Machine Learning for High Energy Physics

Notes taken by Manuel Morales & James Moore
at the MLHEP summer school

Part I
Introduction to machine learning

1 Introduction to supervised learning

1.1 Problem setup

Definition 1.1: Consider a set $\mathcal{X}$ of objects and a set of targets $\mathcal{Y}$. Suppose also that there is an unknown function $f : \mathcal{X} \to \mathcal{Y}$ mapping objects into targets (which may be stochastic - by which we mean the objects and targets are drawn from some joint probability distribution).

A *dataset* is a finite collection $D = \{(x_i, f(x_i)) : i = 1, ..., N\} \subseteq \mathcal{X} \times \mathcal{Y}$. The goal of *supervised learning* is to *approximate f given a dataset D* (i.e. learn to recover targets from objects).

Example 1.2:

(i) **Classification of iris flower species.** The objects $\mathcal{X}$ are individual flowers, described by the length and width of their sepals and petals. The targets $\mathcal{Y}$ are the species to which the flowers belong. The mapping $f : \mathcal{X} \to \mathcal{Y}$ takes a flower to its species (and depends on different shapes of sepals, petals, etc corresponding to the different species). Note this is a *non-deterministic* mapping because the growth of a certain flower is associated with a number of random processes.

(ii) **Spam filtering.** The objects $\mathcal{X}$ are emails (sequences of characters). The targets are the categories $\mathcal{Y} = \{\text{spam}, \text{not spam}\}$. The mapping $f : \mathcal{X} \to \mathcal{Y}$ tells us whether a particular email $x \in \mathcal{X}$ is spam or not, and is again *non-deterministic* (the style and content can vary from author to author).

(iii) **CAPTCHA recognition.** The set of objects $\mathcal{X}$ is a set of CAPTCHA images (which are really vectors of pixel level brightness). The set of targets $\mathcal{Y}$ are sequences of characters, relating to the correct word that the CAPTCHA shows. The mapping $f : \mathcal{X} \to \mathcal{Y}$ is the inverse of the algorithm used to generate the CAPTCHA - this is *almost* deterministic, but depends on the level of distortion (very high distortion can make the map less deterministic).

(iv) **Particle identification in a HEP experiment.** The objects $\mathcal{X}$ are particles, described by the detector experiments (e.g. track parameters, calorimeter energy, deposit, etc.). The targets $\mathcal{Y}$ are the types of the particles (e.g. electrons, muons, protons, etc.). The mapping $f : \mathcal{X} \to \mathcal{Y}$ is the inverse of the physical process generating the detector response (by definition, this is a stochastic process).

1.2 Feature types

Typically the objects in our set $\mathcal{X}$ will have some structure.

Definition 1.3: Let $\mathcal{X}, \mathcal{Y}$ be the sets of objects and targets in a supervised learning problem. Suppose that $\mathcal{X}$ is a collection of *tuples*, so that given a dataset $D \subseteq \mathcal{X} \times \mathcal{Y}$, for each $(x_i, f(x_i)) \in D$ we can write x_i as:

$$x_i = (x_i^1, x_i^2, ..., x_i^d),$$

for some d (which need not be the same for all $x_i \in \mathcal{X}$). We say that x_i^j is a *feature* of the object x_i (note we use the upper index for the *feature*, and the lower index to label the object in the dataset).

Many algorithms require that the *dimensionality* d of the data is the same for all objects. In this case, the dataset can be represented by a *design matrix*:

$$X = \begin{pmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{pmatrix}.$$

The column index corresponds to the features, and the row index corresponds to the position of the object in the dataset.

Example 1.4: Consider the problem of classifying species of irises. In this case, features might include *sepal length*, *sepal width*, *petal length* and *petal width*. In this case, all of the features are real numbers.

Since each flower will have these features, we can describe all objects as 4-tuples, and hence we can organise the data into a $N \times 4$ design matrix.

Definition 1.5: In general, features x_i^j might be of varying natures. We give special names to some common cases:

- *Numeric features* take values in (for example) the real numbers. For example sepal length in the classification of iris species, building height, or particle transverse mometnum.
- *Categorical nominal features* take values in a finite set, with no natural ordering. For example colour, city of birth or particle type.
- *Categorical ordinal features* take values in a set equipped with a natural order, for example level of education, age, or particles passing loose, medium or tight selection criteria.
- *Binary features* take values in a set of size 2, for example $\{\text{true}, \text{false}\}$, $\{0, 1\}$, $\{-1, +1\}$.

A natural question that arises in the handling of data is how one might convert a categorical nominal feature into a binary or numeric feature (since binary or numeric features can be more easily handled).

A naïve approach would be to assign each category a number (e.g. red$= 1$, green$= 2$, etc). However, this introduces an ordering on the feature, which could have a negative effect on learning algorithms later on.

A better solution is *one-hot encoding*:

Definition 1.6: Let $x_i = (x_i^1, ..., x_i^{j-1}, x_i^j, x_i^{j+1}, ..., x_i^d)$ be an object in a dataset, and let x_i^j be a categorical nominal feature for the object, taking values in some finite set S. A *one-hot encoding* replaces the feature x_i^j with $|S|$ binary features $\{x_i^{j,s} : s \in S\}$ obeying:

$$x_i^{j,s} = \begin{cases} 0 & \text{if } x_i^j \neq s, \\ 1 & \text{if } x_i^j = s. \end{cases}$$

The new object then takes the form

$$x_i = (x_i^1, ..., x_i^{j-1}, x_i^{j,s_1}, ..., x_i^{j,s_{|S|}}, x^{j+1}, ..., x_i^d),$$

where $s_i \in S$ are the elements of S.

Example 1.7: Suppose x is a categorical nominal feature taking values in the set $\{\text{red, blue, green}\}$. A one-hot encoding of the object is:

$$(x^{\text{red}}, x^{\text{blue}}, x^{\text{green}}),$$

where $x^{\text{red}} = 0$ if $x \neq$ red, 1 if $x =$ red, etc.

For example, $x =$ red is replaced by $(1, 0, 0)$, whilst $x =$ green is replaced by $(0, 0, 1)$.

One-hot encoding can also be applied to categorial ordinal features:

Definition 1.8: Let $x_i = (x_i^1, ..., x_i^{j-1}, x_i^j, x_i^{j+1}, ..., x_i^d)$ be an object in a dataset, and let x_i^j be a categorical *ordinal* feature for the object, taking values in some finite set S with ordering $<$. Suppose that the elements of S are ordered as $s_1 < s_2 < \cdots < s_{|S|}$. A *one-hot encoding* replaces the feature x_i^j with $|S|$ binary features $\{x_i^{j,s} : s \in S\}$ obeying:

$$x_i^{j,s} = \begin{cases} 0 & \text{if } s < x_i^j, \\ 1 & \text{if } x_i^j \leq s. \end{cases}$$

The new object then takes the form

$$x_i = (x_i^1, ..., x_i^{j-1}, x_i^{j,s_1}, ..., x_i^{j,s_{|S|}}, x^{j+1}, ..., x_i^d).$$

Example 1.9: Suppose x is a categorical ordinal feature taking values in the ordered set $\{\text{bachelors, masters, PhD}\}$, with ordering bachelors $<$ masters $<$ PhD. A one-hot encoding of the object is:

$$(x^{\text{bachelors}}, x^{\text{masters}}, x^{\text{PhD}}).$$

For example, $x =$ bachelors is replaced by $(1, 0, 0)$, whilst $x =$ PhD is replaced by $(1, 1, 1)$.

Note that this *retains* the ordering of the original categorical ordinal feature.

1.3 Learning algorithms

Definition 1.10: Let $\mathcal{X}, \mathcal{Y}, f$ be the respective objects, targets and function of a supervised learning problem. Given a dataset $D = \{(x_i, f(x_i)) : i = 1, 2, ..., N\} \subseteq \mathcal{X} \times \mathcal{Y}$, a *learning algorithm* $\mathcal{A}$ returns an approximation $\hat{f} = \mathcal{A}(D)$ (which depends on the dataset) to the true function f.

Definition 1.11: The k *nearest neighbours algorithm* can be applied when the features of the objects have some notion of distance (i.e. a *metric*). We then define:

$$\hat{f}(x) = \frac{1}{k} \sum_{i : x_i \in D_x^k} f(x_i),$$

where D_x^k is the set of k objects in the dataset D closest to x.

