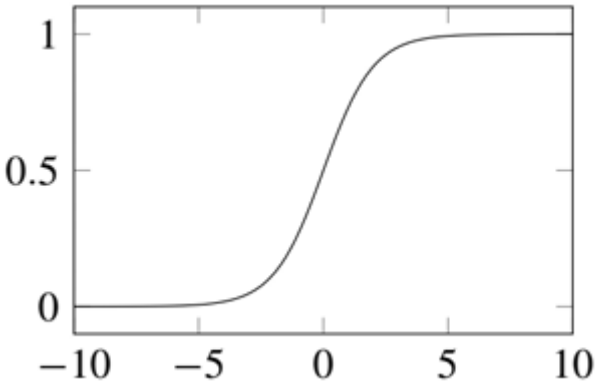
11.3. Logistic regression (two-class)

In this scheme we aim to model the conditional probability $p(C_k|\phi)$ directly

$$p\left(C_1 \mid \phi\right) = \frac{1}{1+\exp\left(-\mathbf{w}^T\phi\right)} = \sigma\left(\mathbf{w}^T\phi\right)$$

$$p\left(C_2 \mid \phi\right) = 1 - p\left(C_1 \mid \phi\right)$$

where $\sigma(a)$ is known as the sigmoidal function.



Given a dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}$ where $i = 1, \ldots, N$ and $t_i \in \{0, 1\}$ we model the

likelihood of a single sample using a **Bernoulli** distribution, using the logistic regression model for conditioned class probability

$$p(t_n|\mathbf{x}_n, \mathbf{w}) = y_n^{t_n}(1-y_n)^{1-t_n} \quad \text{where} \quad y_n = p(t_n = 1|\mathbf{x}_n, \mathbf{w}) = \sigma(\mathbf{w}^T\phi_n)$$

we use maximum likelihood to maximize the probability of correctly classifying the samples:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}(1-y_n)^{1-t_n} \quad , \quad y_n = \sigma(\mathbf{w}^T\phi_n)$$

where

- y_n is the value predicted
- t_n is the real value (sometimes referred to as target or reals)

At this point we can opt to use the negative log-likelihood as a loss function to minimize (this is also known as the **cross-entropy error function**)

$$\begin{aligned} L(\mathbf{w}) &= -\ln\Big(p(\mathbf{t}|\mathbf{X}, \mathbf{w})\Big) \\ &= -\sum_{n=1}^{N}\Big(t_n \ln(y_n) \; + \; (1-t_n)\ln(1-y_n)\Big) \\ &= +\sum_{n=1}^{N} L_n \end{aligned}$$

The gradient for this loss function can then be derived as follows:

$$\begin{aligned} \frac{\partial L_n}{\partial y_n} &= -\left(-\frac{t_n}{y_n} - \frac{(1-t_n)}{1-y_n}\right) = \frac{y_n - t_n}{y_n(1-y_n)} \\ \text{where } \frac{\partial y_n}{\partial \mathbf{w}} &= y_n(1-y_n)\phi_n \\ \frac{\partial L_n}{\partial \mathbf{w}} &= \frac{\partial L_n \partial y_n}{\partial y_n \partial \mathbf{w}} = (y_n - t_n)\phi_n \Longrightarrow \nabla L(\mathbf{w}) = \sum_{n=1}^{N}(y_n - t_n)\phi_n \end{aligned}$$

Note that it has the same form as the sum of squared errors (SSE) for linear regression.
Unfortunately there's no closed form to be derived. You can optimize it using gradient-based techniques.

11.4. Logistic regression (multi-class)

In the multi-class we represent the posterior probabilities using a **softmax** transformation of linear functions of feature variables:

$$p(C_k|\phi) = y_k(\phi) = \frac{\exp(\mathbf{w}_k^T\phi)}{\sum_j \exp(\mathbf{w}_j^T\phi)}$$

Take a moment to understand what's happening here,

- $\mathbf{w_k}$ is the weight vector for class C_k. It is a column vector of the same dimension as the feature vector ϕ
- ϕ is the feature vector of an input sample. It is also a column vector

The dot product $\mathbf{w}_k^T\phi$ represents a linear combination of the input features ϕ, where each feature ϕ_i is weighed by its corresponding weight w_k^i. We can think of this geometrically as a projection, we're measuring how close the feature space is to the weight space (what we're trying to learn and consequentially predict the feature space from).
This score is also known as the **logit** score for K.
At this point we're ready to compute the likelihood assuming a 1-of-K encoding

$$p(\mathbf{T}|\mathbf{\Phi}, \mathbf{w}_1, \ldots, \mathbf{w}_K) = \prod_{n=1}^{N} \underbrace{\left(\prod_{k=1}^{K} p(C_k|\phi_n)^{t_{nk}} \right)}_{\substack{\text{One term corresponding} \\ \text{to correct class}}} = \prod_{n=1}^{N} \left(\prod_{k=1}^{K} y_{nk}^{t_{nk}} \right)$$

which we can minimize using the cross-entropy error function for which we can then compute the gradient for each weight

$$L(\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\ln p(\mathbf{T}|\mathbf{\Phi}, \mathbf{w}_1, \ldots, \mathbf{w}_K) = -\sum_{n=1}^{N} \left(\sum_{k=1}^{K} t_{nk} \ln g_{nk} \right)$$

$$\nabla L_{\mathbf{w}_j}(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{n=1}^{N} (y_{nj} - t_{nj})\boldsymbol{\phi}_n$$

There's no closed form solution due to the non-linearity, the error function is convex and can be optimized by standard gradient-based optimization techniques.

