4. Linear regression II

Machine Learning 1, UNIZG FER, AY 2022/2023

Jan Šnajder, lectures, v1.10

Last time we talked about **linear regression**: we defined the linear model, the quadratic error function, and optimization, which boiled down to computing the pseudoinverse of the design matrix. We also looked at probabilistic interpretation of regression and we concluded that – if we assume that labels contain **noise** that is normally distributed – then it is exactly the **quadratic error** which maximizes the probability of the labeled dataset. That provided us with a probabilistic justification for the use of quadratic error, and it also established (our first) link to the world of probabilistic machine learning.

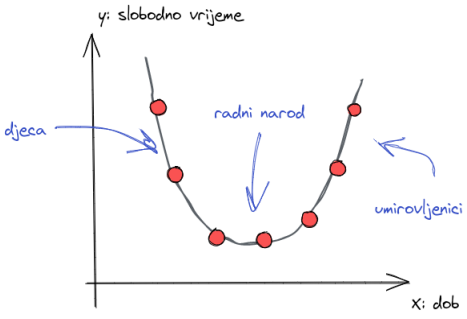
In this lecture, we continue to look at regression. In particular, we'll talk about **mapping into feature space** and **regularization**. It turns out these are two very important concepts in machine learning, and they are being employed in a wide range of machine learning algorithms, not just regression.

1 Nonlinear regression

For starters, let's recall the **linear regression model**. The model is defined as a linear combination of features. In other words: the dependent variable is a **linear function** of the independent variables:

$$h(\mathbf{x};\mathbf{w}) = \sum_{j=0}^{n} w_j x_j = \mathbf{w}^{\mathrm{T}}\mathbf{x}$$

However, in practice the will relationship between the dependent variable and the independent variables will often be **nonlinear**. For example, free time (as a dependent variable) based on person's age (as an independent variable):



▸ EXAMPLE

To motivate nonlinear regression, here are a few more examples:

- (1) The price of a car in terms of mileage – the price drops sharply and then stagnates (in the last lecture, we misused this example as an example of linear dependence, but in reality this is probably not the case)
- (2) Box Office Revenue given the cost of production – growing and then stagnating

- (3) Grade average with respect to the number of hours spent on lectures – a concave function (except for the machine learning course!)

1.1 Nonlinear regression model

To model nonlinear dependencies, we need a **nonlinear model**. For example, let's say we do simple regression ($n = 1$). Instead of a linear model

$$h(x; w_0, w_1) = w_0 + w_1 x \quad \in \mathcal{H}_1$$

we can define the model as a second-degree polynomial:

$$h(x; w_0, w_1, w_2) = w_0 + w_1 x + w_2 x^2 \quad \in \mathcal{H}_2$$

With such a model, we could very accurately model the above examples. Note that the model $\mathcal{H}_2$ is **more complex** (or "of a higher capacity") than the model $\mathcal{H}_1$: the model $\mathcal{H}_2$ includes nonlinear hypotheses in addition to all linear hypotheses included in model $\mathcal{H}_1$. Specifically, setting $w_2 = 0$ switches off the quadratic feature term, and the polynomial model degenerates to linear model. So, we have $\mathcal{H}_1 \subset \mathcal{H}_2$. Note also that this increased complexity comes with a price: the model $\mathcal{H}_2$ has one more parameter than model $\mathcal{H}_1$ (namely, parameter w_2).

Let's look at another example. We perform multiple regression with $n = 2$. For example, we want to predict apartment prices based on its size (x_1) and age (x_2). The linear model is:

$$h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$$

For a nonlinear model, we could again use a second-degree polynomial:

$$h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

Here, there is an interesting addition to the quadratic features: the $x_1 x_2$ feature. This feature is called an **interaction feature** (or a **cross-term**) and models the interaction between two input variables. For example, if the apartment is large and quite old, then this feature will have a high value (this, for example, may indicate a large old mansion ... maybe this is an exclusive and highly priced apartment?).

Using second-degree polynomials is just one possibility among many others for obtaining a nonlinear model. We could have achieved an even greater degree of nonlinearity had we defined the model as polynomial a third-degree polynomial or higher. The inclusion of interaction features is another way to make the model nonlinear. We can, of course, also combine these two ways. Furthermore, nonlinearity can be realized using other functions that are not polynomials (although we will not go into that here).

Let's look at a couple of regression models, based on polynomials, which differ in the way nonlinearities are modeled:

- Linear multiple regression:

  $$h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

  This model has multiple features (n in total), but it is completely linear, therefore it does not model any nonlinearity.

- Single (simple) polynomial regression ($n = 1$):

  $$h(x; \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_d x^d$$

  This model has only one feature ($n = 1$), but uses a polynomial of degree d to model the nonlinearity. Note that d here is a **hyperparameter**, as it determines the complexity of the model.