Example 1.12: Consider a *classification problem* where the target space $\mathcal{Y}$ consists of only two values. In this case the k nearest neighbours algorithm approximates the function $f : \mathcal{X} \to \mathcal{Y}$ by:

$$\hat{f}(x) = \operatorname*{argmax}_C \sum_{i : x_i \in D_x^k} 1_{\{f(x_i) = C\}},$$

where $1_{\{f(x_i)=C\}}$ is the relevant indicator function. Note that instead of averaging the target, we take the value of C which maximises the sum, either 0 or 1.

In general, how does one find an algorithm $\mathcal{A}$ giving an approximation $\hat{f} = \mathcal{A}(D)$ to the true mapping function? Many algorithms work by solving an *optimisation task* - namely, minimising a *loss function*:

Definition 1.13: Let $\hat{f} = \mathcal{A}(D)$ be the approximation to the true mapping function for a supervised learning problem $f : \mathcal{X} \to \mathcal{Y}$, with dataset $D \subseteq \mathcal{X} \times \mathcal{Y}$ given. A *loss function* is a function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, chosen to measure the quality of predictions. The *loss* of our approximation at the point $x_i \in \mathcal{X}$ in the dataset is given by $\mathcal{L}(f(x_i), \hat{f}(x_i))$.

Example 1.14: An example of a loss function is *least squares error*, given by $\mathcal{L}(f(x_i), \hat{f}(x_i)) = (f(x_i) - \hat{f}(x_i))^2$, where the target space $\mathcal{Y} = \mathbb{R}$.

Definition 1.15: Given a loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ for a supervised learning problem, we can obtain an approximation $\hat{f} = \mathcal{A}(D)$ of the true mapping $f : \mathcal{X} \to \mathcal{Y}$ from a datset D via *loss minimisation*:

$$\hat{f} = \operatorname*{argmin}_{\tilde{f}} \mathop{\mathbb{E}}_{(x_i, f(x_i)) \in D} \mathcal{L}(f(x_i), \tilde{f}(x_i)),$$

where the argmin is taken over all functions $\tilde{f} : \mathcal{X} \to \mathcal{Y}$, and the notation $\mathbb{E}$ means the expected value of the loss over the dataset (in the case that the mapping is stochastic, this is important).

Example 1.16: An example is *linear regression*, where we take $\mathcal{X} = \mathbb{R}^d$, $\mathcal{Y} = \mathbb{R}$. In this case, the functions $\tilde{f}$ which could constitute possible mappings $\mathcal{X} \to \mathcal{Y}$ are restricted to simply be *linear functions*, of the form:

$$\tilde{f}_{\mathbf{w},\mathbf{b}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + \mathbf{b}.$$

Given a dataset $D \subseteq \mathcal{X} \times \mathcal{Y}$ of size N, minimising the least squared loss corresponds to determining $\mathbf{w}$, $\mathbf{b}$ such that:

$$\frac{1}{N} \sum_{i=1,\ldots,N} \left(f(x_i) - \hat{f}_{\mathbf{w},\mathbf{b}}(x_i) \right)^2$$

is minimised.

1.4 Assumptions about data and model complexity

Given no assumptions about a dataset, there are typically infinitely many solutions to a loss minimisation problem. For example, if we wish to interpolate a finite set of points for a one-dimensional function, infinitely many curves will perfectly minimise the loss (defined in this case as the sum of the distances between the curve and the point at each point).

To combat these, we introduce the concept of *test data*:

Definition 1.17: A set of *test data* is an independent dataset drawn from the same population as the dataset used to construct the solution of our loss minimisation problem (this original set is called the *training data*).

We then demand that our expected loss on the whole population (i.e. both the test and training data) is minimised.

Example 1.18: Consider a linear interpolation problem in $\mathbb{R}^d$ which we solve both with the k nearest neighbour algorithm, and with linear regression. Suppose the true dependence is also linear.

It turns out that if we validate our approaches using a test set, the error on the test set for k nearest neighbour and linear regression agree well for lower dimensions d, but k nearest neighbour becomes much worse for larger numbers of dimensions.

This feature is called the *curse of dimensionality* - as the number of dimensions of the feature space grows, the data becomes very sparse, and the k nearest neighbour approach begins to fail. In this case, to keep the error low, the number of training points would have to increase exponentially with the dimension.

The reason we get the distinction is that the two algorithms have different assumptions:

- The k nearest neighbour (kNN) approach assumes that *similar objects have similar targets*.
- The linear regression approach assumes that *targets are linear in features*.

In this case, both assumptions are correct, but linearity is much stronger which allows us to fight the curse of dimensionality.

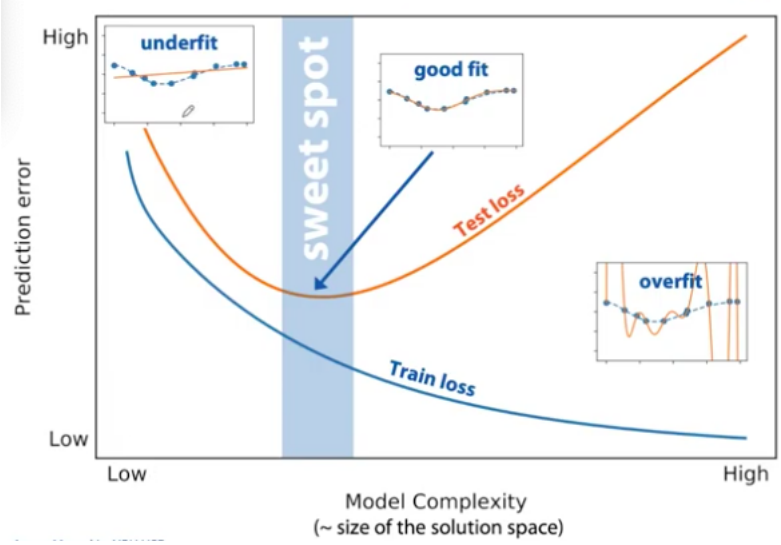
In particular, imposing assumptions about the data *restricts the possible function space of solutions* $\tilde{f}$ for the loss minimisation problem:

$$\hat{f} = \operatorname*{argmin}_{\tilde{f}} \mathop{\mathbb{E}}_{(x_i, f(x_i)) \in D} \mathcal{L}(f(x_i), \tilde{f}(x_i)),$$

Linearity is a much stronger constraint than kNN imposes (kNN allows for a much more flexible functional form), restricting the space of functions much more. Sometimes these restrictions help us to *overcome the curse of dimensionality*.

Definition 1.19: The *complexity* of a model refers to the heuristic size of the function space over which we minimise in the loss minimisation problem. A larger complexity means a larger function space, whilst a smaller complexity means a smaller function space.

In general, changing the model complexity can allow us to find the 'sweet spot' where expected loss on both the training and test data sets is minimised:



The goal is to find the *right level of limitations* - not too strict, not too loose.

2 Linear models

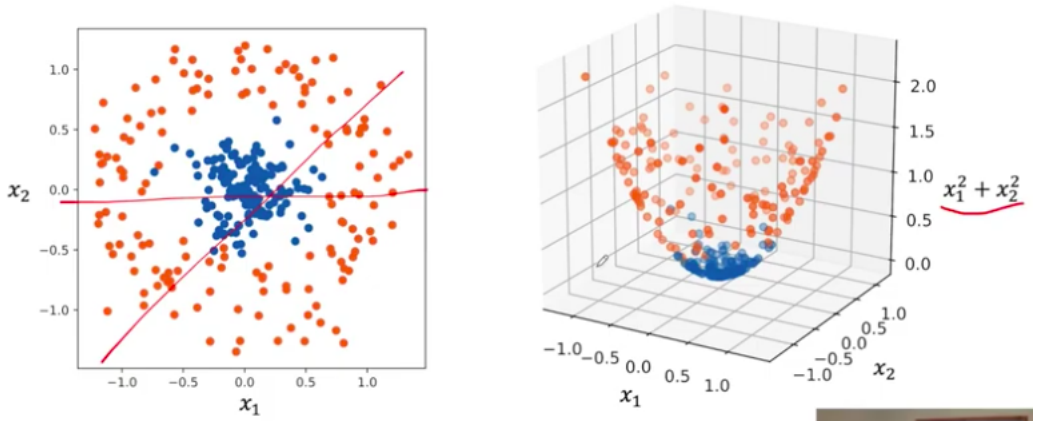
2.1 Linear models and linear regression

Definition 2.1: A *linear model* is a model $\hat{f}(x)$ which is linear in its input (note this assumes a vector space structure on both the objects and targets).