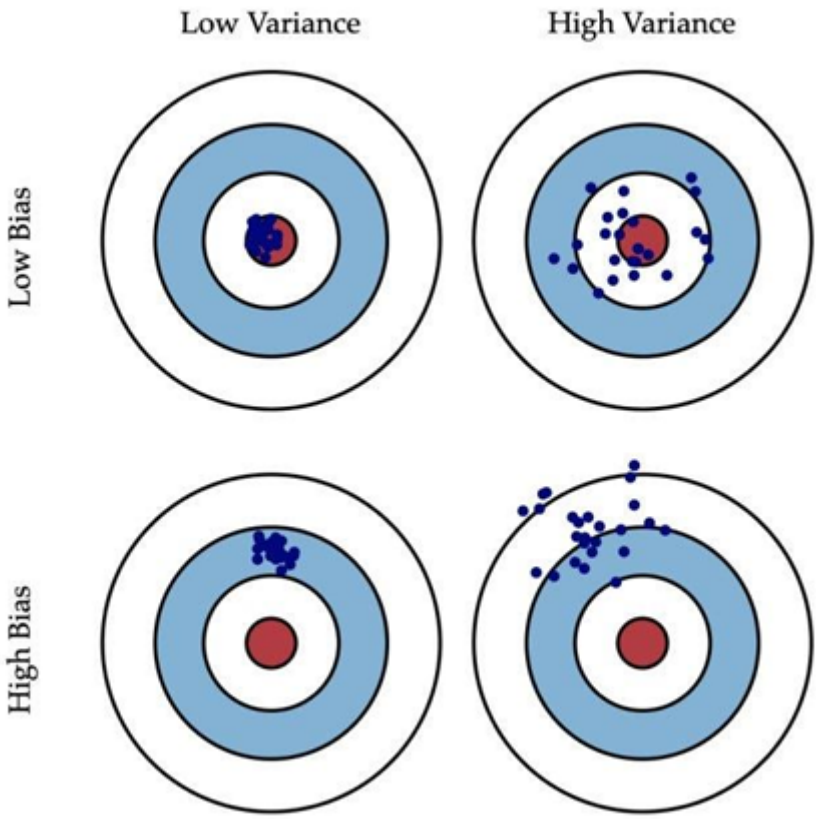
One might think that a step function would better represent a sample belonging to a single class. The step function, however, is non-differentiable making it unsuitable for gradient-based optimization, instead it would require an alternative optimization technique such as a combinatorial or evolutionary algorithm.

12. Model evaluation and selection

12.1. Bias-variance decomposition

Let $t_i = f(x_i) + \epsilon_i$ be our data points assuming $\mathbb{E}[\varepsilon] = 0$ and $Var[\varepsilon] = \sigma^2$, we'll denote our model with $\hat{t_i} = y(x_i)$ learned from *dataset* $\mathcal{D} = \{x_i, t_i\}$. Finally we'll take $\mathbb{E}\left[(t - y(\mathbf{x}))^2\right]$ to be our performance measure. We can decompose the **expected square error** as

$$
\begin{aligned}
\mathbb{E}[(t - y(\mathbf{x}))^2] &= \mathbb{E}[t^2 + y(\mathbf{x})^2 - 2ty(\mathbf{x})] \\
&= \mathbb{E}[t^2] + \mathbb{E}[y(\mathbf{x})^2] - \mathbb{E}[2ty(\mathbf{x})] \\
&= \mathbb{E}[t^2] \pm \mathbb{E}[t]^2 + \mathbb{E}[y(\mathbf{x})^2] \pm \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})] \\
&= Var[t] + \mathbb{E}[t]^2 + Var[y(\mathbf{x})] + \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})] \\
&= Var[t] + Var[y(\mathbf{x})] + (f(\mathbf{x}) - \mathbb{E}[y(\mathbf{x})])^2 \\
&= \underbrace{Var[t]}_{\sigma^2} + \underbrace{Var[y(\mathbf{x})]}_{\text{Variance}} + \underbrace{\mathbb{E}[f(\mathbf{x}) - y(\mathbf{x})]^2}_{\text{Bias}^2}
\end{aligned}
$$



In essence the model **variance** measures the inherent difference between models learned from different datasets. For the more experienced ML student it can be thought of as the over/under of how underfit or overfit your model is. It naturally decreases with simpler models which will require fewer samples to saturate, consequentially it will also decrease given more samples.

$$
\text{variance} = \int \mathbb{E}\left[(y(\mathbf{x}) - \overline{y}(\mathbf{x}))^2\right] p(\mathbf{x}) \mathrm{d}\mathbf{x}
$$

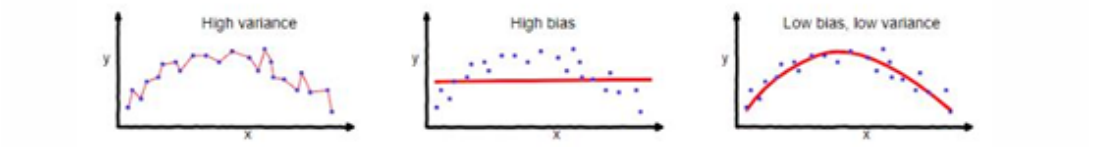
where

$$
\overline{y}(\mathbf{x}) = \mathbb{E}[y(\mathbf{x})]
$$

the model **bias** on the other hand measures the difference between the ground truth and what you can expect your model to learn, you can sort of assimilate this definition to that of *estimator bias*, where, given the estimator analytical form, we can compute the expected value and compare it to the analytical form of what we're trying to estimate. Think of the mean estimator for the beta distribution and how you have to correct it to get the

actual mean.
Bias happens when one of our learning hypotheses is wrong, it decreases with more complex models, you can think of it as a sort of measure of how correct your assumptions are (trying to fit a linear regressor on data from a quadratic function)

$$\text{bias}^2 = \int (f(\mathbf{x}) - \bar{y}(\mathbf{x}))^2 p(\mathbf{x}) \mathrm{d}\mathbf{x}$$



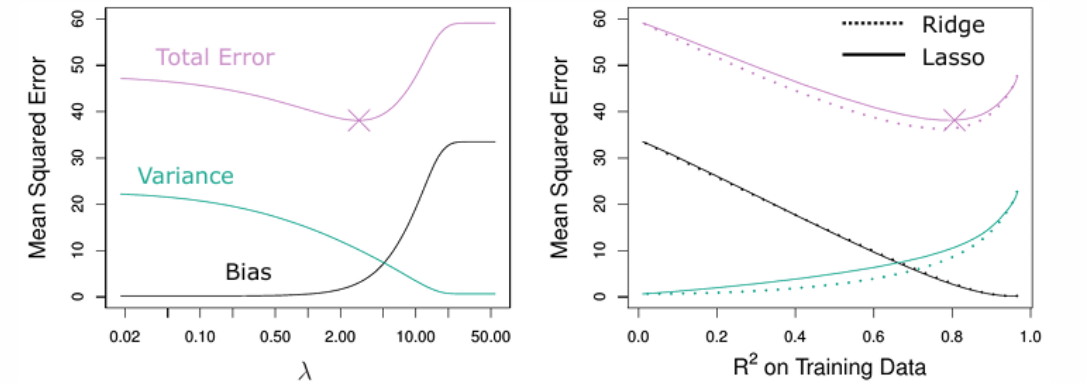
Finally **data noise** (σ^2) is the variance of data itself and is *irreducible*.

The professor proposes a case-study on K-NN, if we analyse the **MSE** we get

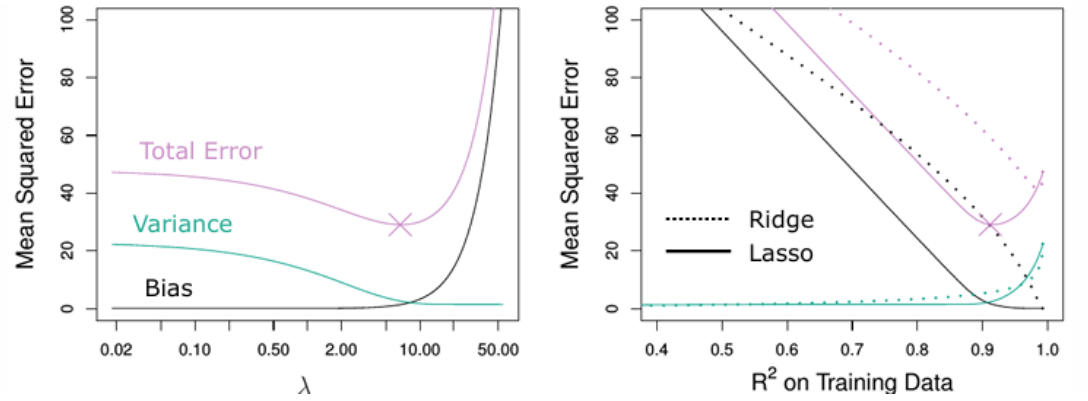
$$\mathbb{E}[(t^* - y(\mathbf{x}^*))^2] = \sigma^2 + \frac{\sigma^2}{K} + \left(f(\mathbf{x}^*) - \frac{1}{K} \sum_{i=1}^{K} f(\mathbf{x}_i) \right)^2$$

where the data noise σ^2 is the *irreducible* error we talked about earlier, the model **variance** is $\frac{\sigma^2}{K}$ decreases as K increases and finally the **bias** increases as K increases. This last part, while not as intuitive, is a direct effect of the inherent smoothness of high K value models. This smoothing effect means that the model gives more weight to a larger number of neighbours, effectively averaging their labels. While this reduces the model's variance, it also makes the model less sensitive to the nuances in the data. The larger the K, the more the model averages out the local variations, leading to a higher bias because it oversimplifies the true patterns in the data. Another reason for it is what we previously discussed as a possible reason for high bias in our models i.e. a wrong hypothesis, K-NN relies on the assumption that nearby points in the feature space are likely to belong to the same class. When K is small, the model focuses on very local structures, capturing the fine details of the data. As K increases, the model starts considering neighbours that are further away, which may belong to different classes. This causes the model to generalize more, thus missing local patterns and increasing bias.

Keep in mind that not all bias is bad, this is where regularization kicks in. We add bias as a way to avoid overfitting our model on unseen data.



45 of 45 features correlated to output



2 of 45 features correlated to output

Generally speaking Lass regularization outperforms Ridge when few features are related to the output.