Example 2.2: Examples include:

- Linear regression, for example $\hat{f}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$.
- Classification, for example $\hat{f}(\mathbf{x}) = 1_{\{\boldsymbol{\theta}^T \mathbf{x} > 0\}}$. This is non-linear as a function, but divides the target space by a line.

Linear models have a *hidden power*. Problems which appear linearly inseparable (i.e. cannot be solved with linear models) can be made into problems which are tractable with linear models by *transforming the features*. As an example, the data on the left below can be made linearly separable by introducing a new feature based on the sum of the squares of the coordinates of the points:



Now a hyperplane separates the blue and orange sets.

In general, linear models motivate the construction of *deep models*. Neural networks are just linear models with *activation functions* between them (effecting some transformation). Thus better understanding linear models will help us understand deep neural networks.

We have already introduced linear regression as a linear model. Here, we update the notation:

Definition 2.3: The *model prediction* is $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$, where $\boldsymbol{\theta} \in \mathbb{R}^d$ is our *parameter vector*, $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^d$ is our *features vector*, and:

$$\frac{1}{N} \sum_{i=1,\ldots,N} \left(f(\mathbf{x}_i) - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i) \right)^2$$

is our mean squared error loss, given a dataset. We wish to minimise this over $\boldsymbol{\theta}$.

Other loss functions are available. For example:

- **Mean absolute error (MAE).** This is given by:

$$\frac{1}{N}\sum_{i=1,\ldots,N}\left|f(\mathbf{x}_i)-\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right|.$$

  This grows linearly for large errors, rather than quadratically like mean squared error loss. In particular, it doesn't penalise large errors as much as mean squared error loss - it might be more suitable if there are outliers in your data which you do not wish to be biased against.

- **Mean absolute percentage error (MAPE).** This is given by:

$$\frac{1}{N}\sum_{i=1,..,N}\left|\frac{f(\mathbf{x}_i)-\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)}{f(\mathbf{x}_i)}\right|.$$

- **Mean squared logarithmic error (MSLE).** This is given by:

$$\frac{1}{N}\sum_{i=1,\ldots,N}\Big(\log(f(\mathbf{x}_i)+1)-\log(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)+1)\Big)^2.$$

  The 1's are there just to allow zero values of the targets.

Both MAPE and MSLE provide a means of computing *relative error*, so are useful when the targets span through a large range of magnitudes.

In general, applying different loss functions are also related to different assumptions about the data. A different loss function might be more suitable to a particular dataset.

2.2 Analytic solution

Recall that earlier we defined the design matrix X, a matrix with rows given by the objects and columns given by the features. We can use this to rewrite the mean squared error loss as:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1,\dots,N} \left(f(\mathbf{x}_i) - \boldsymbol{\theta}^T \mathbf{x}_i \right)^2 = \frac{1}{N} ||\mathbf{y} - X\boldsymbol{\theta}||^2,$$

where $\mathbf{y} = (f(\mathbf{x}_1), ..., f(\mathbf{x}_N))^T$.

To minimise this function, we can ignore the $1/N$ as it plays no role. Thus we wish to minimise $||\mathbf{y} - X\boldsymbol{\theta}||^2$ over $\boldsymbol{\theta}$. Analytically, the requirements are:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \mathcal{L}_{\text{MSE}} = \mathbf{0}, \qquad \text{the matrix } \frac{\partial^2}{\partial \theta_i \partial \theta_j} \mathcal{L}_{\text{MSE}} \text{ of second derivatives should be positive definite.}$$

The first condition ensures we have an extremum, the second condition ensures that extremum is a minimum.

Proposition 2.4: If the columns of X are linearly independent, the solution to the above minimisation problem is $\boldsymbol{\theta} = (X^T X)^{-1} \mathbf{y}$.

Proof: Note that in general, the matrix $X^T X$ is always *positive semi-definite*, since:

$$\mathbf{v}^T X^T X \mathbf{v} = ||X\mathbf{v}||^2 \geq 0$$

for arbitrary $\mathbf{v}$. We see it is *precisely* positive definite when the columns of X are linearly independent (imagine the expression $X\mathbf{v}$ to be a linear combination of the columns of X). This additionally implies X is invertible (e.g. all its eigenvalues are positive).

We can now consider the optimisation problem. The first derivative of $\mathcal{L}_{\text{MSE}}$ is easily computed:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \mathcal{L}_{\text{MSE}} = \frac{1}{N} \frac{\partial}{\partial \boldsymbol{\theta}} (\mathbf{y} - X\boldsymbol{\theta})^T (\mathbf{y} - X\boldsymbol{\theta}) = -\frac{2}{N} X^T (\mathbf{y} - X\boldsymbol{\theta}) = \mathbf{0}.$$

This implies that $X^T \mathbf{y} - X^T X \boldsymbol{\theta} = \mathbf{0}$. Since $X^T X$ is invertible, we can compute the solution:

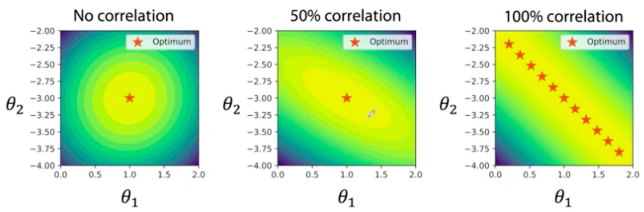
$$\boldsymbol{\theta} = (X^T X)^{-1} X^T \mathbf{y}.$$

To check that this is a minimum, we must look at the second derivative. We find that the matrix of second derivatives is precisely:

$$2X^T X.$$

In particular, this matrix is positive definite by the above. □

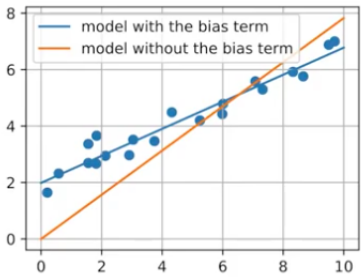
In practice, there may be some linear dependence of the columns of X (i.e. there may be some 'feature correlations'). This affects the invertibility of the matrix $X^T X$ - we get *flat directions*, and hence multiple solutions to the problem.



In the above we also assumed that our model took the form $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$ for ease of calculation, but actually we can consider the more general linear model $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x} + \theta_0$, where the term θ_0 is called the *bias term*. There is no need to redo any calculations here; we simply add a constant feature to the design matrix to account for this:

$$X = \begin{pmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{pmatrix} \quad \rightarrow \quad X = \begin{pmatrix} 1 & x_1^1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N^1 & x_N^2 & \cdots & x_N^d \end{pmatrix}$$

Now when we multiply $X\boldsymbol{\theta}$, one of the parameters will play the role of the bias term.



2.3 Numerical and stochastic optimisation: gradient descent

We have now discussed the analytic solution for the mean squared error loss minimisation problem in the context of a linear model. However, most loss functions do not allow an analytic solution, so it is important to discuss numerical solutions.

We being with some revision of the *gradient*. Recall that the gradient of a scalar function on $\mathbb{R}^d$ is given by:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), ..., \frac{\partial f}{\partial x_d}(\mathbf{x}) \right).$$

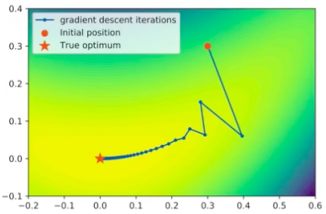
The gradient always points in the *direction of greatest increase* of the function. Hence if we start at a point, and move in the opposite direction to the gradient, we will get closer to a local minimum - this aids optimisation.

Let us formalise this:

Definition 2.5: Given a smooth function $f : \mathbb{R}^d \to \mathbb{R}$ and an initial point $\mathbf{x}^{(0)}$, we define a recursive sequence of points via:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - \alpha \nabla f(\mathbf{x}^{(k-1)}),$$

where $\alpha \in \mathbb{R}_{>0}$ is a constant called the *learning rate*. This construction is called *gradient descent iteration*.