12.2. Model selection and assessment in practice

Given $\mathcal{D} = \{\mathbf{x}_i, t_i\}$ with $i = 1, \ldots, N$ we can select the best model based on the loss function L computed on $\mathcal{D}$, this is called the **training error** and is not a good measure of the **prediction error** which is the real world performance of our model on unseen data

- Regression

$$L_{true} = \int \int (t - y(\mathbf{x}))^2 p(\mathbf{x}, t) \mathrm{d}\mathbf{x} \mathrm{d}t$$

- Classification

$$L_{true} = \int \int I\left(t \neq y(\mathbf{x})\right) p(\mathbf{x}, t) \mathrm{d}\mathbf{x} \mathrm{d}t$$

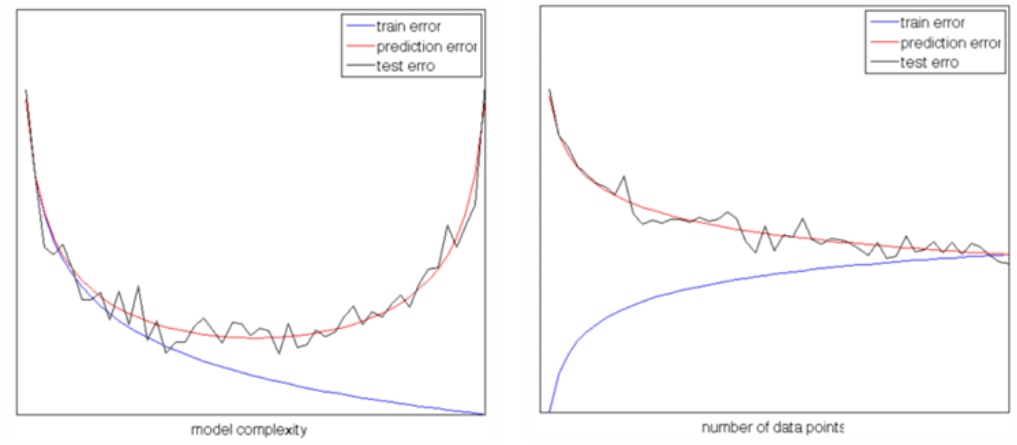
Unfortunately we usually don't have a good model for $p(\mathbf{x}, t)$ therefore we introduce the concept of a subset of $\mathcal{D}$ reserved for testing our model on unseen data, a **test set** on which we compute the **test error**

- Regression

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (t_n - y(\mathbf{x}_n))^2$$

- Classification

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (I(t_n \neq y(\mathbf{x}_n)))$$



The test set error as an estimator for prediction error

Generally speaking, train-test error analysis can help us identify

- **High-bias**: where the training and test error are close but higher than expected.
- **High-variance**

Validation set

In order to avoid over-fitting the test set we introduce the validation set, the steps are as follows:

- Use **training data** to learn model parameters
- For each model learned use **validation data** to compute the validation error
- We select the model with the lowest **validation error** and finally use **test data** to estimate the prediction error

This, of course can lead to unreliable model selection whenever the validation data is not plentiful enough, on the other hand we might end up over-fitting the validation set and not choose the best model.
In order to avoid validation over-fitting we can opt to use one of the following schemes to "shuffle" the validation/training partitions

Leave-One-Out cross validation (LOO)
For each sample $\{\mathbf{x}_i, t_i\} \in \mathcal{D}$ we train the model once on $\mathcal{D} \setminus \{\mathbf{x}_i, t_i\} \in \mathcal{D}$ and then compute the error of the resulting model on $\{\mathbf{x}_i, t_i\} \in \mathcal{D}$
The prediction error can then be estimated as the average of all the error computed using a single sample

$$L_{LOO} = \frac{1}{N}\sum_{i=1}^{N}(t_i - y_{\mathcal{D}_i}(\mathbf{x}_i))^2$$

where $t_{\mathcal{D}_i}$ is the model trained on $\mathcal{D} \setminus \{\mathbf{x}_i, t_i\} \in \mathcal{D}$

This scheme provides an **almost unbiased** estimate of the prediction error, it is however extremely expensive especially on larger datasets

K-Fold cross validation
We randomly split the training data $\mathcal{D}$ into k folds: $\mathcal{D}_1, \ldots, \mathcal{D}_k$.
For each fold $\mathcal{D}_i$ we train model on $\mathcal{D} - \{\mathcal{D}_i\}$ and then compute the error on $\mathcal{D}_i$

$$L_{\mathcal{D}_i} = \frac{k}{N}\sum_{(\mathbf{x}_n, t_n)\in\mathcal{D}_i}(t_n - y_{\mathcal{D}\setminus\mathcal{D}_i}(\mathbf{x}_n))^2$$

finally we estimate the prediction error as the average error computed

$$L_{k-fold} = \frac{1}{k}\sum_{i=1}^{k} L_{D_i}$$

L_{k-fold} provides a **slightly biased** estimate of the prediction error but it is much cheaper with low k values (around 10)
Other metrics can also be used to evaluate models based on their complexity:

- Mallows's

$$C_p : \; C_p = \frac{1}{N}(RSS + 2M\hat{\sigma}^2)$$

- Akaike Information Criteria:

$$AIC = -2lnL + 2M$$

- Bayesian Information Criteria:

$$BIC = -2\ln L + M\ln(N)$$

- Adjusted R2:

$$AdjustedR^2 = 1 - \frac{RSS/(N-M-1)}{TSS/(N-1)}$$

where M is the number of parameters, N is the number of samples, $\hat{\sigma}^2$ is the estimate of noise variance, RSS is the residual sum of squares and finally TSS is the total sum of squares.
AIC and BIC are generally used when maximizing the log-likelihood while BIC will generally penalize more than AIC model complexity.

12.3. Dealing with model complexity

A common pitfall is to think that using more features is always better, in fact one might argue that trying to fit

$$y = w_0 + w_1 x + w_2 x^2$$

is better than

$$y = w_0 + w_1 x$$

since we can always set $w_2 = 0$ but this actually increases the probability of over fitting the data and implies larger model variance since we might not have enough data to infer that $w_2 \approx 0$. With this in mind we aim to reduce the model variance in one (or more) of three common ways:

- **Feature selection**: we should design the feature space by selecting the most effective subset of all the possible features
- **Dimensionality reduction**: The input space can be mapped to lower dimensional space
- **Regularization**: Shrink parameter values towards 0

Feature selection

One approach for **feature selection** might be to brute force all possible feature combinations but this is obviously overly expensive in terms of compute. We generally can elect to use one of 3 common practices:

- **Filtering** where features are ranked on some evaluation metrics (e.g. correlation, variance etc..) and select the top k, this can be very fast but fails to capture any subset of mutually dependent features (it might make our model worse !)
- **Embedded**: we select (read turn off) features as we train using Lasso or Decision trees, this naturally captures feature relationships and is guaranteed not to make our model worse but can be very computationally expensive

- **Wrapper**: a search algorithm is used to find a subset of features that are evaluated by training a model with them and assessing its performance, can be implemented in one of two ways
  - Forward selection: starting from an empty model and adding features one at a time
  - Backwards elimination: where we start with all the features and remove them one at a time.

Dimensionality reduction

The main difference between feature selection and dimensionality reduction is that the latter uses **all** the features and maps them (through combinations) into a lower-dimensionality space, there are a lot of ways to implement this but we'll focus on two main ones.

Principal component analysis aims to find an orthonormal base of input which accounts for most of the variance

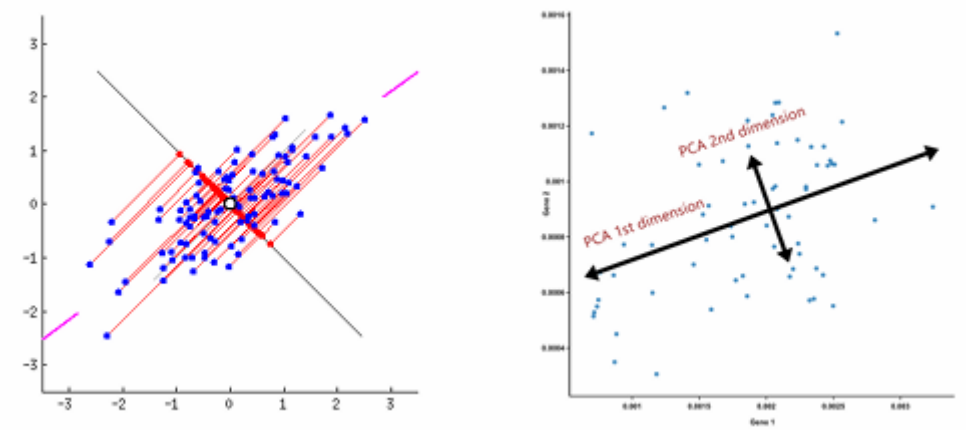
1. We standardize the data in order to allow each feature to contribute equally
2. Compute the covariance matrix on the standardized data in order to capture how much each pair of features vary together

$$S = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \bar{x})(x_n - \bar{x})^T$$

3. Compute the eigenvalues and eigenvectors for S, then the eigenvector with the largest eigenvalue will be the first principal component (**PC**) and so on.
   The k-th largest eigen value will have a proportion of variance captured:

$$\frac{\lambda_k}{\sum_i \lambda_i}$$

At this point we've built a new **covariance-ordered** orthonormal basis for the feature space where ALL the features are taken into account, allowing us to select the first k PCs to build a reduced dimensional representation of the data.
One can think of this approach as a way to build a basis of linearly independent vectors that represent a certain space.



In Fig.1 we find the highest variance and set it as PC, in Fig.2 we then find the second dimension

So far our m.o. has been to trade lower variance for higher bias (which, up to some extent, is a good thing), but is it possible to reduce model variance without increasing bias ?

12.4. Ensemble models

We can achieve lower variance without increasing by learning several models and combining them, the following is a set of methods for implementing ensemble learning

Bagging (Bootstrap aggregating)

Assume we have N datasets from which we learn N models $y_1, y_2, \ldots, y_N$, we can define a new aggregate model by taking the sample mean

$$y_{\text{AGG}} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

if the datasets are **independent** the model variance of the aggregate model y_{AGG} will $\frac{1}{N}$ of the i-th model, we can prove this by taking random variable $\bar{x} = \frac{1}{N}\sum_N x$ for which the variance will be as follows.

$$Var(\bar{x}) = \frac{1}{\mathrm{N}^2}\sum_N Var(x) = \frac{1}{N}Var(x)$$

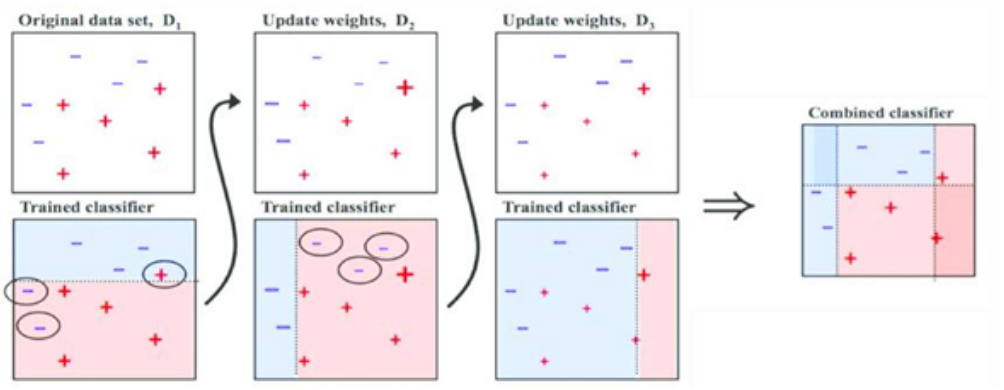
In practice, we generally don't have N independent datasets, therefore we can implement **bagging** where we generate N datasets applying **random sampling with replacement**, we then train a model on each dataset generated. At this point we can compute the prediction by sampling all the trained models and combining the output using **majority voting** for classification models and **averaging** for regression models.
This scheme *can* reduce variance even if the sampled datasets are not independent, it's unlikely to help if the model is robust to changes in the training data (high model bias), so it won't help with bad model