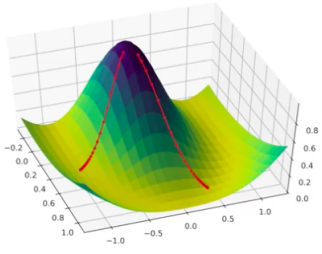
It is possible to show that:

Proposition 2.6: For smooth, *convex* functions with a single minimum $\mathbf{x}^* \in \mathbb{R}^d$, we have:

$$f(\mathbf{x}^{(k)}) - f(\mathbf{x}^*) = O\left(\frac{1}{k}\right),$$

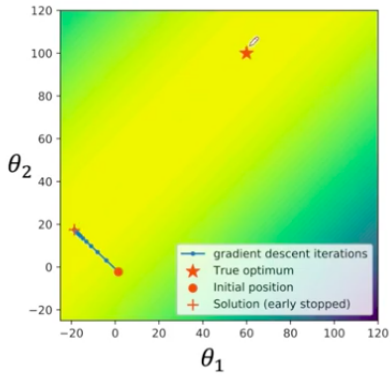
i.e. as we take more and more points, we approach the true minimum of the function.

Gradient descent may also be applied to non-convex functions, but we may reach a minimum which is not *global*, and the result may *depend on the starting point*.



A useful property of gradient descent is that it can be used for *regularisation* when our loss function is ill-defined; for example, when our $X^T X$ matrix above is not invertible (or not stably invertible).

In such cases, the true minimum typically has very large parameter values. These often correspond to *overfitting*. To avoid this, we can start from a point where our initial parameters are *small* and we can stop the gradient descent early (this is called *early stopping*). This will typically be a better solution than the true minimum.



There is an important modification to the gradient descent algorithm called *stochastic gradient descent*. Recall that in machine learning we typically optimise loss functions which are averages over some set of objects:

$$L = \frac{1}{N} \sum_{i=1,\ldots,N} \mathcal{L}\left(f(x_i), \hat{f}_\theta(x_i)\right).$$

For large N, gradient descent is computationally inefficient and may be unfeasible in terms of memory consumption. Stochastic gradient descent deals with this problem:

Definition 2.7: Given a function $\hat{f}_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}$ which depends on a parameter $\boldsymbol{\theta} \in \mathbb{R}^d$, a starting point $\boldsymbol{\theta}^{(0)}$, and a set of objects $x_1, \ldots, x_N \in \mathcal{X}$, we define a recursive sequence of points as follows:

- At step k of the process, pick some $l_k \in \{1, \ldots, N\}$ at random.
- Define:

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}\left(f(x_{l_k}), \hat{f}_{\boldsymbol{\theta}^{(k-1)}}(x_{l_k})\right).$$

This is called *stochastic gradient descent iteration*.

In the case of a smooth convex function, stochastic gradient descent also leads you to the unique minimum, but at a slower rate of $O(1/\sqrt{k})$ instead. This *can* be improved by 'batching' and other tricks.

2.4 Feature expansion

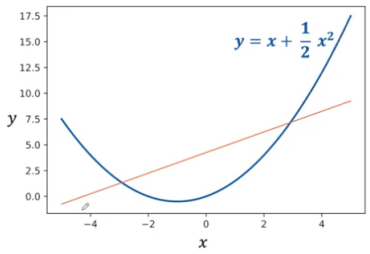
Let us now see how linear models can be made more powerful by transforming their features.

Definition 2.8: Given a design matrix X of dimension $N \times d$, a *feature transformation* via the function $\Phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ is the induced map of the design matrix:

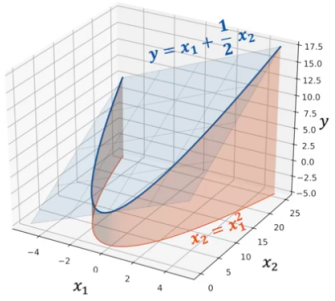
$$X = \begin{pmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{pmatrix} \quad \to \quad \Phi(X) = \begin{pmatrix} \Phi^1(x_1^1, ..., x_1^d) & \cdots & \Phi^{d'}(x_1^1, ..., x_1^d) \\ \Phi^1(x_2^1, ..., x_2^d) & \cdots & \Phi^{d'}(x_2^1, ..., x_2^d) \\ \vdots & \ddots & \vdots \\ \Phi^1(x_N^1, ..., x_N^d) & \cdots & \Phi^{d'}(x_N^1, ..., x_N^d) \end{pmatrix}$$

Finding good functions Φ to transform with is called *feature engineering*. It is an important part of machine learning and requires a deep understanding of the underlying problem and the data.

Example 2.9: (**Polynomial features.**) Consider trying to fit the curve $y = x + \frac{1}{2}x^2$. This cannot be solved using only linear regression; a single line will not work.



However, we can introduce a new feature via a feature transformation. Let $(x_1, x_2) = (x, x^2)$. Then in the new feature space, a linear estimate is really an estimate of the form $\hat{f}(\mathbf{x}) = \theta_1 x_1 + \theta_2 x_2 = \theta_1 x + \theta_2 x^2$, which facilitates fitting.



Example 2.10: (**Polynomial features - general case**) Let $(x_i^1, x_i^2, ..., x_i^d)$ be the original features of a dataset. We introduce as new features all unique multiplicative combinations of the form:

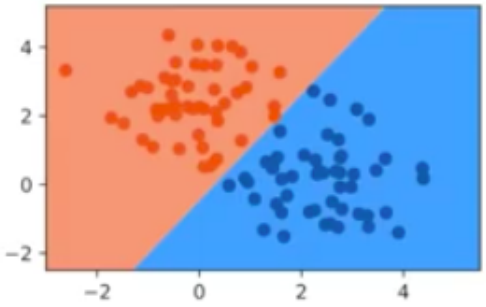
$$(x_i^1)^{p_1}(x_i^2)^{p_2}...(x_i^d)^{p_d},$$

where $p_1 + p_2 + \cdots + p_m \leq p$ (with p the degree we expect to work).

3 Classification with linear models

3.1 Classification with linear regression

Consider a classification problem, with two classes, as shown below:

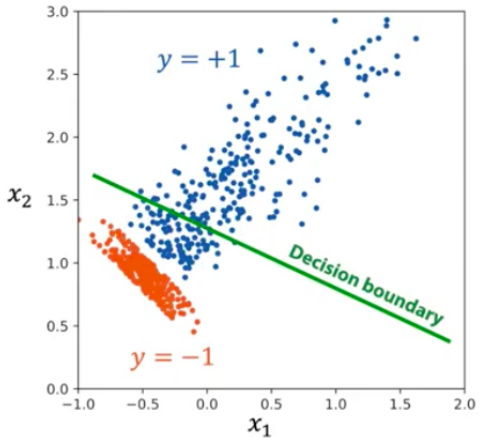


Since there are only two classes, we can assign numeric values to the classes - for our convenience, we choose $y = +1$ for one class (the 'positive' class) and $y = -1$ for the other class (the 'negative' class)

We can try to predict these targets with a linear model. To do so, we solve a linear regression problem using the model $\hat{y}_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$. To obtain the classified function, we then simply take the sign of the result:

$$\hat{f}(\mathbf{x}) = \text{sign}(\hat{y}(\mathbf{x})).$$

This may seem like a reasonable approach, but you can face problems. Consider the following set:



Here, the MSE loss makes the model avoid high errors, at the price of *pushing the decision boundary* towards the class with higher spread.

Therefore, we might try to use a different loss function. A good first example is 0-1 *loss*.

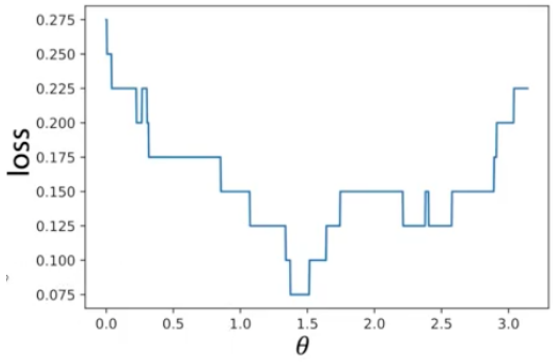
Definition 3.1: For the above classification problem, we define the 0-1 *loss by*:

$$\mathcal{L}_{0\text{-}1} = \frac{1}{N} \sum_{i=1,\ldots,N} 1_{\{\boldsymbol{\theta}^T \mathbf{x}_i y_i < 0\}}$$

where $y_i \in \{-1, +1\}$. We call $M = \boldsymbol{\theta}^T \mathbf{x} y$ the *margin*. When $M > 0$, we have a *correct* classification, but when $M < 0$ we have an *incorrect* classification. The higher the magnitude of M, the further an object is from the decision boundary.

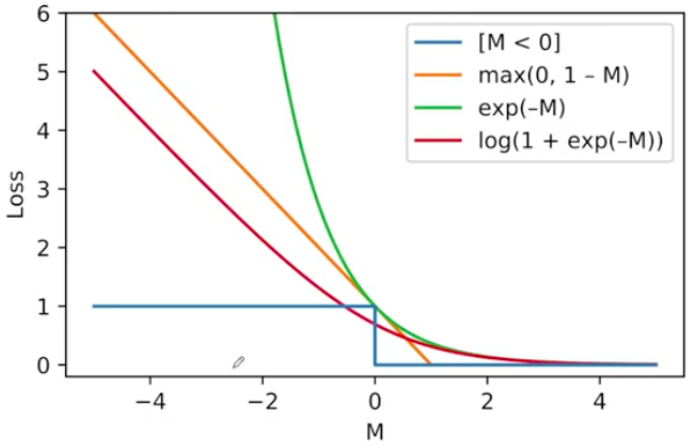
The sum computes the number of times our linear regression misclassifies a point.

Unfortunately, the resulting 0-1 loss function is a piecewise constant function of $\boldsymbol{\theta}$:



This means that we cannot use *gradient descent* to minimise our loss (of course, there are other methods - we will discuss them later).

Instead of using 0-1 loss then, we could replace the sharp 'step' function described by the loss by a *differentiable upper bound*:



Examples are shown in the above figure (the red line can be scaled to be above the step function). We will discuss the red line (*log loss*) in much more detail as we discuss *logistic regression*.

3.2 Logistic regression

Let's depart from the discussion above, and approach things from a different way to start with. Consider modelling the *class probabilities*: we introduce a function $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x})$ that gives the probability of an object $\mathbf{x}$ being in the class $y = +1$, so that:

$$\mathbb{P}(y = +1|\mathbf{x}) = \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}), \qquad \mathbb{P}(y = -1|\mathbf{x}) = 1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}).$$

How can we optimise such a model? We can fit with the *maximum log likelihood*.

Recall that the *likelihood* is defined by:

$$\text{likelihood} = \prod_{i=1,\dots,N} \mathbb{P}(y_i|\mathbf{x}_i) = \prod_{i=1,\dots,N} \left(1_{\{y_i=+1\}} \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i) + 1_{\{y_i=-1\}} \left(1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)\right).$$

Maximising the logarithm likelihood allows us to obtain the best values of the parameter $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} = \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1,\dots,N} \left(1_{\{y_i=+1\}} \log\left(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + 1_{\{y_i=-1\}} \log\left(1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)\right).$$

Note that the indicator functions can be moved out of the logarithm because only one of them is equal to 1 at a time.

Once we have fitted the parameter values, we can for example predict the class the *highest probability* for each $\mathbf{x} \in \mathcal{X}$ in the object space. More generally, we could not just choose the highest probability class but try to use a probability threshold that is suitable for our problem.

Let's try to relate this back to linear models now. We need to ask: how can we map a linear model output to a probability value in $[0, 1]$? A common choice is the *sigmoid function*:

Definition 3.2: The *sigmoid function* is defined by:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

It has the useful property that $1 - \sigma(x) = \sigma(-x)$.

This can be applied to the above example, since we can define:

$$\mathbb{P}(y = +1|\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \mathbf{x}).$$

where $\boldsymbol{\theta}^T \mathbf{x}$ is our linear model. The linear model $\boldsymbol{\theta}^T \mathbf{x}$ then has the meaning of *log odds ratio* between the two classes:

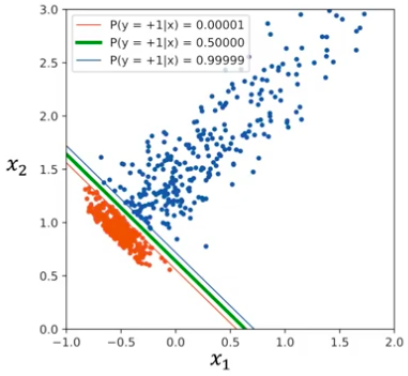
$$\log\left(\frac{\mathbb{P}(y = +1|\mathbf{x})}{\mathbb{P}(y = -1|\mathbf{x})}\right) = \log\left(\frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}} \cdot \frac{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}{e^{-\boldsymbol{\theta}^T \mathbf{x}}}\right) = \boldsymbol{\theta}^T \mathbf{x}.$$

Therefore to bring everything together, we use the negative log likelihood as our loss function:

$$\begin{aligned}
\mathcal{L} &= -\sum_{i=1,\dots,N} \left(1_{\{y_i=+1\}} \log\left(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + 1_{\{y_i=-1\}} \log\left(1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)\right) \\
&= \sum_{i=1,\dots,N} \left(1_{\{y_i=+1\}} \log\left(\sigma(\boldsymbol{\theta}^T \mathbf{x}_i)\right) + 1_{\{y_i=-1\}} \log\left(\sigma(-\boldsymbol{\theta}^T \mathbf{x}_i)\right)\right) \\
&= -\sum_{i=1,\dots N} \log\left(\sigma(\boldsymbol{\theta}^T \mathbf{x}_i y_i)\right) = \sum_{i=1,\dots N} \log\left(1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_i y_i}\right).
\end{aligned}$$

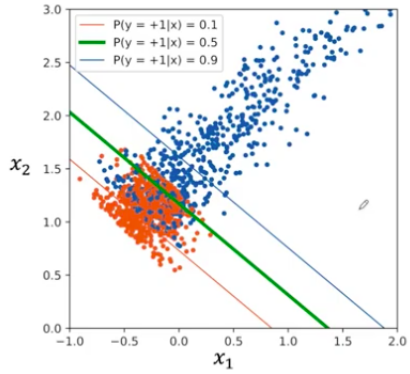
This is the same as the 'red' example we used above when we discussed upper bounds on 0-1 loss. We can apply gradient descent or stochastic gradient descent here.

Applying to the dataset above, the decision boundary is in the correct place with logistic regression:



Note that when the classes are linearly separable as above, for any correct decision boundary, mapping $\boldsymbol{\theta} \rightarrow C \cdot \boldsymbol{\theta}$ for some $C > 1$ keeps the boundary at the same place, yet improves the loss. The ideal fit is when the sigmoid turns into a step function (at infinitely large $\boldsymbol{\theta}$.

If the classes overlap however, the predicted class probability changes smooth, and the loss has a finite minimum.



3.3 Multinomial logistic regression

Similarly to the binary case, we can model the class probabilities when there are more classes than 2. Let's model the *unnormalised* class probabilities as:

$$\tilde{\mathbb{P}}(y = k|\mathbf{x}) = \exp\left(\boldsymbol{\theta}_k^T \mathbf{x}\right),$$

where we now have K parameter vectors, $k = 1, ..., K$. The normalised probabilities can be straightforwardly obtained:

$$\mathbb{P}(y = k|\mathbf{x}) = \frac{\exp\left(\boldsymbol{\theta}_k^T \mathbf{x}\right)}{\sum_{k'=1,...,K} \exp(\boldsymbol{\theta}_{k'}^T \mathbf{x})}$$

This function is called *softmax* and is commonly used in neural networks.