architectures, it can however help when dealing with cases of overfitting or unstable learners

Boosting

The key idea behind boosting is to **iteratively** train a series of weak learners, with each iteration focusing on the samples that were misclassified by the previous iteration, the main boosting algorithm is called **AdaBoost**

1. Weigh all training samples equally
2. Train the model the said training set
3. Compute the model error on the training set
4. Increase weights on misclassified samples
5. Re-compute the errors on weighted training set
6. Increase weights again on misclassified cases
7. Repeat steps 5 and 6
   The final model will be a **weighted prediction of each model**



Boosting a classifier

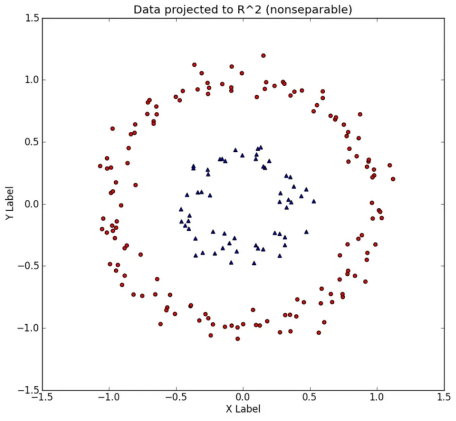
The main differences from bagging are that boosting trains models **sequentially**, uses random subsets of data with replacement, whereas boosting adjusts the focus on misclassified instances, bagging primarily reduces variance, whereas boosting reduces **both** bias and variance, bagging averages or votes on the predictions, while boosting uses a **weighted combination** of predictions.

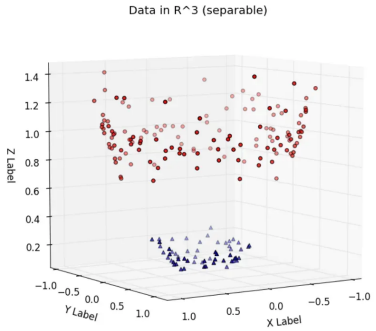
In conclusion

Scenario	Bagging	Boosting
Reduces variance	Yes	Yes but not as main effect
Reduces bias	No	Yes, without overfitting
Works well with stable models	No	Might help
Performs well with noisy data	Yes	No
Training type	Can be parallelized	Inherently sequential
Helpfulness	Will generally help but performance boost might be tiny	Can break or make the model

13. Kernel models

We often want to capture **non linear** patterns in data, that's just how real-world phenomena work, these might include non linear regression models where relationship between input and output is non linear or even non linear classification where the classes may not be separable by a linear boundary





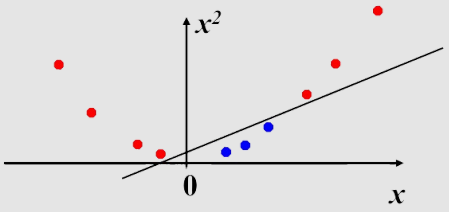
Kernel methods allows to make linear models work in non linear input spaces by mapping data to higher dimensions where it exhibits linear patterns, let's look at a quick example.

Example

This binary classification problem has no linear separation boundary (in 1D)

We can however map the input space to a feature space with two features such that

$$x \to \{x, x^2\}$$



Kernel functions

The **kernel function** is defined as the scalar product between the feature vectors of two data samples

$$k(x, x') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$$

it is symmetric, they can be interpreted as a similarity measure between $\mathbf{x}$ and $\mathbf{x}'$. Finally, a couple of further classifications of kernels can be made

- **Stationary kernels**:

$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$$

- **Homogeneous kernels**: also known as radial basis functions

$$k(\mathbf{x}, \mathbf{x}') = k(||\mathbf{x} - \mathbf{x}'||)$$

Kernel functions are used to *rework* the representation linear models to replace all the terms featuring $\phi(\mathbf{x})$ with terms that only involve $k(\mathbf{x}, \cdot)$ in other words we can sample a linear model only on the basis of the similarities between data samples which are computed using the kernel function.
This scheme is sometimes referred to as **kernel trick**. The following will be applications of the kernel trick.