Note that there is a transformational symmetry here: transforming $\boldsymbol{\theta}_k \mapsto \boldsymbol{\theta}_k + \mathbf{v}$ by some constant vector $\mathbf{v}$ does not affect the normalised probability. We have:

$$\tilde{\mathbb{P}}(y = k|\mathbf{x}) = e^{\boldsymbol{\theta}_k^T \mathbf{x}} \to e^{\mathbf{v}^T \mathbf{x}} \cdot e^{\boldsymbol{\theta}_k^T \mathbf{x}} = e^{\mathbf{v}^T \mathbf{x}} \cdot \tilde{\mathbb{P}}(y = k|\mathbf{x}).$$

The remaining exponential cancels out in the ratio in the normalised probabilities. Therefore, we have some extra degree of freedom, which we are free to choose - we can therefore set $\boldsymbol{\theta}_K = \mathbf{0}$ without loss of generality for example (i.e. we set the last parameter to zero). We are left with $K - 1$ parameter vectors.

Individual linear outputs $\boldsymbol{\theta}_k^T \mathbf{x}$ now have the meaning of *log odds ratio* between the classes k and K:

$$\log\left(\frac{\mathbb{P}(y = k|\mathbf{x})}{\mathbb{P}(y = K|\mathbf{x})}\right) = \log\left(\frac{\tilde{\mathbb{P}}(y = k|\mathbf{x})}{\tilde{\mathbb{P}}(y = K|\mathbf{x})}\right) = \log\left(\frac{e^{\boldsymbol{\theta}_k^T \mathbf{x}}}{e^0}\right) = \boldsymbol{\theta}_k^T \mathbf{x}.$$

Putting everything into the negative log likelihood function, we again get our loss function:

$$\mathcal{L} = -\sum_{i=1,...,N} \log\left(\frac{\exp(\boldsymbol{\theta}_{y_i}^T \mathbf{x}_i)}{1 + \sum_{k'=1,...,K-1} \exp\left(\boldsymbol{\theta}_{k'}^T \mathbf{x}_i\right)}\right),$$

where $\boldsymbol{\theta}_K = \mathbf{0}$. Again, this can be optimised *numerically* using gradient descent or stochastic gradient descent.

3.4 Multiclass classification: general approach

In fact, you are not limited to use logistic regression for multiclass classification - you are able to use any linear model.

For a problem with K classes, introduce K predictors:

$$\hat{f}_k(\mathbf{x}) : \mathcal{X} \to \mathbb{R}$$

for $k = 1, ..., K$ each of which outputs a corresponding *class score*. We can use this to predict the class with the *highest score*:

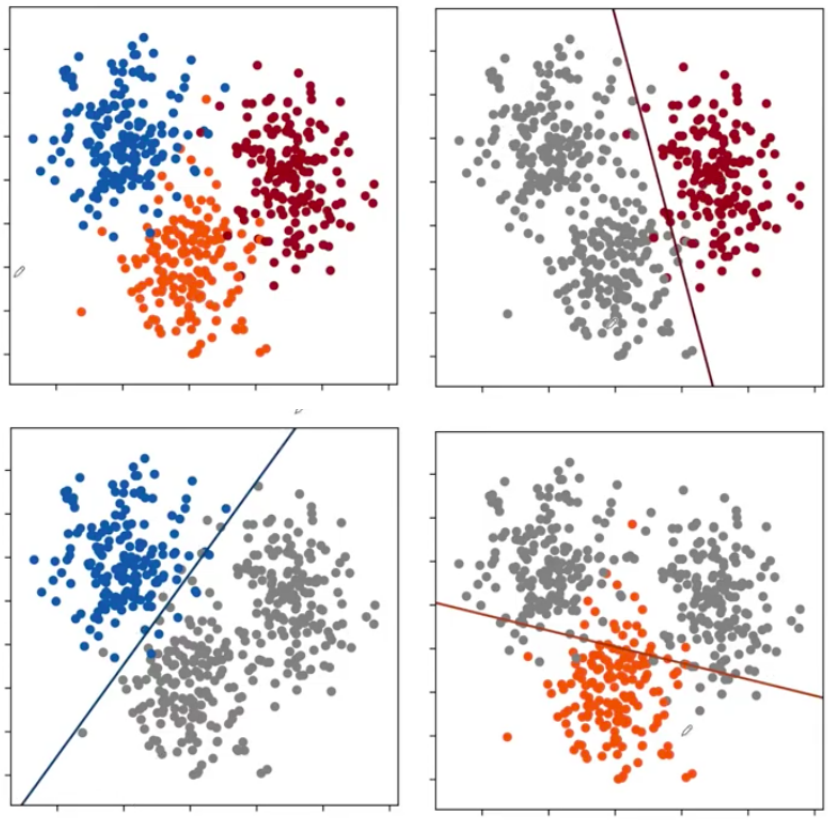
$$\hat{y}_i = \underset{k}{\operatorname{argmax}} \hat{f}_k(\mathbf{x}_i).$$

For example, any binary linear classification model can be converted to a multiclass classification with the *one-versus-rest* strategy:

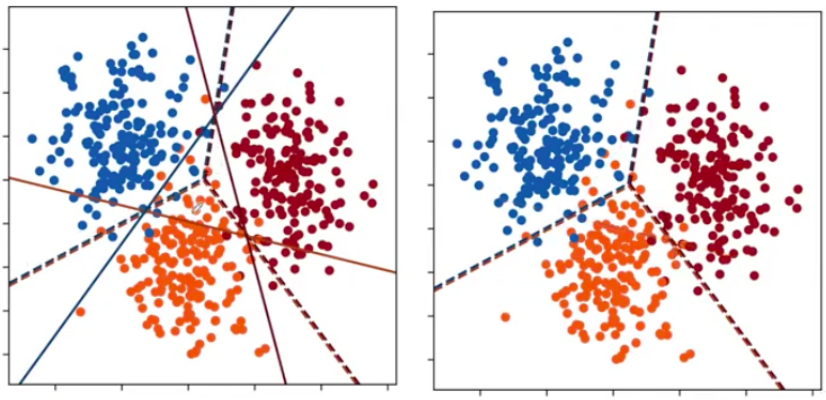
- For each class k train a binary model $\hat{f}_k(\mathbf{x}) = \boldsymbol{\theta}_{(k)}^T \mathbf{x}$ separating the given class from all others, $\hat{y}_{(k)}^{\text{one-vs-rest}} = \operatorname{sign}(\hat{f}_k(\mathbf{x}))$.
- Use the outputs of $\hat{f}_k$ as class scores for multiclass classification:

$$\hat{y}_i = \underset{k}{\operatorname{argmax}} \hat{f}_k(\mathbf{x}_i).$$

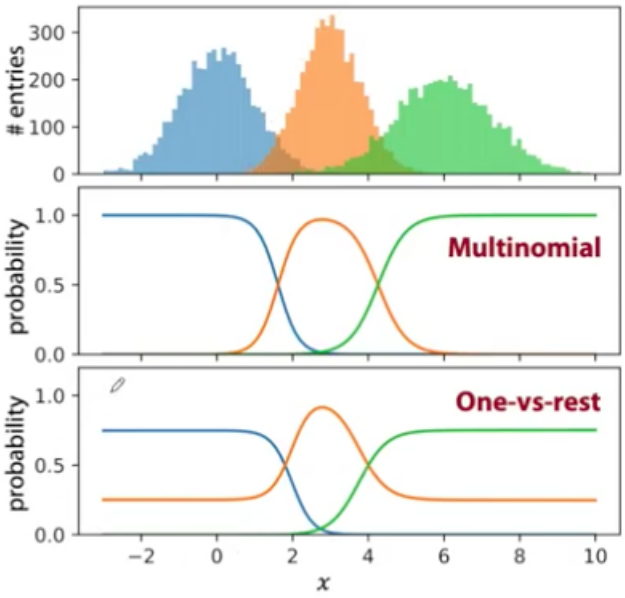
Example 3.3: In the following three-class classification, the one-versus-rest splits given by using the appropriate binary classifiers are shown below:



The resulting decision boundaries can be built up as follows:



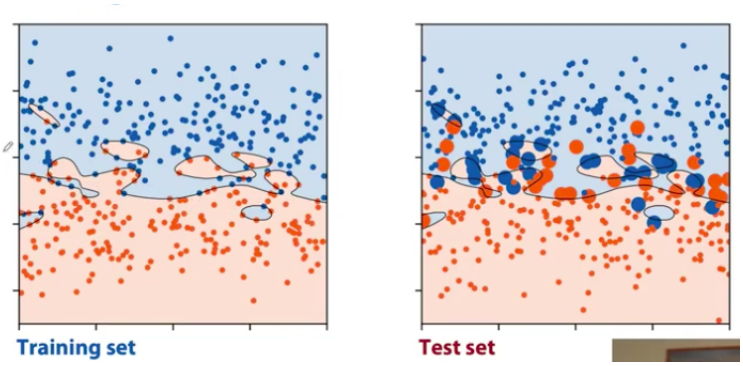
Is the one-vs-rest approach better than multinomial logistic regression however? Some problems are not linearly separable, so one-vs-rest results in biased class probabilities whilst logistic regression is still quite accurate:



4 Model regularisation

4.1 The problem of overfitting

Overfitting is the tendency of a model to adjust to random fluctuations in the data. This can be detected by a model making poor predictions on the *test set*.