Kernel ridge regression

Recall the ridge regression loss function

$$L(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \left(\mathbf{w}^T \phi(\mathbf{x}_n) - t_n \right)^2 + \frac{\lambda}{2} \mathbf{w}^T \mathbf{w} = \frac{1}{2} (\mathbf{t} - \mathbf{\Phi w})^T (\mathbf{t} - \mathbf{\Phi w}) + \frac{\lambda}{2} \mathbf{w}^T \mathbf{w}$$

We traditionally try to solve by setting the gradient of L with respect to $\mathbf{w}$ to zero

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \lambda \mathbf{w} - \mathbf{\Phi}^T (\mathbf{t} - \mathbf{\Phi w}) = 0$$

in this, instead, we perform a substitution on the weights vector

$$\mathbf{w} = \mathbf{\Phi}^T \lambda^{-1} (\mathbf{t} - \mathbf{\Phi}\mathbf{w}) = \mathbf{\Phi}^T \mathbf{a}$$

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \lambda \mathbf{w} - \mathbf{\Phi}^T (\mathbf{t} - \mathbf{\Phi}\mathbf{w}) = 0$$
$$\mathbf{\Phi}^T \left(\lambda \mathbf{a} - \left(\mathbf{t} - \mathbf{\Phi}\mathbf{\Phi}^T \mathbf{a}\right)\right) = 0$$
$$\mathbf{\Phi}\mathbf{\Phi}^T \mathrm{a} + \lambda \mathrm{a} = \mathrm{t}$$
$$\mathbf{a} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{t}$$

where $\mathrm{K} = \mathbf{\Phi}\mathbf{\Phi}^T$ is called the **Gram matrix**, it is a matrix where each element is the inner product between feature vectors (recall the definition of $k(\cdot, \cdot)$) which represents the similarities between each pair of samples in the training data

$$K = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \dots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \dots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

This **w**-sub representation is called the dual representation.
We can now sample the model using this dual representation thanks to:

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) = \mathbf{a}^T \mathbf{\Phi} \phi(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T (K + \lambda \mathbf{I}_N)^{-1} \mathbf{t}$$

where $\mathbf{w} = \mathbf{\Phi}^T \mathbf{a}$, $\mathbf{a} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{t}$ and finally $k_n(\mathbf{x}) = k(\mathbf{x}_n, \mathbf{x}) \forall \mathbf{x}_n \in D$.

The benefits of this (dual) representation can be:

- That it is computationally convenient when the number of features grows quickly since we're inverting $(\mathbf{K} + \lambda \mathbf{I})$ which is an $N \times N$ matrix, therefore grows with the number of samples rather than the features.
- It does not require us to compute ϕ, which can be difficult to do for complex data types such as text, graphs, sets etc.... It's generally much cheaper to compute the similarity between samples than ϕ.

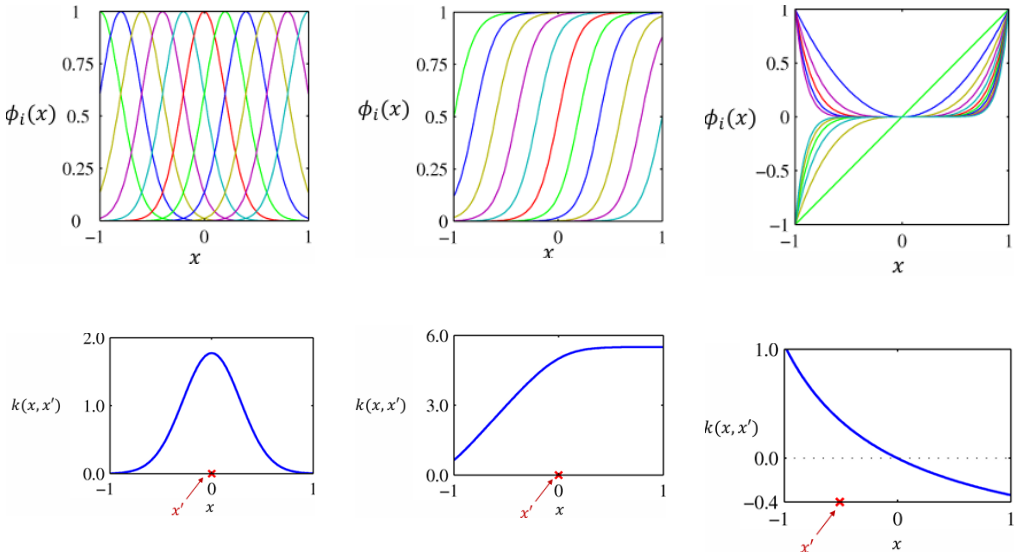
Mercer's theorem

Any continuous, symmetric, positive semi-definite kernel function $k(x, y)$ can be expressed as a dot product in a high-dimensional space.

Theorem on N-S conditions for kernel functions

One thing to consider is that for a function $k(x, x')$ to be a kernel it (necessary and sufficient) the gram matrix K must be positive semi-definite for all possible choices of the set x_n.
This means that for any non zero real vector $\mathbf{x}$ the following holds true

$$\mathbf{x}^T K \mathbf{x} > 0$$



Some examples of quadratic and trigonometric kernel functions

Kernel design

Generally speaking it is preferred to design **from existing** valid kernel functions by applying a set of rules that are guaranteed to result in a new **valid** kernel, like the following