Recall that we need to use a model complexity such that we don't underfit, and we don't overfit - we need to find the 'sweet spot' in between.

4.2 Prediction error decomposition

Before talking about regularisation, let's discuss the prediction error of a model. Assume that there's the following unknown relation between features and targets:

$$y = f(x) + \epsilon,$$

where ϵ is some random noise, $\mathbb{E}[\epsilon] = 0$ and $\mathrm{Var}[\epsilon] = \sigma_\epsilon^2$. Let's denote a training set as τ. We want to study the *expected squared error* for the model $\hat{f}_\tau$ trained on the dataset τ:

$$\mathrm{exp.sq.err}(x) = \mathop{\mathbb{E}}_{\tau,y}\left[(\hat{f}_\tau(x) - y)^2 | x\right].$$

The target y is sampled independently of the training set.

We can rewrite this expression in the following form, by adding and subtracting the 'prediction of the expected model' and the 'ground truth' $f(x)$ (without the noise):

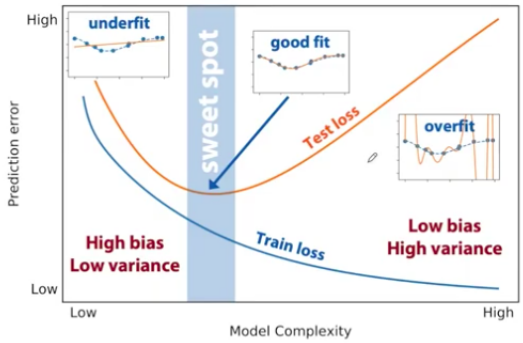
$$\begin{aligned}
\mathrm{exp.sq.err}(x) &= \mathop{\mathbb{E}}_{\tau,y}\left[(\hat{f}_\tau(x) - y)^2 | x\right] \\
&= \mathop{\mathbb{E}}_{\tau,y}\left[\left((\hat{f}_\tau(x) - \mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(x)]) + (\mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(x)] - f(x)) + (f(x) - y)\right)^2 \middle| x\right] \\
&= \mathop{\mathbb{E}}_{\tau}\left[\left(\hat{f}_\tau(x) - \mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(x)]\right)^2\right] + \left(\mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(x)] - f(x)\right)^2 + \mathop{\mathbb{E}}_{y}[(f(x) - y)^2 | x].
\end{aligned}$$

In the final step, we have expanded everything and used the fact that the cross-term expectation are 0 (this follows from the fact that τ, y are sampled independently).

The remaining terms have important interpretations:

- The first term is the *variance of the model*. It quantifies how 'unstable' the model is with respect to the noise in the training data.
- The second term is the *squared bias*. It is how much the 'expected model' differs from the ground truth.
- The third term is the *irreducible error*. Substituting the true dependence y, we find it is just given by σ_ϵ^2. This error cannot be eliminated.

The first two errors on the other hand, can be removed by changing the model. Typically there's a *tradeoff* between the two types of errors:



For low model complexity, the *bias* is high because we are artificially simplifying the model - it doesn't depend much on the data (it has 'low variance'). The opposite is true for high model complexity. This tradeoff is called the *bias-variance tradeoff*.

Example 4.1: Let's compute the bias and variance of a linear model. For each expectation $\mathbb{E}$ with respect to τ, we will assume that the features are *fixed*, i.e. $X_\tau = X$ (i.e. the design matrix is constant), and only the target vector y_τ is random. This simplification allows us to analytically compute the result.

Recall the solution for the linear regression model with the MSE loss:

$$\hat{f}_\tau(\mathbf{x}) = \boldsymbol{\theta}_\tau^T \mathbf{x}, \qquad \boldsymbol{\theta}_\tau = (X^T X)^{-1} X^T \boldsymbol{y}_\tau.$$

The *bias term* from the above error decomposition is:

$$\text{bias}(\mathbf{x}) = \mathop{\mathbb{E}}_{\tau}[\hat{f}_\tau(\mathbf{x})] - f(\mathbf{x}) = \mathop{\mathbb{E}}_{\tau}\left[\mathbf{x}^T (X^T X)^{-1} X^T \mathbf{y}_\tau\right] - \mathbf{x}^T \boldsymbol{\theta}_{\text{true}}$$

We also assume that the true dependence is indeed linear, i.e. $f(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}_{\text{true}}$ for some $\boldsymbol{\theta}_{\text{true}}$.

Since X does not depend on the sampling dataset τ, we can move lots of things out of the expectation, leaving:

$$\mathbf{x}^T (X^T X)^{-1} X^T \mathop{\mathbb{E}}_{\tau}[\mathbf{y}_\tau - \mathbf{x}^T \boldsymbol{\theta}_{\text{true}} = \mathbf{x}^T (X^T X)^{-1} X^T X \boldsymbol{\theta}_{\text{true}} - \mathbf{x}^T \boldsymbol{\theta}_{\text{true}} = \mathbf{x}^T \boldsymbol{\theta}_{\text{true}} - \mathbf{x}^T \boldsymbol{\theta}_{\text{true}} = 0.$$

Hence we see that the linear regression model is *unbiased*, provided the true dependence is linear.

Now let's consider the *variance term*:

$$\text{variance}(\mathbf{x}) = \mathop{\mathbb{E}}_{\tau}\left[\left(\hat{f}_{\tau}(\mathbf{x}) - \mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(\mathbf{x})]\right)^2\right].$$

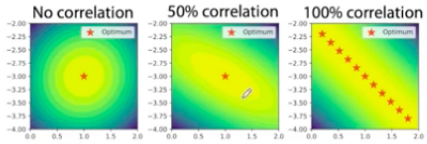
It can be shown that:

$$\text{variance}(\mathbf{x}) = \sigma_{\epsilon}^2 \mathbf{x}^T (X^T X)^{-1} \mathbf{x},$$

so that the variance error component is a *quadratic form*, defined by the $(X^T X)^{-1}$ matrix. We can diagonalise $X^T X$ giving:

$$\text{variance}(\mathbf{x}) = \sigma_{\epsilon}^2 \tilde{\mathbf{x}}^T \Lambda^{-1} \tilde{\mathbf{x}},$$

where $\Lambda = \text{diag}\{\lambda_1, ..., \lambda_d\}$ is the diagonal matrix of eigenvalues of $X^T X$. This means that *small eigenvalues amplify the model variance*. This happens when $X^T X$ is ill-defined, e.g. when the features are correlated.



MSE loss values as a function of model parameters

In a high variance model, we expect that a small perturbation data will lead to a large change in the prediction.

4.3 Regularisation

How can we reduce the variance of a model? If only we could *increase the eigenvalues* of the matrix $X^T X$. In fact, we can do this manually:

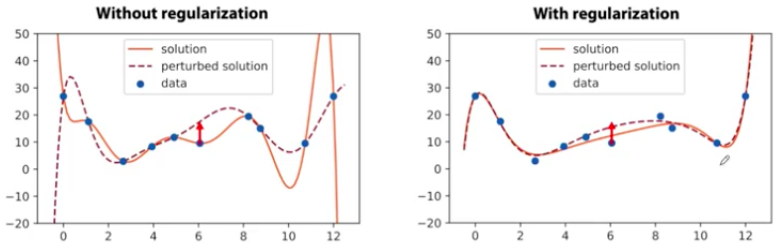
$$X^T X \to X^T X + \alpha I,$$

for $\alpha > 0$, with I a unit $d \times d$ matrix. This is called $L2$ *regularisation*.

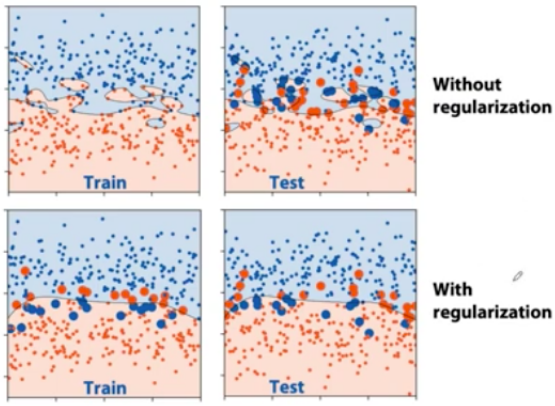
In this case, we change the solution to:

$$\hat{f}_{\tau}(\mathbf{x}) = \mathbf{x}^T (X^T X + \alpha I)^{-1} X^T \mathbf{y}_{\tau}.$$

Now the model is *no longer biased*. We *increased bias to reduce variance* (we no longer get zero for the bias term of this model.



By regularising the model, we increase the training loss and decrease the test loss. This improves the *generalisability* of the model.



In order to understand this trick better, let's reverse engineer the loss function it optimises. We have the solution $\hat{f}_\tau(\mathbf{x}) = \mathbf{x}^T(X^TX + \alpha I)^{-1}X^T\mathbf{y}_\tau$, so the $\boldsymbol{\theta}_\tau$ vector should be given by:

$$\boldsymbol{\theta}_\tau = (X^TX + \alpha I)^{-1}X^T\boldsymbol{y}_\tau.$$

Regrouping the terms, we have:

$$(XT^X + \alpha I)\boldsymbol{\theta}_\tau = X^T\boldsymbol{y}_\tau \qquad \Rightarrow \qquad X^T(X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau) + \alpha\boldsymbol{\theta}_\tau = \mathbf{0}.$$

In fact, this is the equation $\partial\mathcal{L}/\partial\boldsymbol{\theta}_\tau = \mathbf{0}$ for the loss function:

$$\mathcal{L} = ||X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau||^2 + \alpha||\boldsymbol{\theta}_\tau||^2.$$

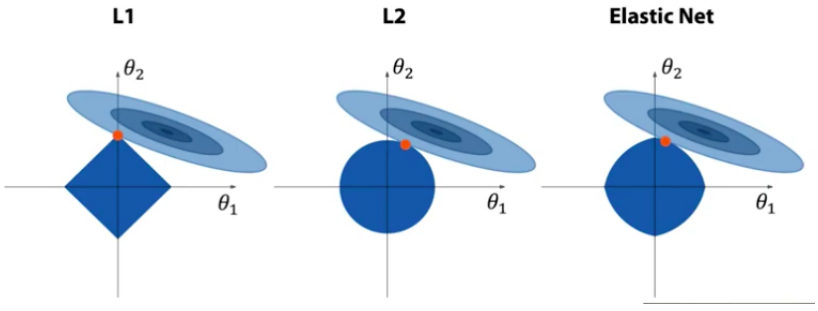
This explains the origin of the term $L2$ *regularisation*. The model minimises the MSE loss with an $L2$ *penalty theorem* (this model is also called *ridge regression*).

There are other similar regularisation methods:

- $L2$ *regularisation* or *ridge regression* takes as the loss function $\mathcal{L} = ||X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau||^2 + \alpha||\boldsymbol{\theta}_\tau||^2$.
- $L1$ *regularisation* or *lasso regularisation* takes as the loss function $\mathcal{L} = ||X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau||^2 + \alpha||\boldsymbol{\theta}_\tau||_1$ (where $||\cdot||_1$ denotes the $L1$ norm.
- *Elastic net regression* is a combination of the two:

$$\mathcal{L} = ||X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau||^2 + \alpha||\boldsymbol{\theta}_\tau||^2 + \beta||\boldsymbol{\theta}_\tau||_1.$$

The introduction of regularisation methods drive the parameters towards smaller values, yet they *induce different properties* of the solution.



For example, the $L1$ regularisation term has unit ball given by a diamond in θ_1, θ_2 space (i.e. the surface where $||\boldsymbol{\theta}||_1 = 1$). It's quite likely that among all the points at the perimeter of the diamond (where the penalty term does not change), the intersection is most likely at the coordinate axes. This means that this loss tends to *sparsify the solution* - it tends to set some parameters to zero.

For the $L2$ regularisation, we instead have *weight sharing* - it uniformly drags the values of the parameters towards zero.

The elastic net is a compromise between the two approaches.

4.4 Probabilistic view

Let's revisit our assumption about the data, $y = f(x) + \epsilon$. Let's now assume that the label noise is *normally distributed*, $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$. This means that the targets, given the objects, are also normally distributed:

$$y|x \sim \mathcal{N}(f(x), \sigma_\epsilon^2).$$

We want our model $\hat{f}_\theta(x)$ to fit the true dependence $f(x)$, i.e. we define a *probabilistic model*:

$$y|x \sim \mathcal{N}(\hat{f}_\theta(x), \sigma_\epsilon^2),$$

where the mean of the normal distribution is described by our function.

Our model can be fitted with the maximum likelihood approach:

$$L = \prod_{i=1,\dots,N} \mathcal{N}(y_i|\hat{f}_\theta(x_i), \sigma_\epsilon^2).$$

Maximising this with respect to θ is equivalent to minimising the negative log likelihood,

$$-\log(L) = \sum_{i=1,\dots,N} \log(\mathcal{N}(y_i|\hat{f}_\theta(x_i), \sigma_\epsilon^2)) = -\sum_{i=1,\dots,N} \left[\log\left(\exp\left(-\frac{(y_i - \hat{f}_\theta(x_i))^2}{2\sigma_\epsilon^2}\right)\right) - \log(\sqrt{2\pi}\sigma_\epsilon^2)\right]$$

$$= C \cdot \sum_{i=1,\dots,N} (y_i - \hat{f}_\theta(x_i))^2 + \text{constant}.$$

Thus we just get MSE loss! Thus MSE loss is equivalent to modelling the probability with a normal label noise.

You can do similar tricks for other loss functions and show that different probabilistic models lead to different loss functions.

4.5 Bayesian view

In the Bayesian view, we treat both data (X, y) and model parameters (θ) as random variables. We estimate the parameter distribution given the observed data via Bayes' rule:

$$p(\theta|X, y) = \frac{p(y|\theta, X)p(\theta)}{\int (p(y|\theta, X)p(\theta))\ d\theta}$$

We assume that θ, X are independent variables here. Here, $p(\theta)$ is our prior knowledge about the model parameters (our belief about how they are distributed before we see any data). The function $p(y|\theta, X)$ is our likelihood function. The distribution $p(\theta|X, y)$ is called the *posterior distribution*, and is our knowledge about the model after seeing data. The denominator

is called 'evidence' (the probability of observing the data when the parameter uncertainty is integrated out).

To get some estimate of the parameter, we can calculate the *maximum a posteriori* (where we ignore the denominator since is it is integrated over θ - the maximum is the same as the maximum of the numerator):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, p(y|\theta, X)p(\theta) = \underset{\theta}{\operatorname{argmin}}\, [-\log(p(y|\theta, X) - \log(p(\theta))]$$

Similarly, the maximum is the same as the minimum negative log likelihood. The term $\log(p(\theta))$ is called a *regularising* term.

Example 4.2: Suppose we model the data with a normal distribution $y|x \sim \mathcal{N}(\hat{f}_\theta(x), \sigma_\epsilon^2)$. Suppose the prior is normal too $\theta \sim \mathcal{N}(0, \sigma_\theta^2 I)$, where I is the unit matrix so that the parameters are uncorrelated. Then maximum a posteriori estimate corresponds to minimising the following loss:

$$\mathcal{L} = -\log(p(y|\theta, X)) - \log(p(\theta)) = C_1 \sum_{i=1,\ldots,N} (\hat{f}_\theta(x_i) - y_i)^2 + C_2||\theta||^2 + \text{constant.}$$

In other words a normal prior is equivalent to $L2$ regularisation of the parameters.

Choosing different priors will lead to different regularisation of the parameters.