1. $k(\mathbf{x}, \mathbf{x}') = ck_1(\mathbf{x}, \mathbf{x}')$,where $c > 0$ is a constant
2. $k(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})k_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}')$,where $f(\cdot)$ is any function
3. $k(\mathbf{x}, \mathbf{x}') = q(k_1(\mathbf{x}, \mathbf{x}'))$, where q$(\cdot)$ is a polynomial with non-negative coefficients
4. $k(\mathbf{x}, \mathbf{x}') = exp(k_1(\mathbf{x}, \mathbf{x}'))$
5. $k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}')$
6. $k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}')$
7. $k(\mathbf{x}, \mathbf{x}') = k_3(\phi(\mathbf{x}), \phi(\mathbf{x}'))$, where $\phi(\mathbf{x})$ maps $\mathbf{x}$ to $\mathbb{R}^M$ and $k_3(\cdot;\cdot)$ is a valid kernel in $\mathbb{R}^M$
8. $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{A} \mathbf{x}'$ where $\mathbf{A}$ is a symmetric semidefinite matrix
9. $k(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}'_a) + k_b(\mathbf{x}_b, \mathbf{x}'_b)$ (Where $\mathbf{x} = \{\mathbf{x}_a\} \bigcup \{\mathbf{x}_b\}$ are two subsets no necessarily
   disjoints of variables and k_a, k_b are valid kernels)
10. $k(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}'_a)k_b(\mathbf{x}_b, \mathbf{x}'_b)$ (Where $\mathbf{x} = \{\mathbf{x}_a\} \bigcup \{\mathbf{x}_b\}$ are two subsets no necessarily
    disjoints of variables and k_a, k_b are valid kernels)

Gaussian process

A **Gaussian process** is defined as a distribution probability over a function $y(x)$ such that the set of values $y(x_)$ for an arbitrary $\{x_i\}$ jointly have a gaussian distribution, meaning that, given any input the target will be a gaussian distribution and not a deterministic point.
Gaussian processes are kernel methods that can be applied to solve regression problems, they can be written as a special case of Linear regression over an infinite number of features
We will now apply this concept to rework a linear regression model.

Let's start from the same assumptions we used in Bayesian linear regression

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \phi(\mathbf{x})$$

$$p(\mathbf{w}) = \mathcal{N}\left(\mathbf{w}|0, \tau^2 \mathbf{I}\right)$$

we now compute the prior distribution of the outputs of the regression function:

$$\mathrm{y} = \Phi \mathrm{w} \Longrightarrow p(\mathbf{y}) = \mathcal{N}\left(\mathbf{y}|\boldsymbol{\mu}, \mathbf{S}\right)$$

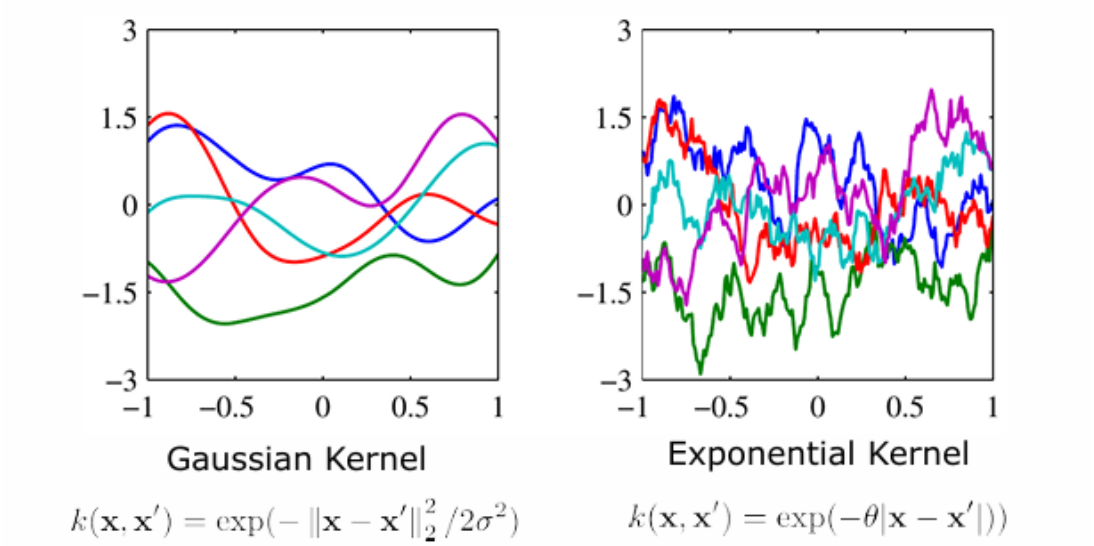
where

$$\mu = \mathbb{E}[\mathrm{y}] = \Phi \mathbb{E}[\mathrm{w}] = 0$$

$$\mathbf{S} = \mathrm{cov}[\mathbf{y}] = \mathbb{E}[\mathbf{y}\mathbf{y}^T] = \boldsymbol{\Phi}\mathbb{E}[\mathrm{w}\mathbf{w}^T]\boldsymbol{\Phi}^T = \tau^2 \boldsymbol{\Phi}\boldsymbol{\Phi}^T = \mathbf{K}$$

where $\mathbf{K}$ is the Gram matrix

$$K_{nm} = k(\mathbf{x}_n, \mathbf{x}_m) = \tau^2 \phi(\mathbf{x}_n)^T \phi(\mathbf{x}_m)$$



The two families of kernels typically used with Gaussian processes

For more on this checkout the exercise sessions.

A summary on kernel functions

Kernel functions allow us to transform the input data into a higher dimensional space without having to compute the coordinates of the data in that space explicitly. This follows from the idea that a higher dimensional space might allow for linear decision boundaries whereas lower dimensional feature spaces might not allow that.

One of the biggest advantages of kernel functions is that they allow us to compute the inner product in the high-dimensional feature space directly from the original input space, avoiding the computational cost of explicitly performing the transformation ϕ. This is known as the "kernel trick."

14. Support vector machines

The main computational burden in kernel methods lies in the need to calculate the kernel matrix (also known as the Gram matrix). For a training set with nnn samples, this matrix is n×nn \times nn×n in size, where each entry K(xi,xj)K(x_i, x_j)K(xi,xj) represents the kernel function evaluated on a pair of samples xix_ixi and xjx_jxj.