- Multiple polynomial regression ($n = 2, d = 2$):

$$h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

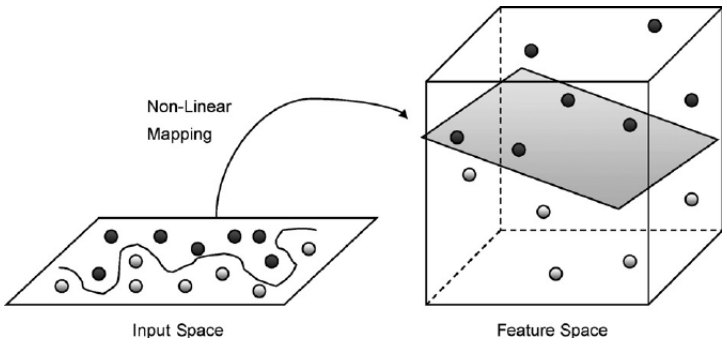
  This model has two features ($n = 2$) and uses a second-degree polynomial $d = 2$ to model nonlinearity. In addition to the quadratic features x_1^2 and x_2^2, the model also includes the cross-term feature $x_1 x_2$, which models the interaction between features x_1 and x_2 (in the language of statistics: a non-additive effect of independent variables on the dependent variable).

These are just a few examples; obviously, we can define a model using an arbitrary degree polynomial, we can chose to include or not to include interaction terms for certain pairs of variables, we can include interaction terms for more than two variables, etc. (And again note that nonlinearity can be realized by use of some other nonlinear function, not necessarily a polynomial function.)

1.2 Mapping to feature space

Although the models we have just considered are actually different, we would like to **unify** them somehow, so that we can treat them in the same way when doing optimization. This brings us to a rather powerful idea in machine learning: whenever we want to move from a linear model to a nonlinear one, instead of changing the model, let's change the data! The way we will change the data is to map them from the **input space** (hrv. *ulazni prostor*) into a different space. That other space is called **feature space** (hrv. *prostor značajki*). The mapping itself we'll refer to as **feature mapping** (hrv. *preslikavanje u prostor značajki*). The idea is that, if we manage to choose a suitable mapping, then the data – although nonlinear in the input space – will be linear in the feature space!

This mapping is ubiquitous in machine learning, both in regression and classification. For example, in classification:



By using a suitable (nonlinear) mapping from the input space to the feature space of a higher dimension, we can achieve that the examples, which were originally not linearly separable in the input space, become linearly separable in the feature space. In this example, we see that the linear boundary (the hyperplane) in the feature space, which perfectly separates the examples of the two classes, corresponds to a nonlinear boundary in the input space. However, instead of trying to find a nonlinear hypothesis in the input space, we use a nonlinear map to map the examples into the space of a larger dimension and then separate them there with a linear hypothesis. In the input space, it will seem as if we have separated the examples by a nonlinear hypothesis.

The basic idea will always be the same: a problem that is nonlinear in the input space will be mapped via a nonlinear mapping into a higher-dimensional feature space where it will (hopefully) become linear, and then we'll simply apply linear models in the feature space.

Let us now formally define how this mapping is done. We will introduce a fixed set of so-called **basis functions** (nonlinear functions of input variables):

$$\{\phi_0, \phi_1, \phi_2, \ldots, \phi_m\}$$

where $\phi_j : \mathbb{R}^n \to \mathbb{R}$. So, every basis function ϕ_j takes the whole example vector $\mathbf{x}$ and maps it to a single number. That number corresponds to one of m features in the m-dimensional feature space. There are a total of $m + 1$ base functions, one for each dimension of the m-dimensional feature space, plus one additional base function, ϕ_0. This additional base function is set to one, $\phi_0(\mathbf{x}) = 1$, and corresponds to the "dummy one", which we already used in the extended vector $\mathbf{x}$. The value of the **mapping function** is then defined as a vector obtained by applying individual basis functions to the example $\mathbf{x}$:

$$\boldsymbol{\phi} : \mathbb{R}^n \to \mathbb{R}^{m+1}$$
$$\boldsymbol{\phi}(\mathbf{x}) = \big(\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_m(\mathbf{x})\big)$$

The feature mapping function maps examples from the n-dimensional input space to the m-dimensional feature space. Typically, it will be the case that $m > n$, that is, we will be mapping into a space whose dimension is higher than that of the input space. Why? Because then the chances are greater that something that was nonlinear in the input space will become linear in the feature space.

We can now easily incorporate this mapping into our linear model, simply by replacing the vector $\mathbf{x}$ with the vector $\boldsymbol{\phi}(\mathbf{x})$:

$$h(\mathbf{x}; \mathbf{w}) = \sum_{j=0}^{m} w_j \phi_j(\mathbf{x}) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x})$$

where, recall, the vector $\boldsymbol{\phi}(\mathbf{x})$ is extended by the "dummy one" feature, i.e., $\phi_0(\mathbf{x}) = 1$.

With this we achieved our goal: examples from the input space are mapped to some suitable feature space, but the model itself remains unchanged – it is still a linear model, the only difference is that instead of the x_j features we now use their mapped counterparts, $\phi_j(\mathbf{x})$.

This model is called the **linear regression model**. It is called so because it is **linear in parameters** (w_j), although it may be nonlinear in the original (input space) features, depending on which mapping function we use.

Here one should pay attention to the subtle differences in terminology: a **linear regression model** is not the same as **linear regression**. Linear regression (which we covered last time) is a special case of the linear regression model where we use no feature mapping at all.

Now that we have defined the feature mapping function in general, let's look at some specific cases. Using and adequate feature mapping function we easily obtain any of the previously considered linear regression models.

- Linear multiple regression:

  $$\boldsymbol{\phi}(\mathbf{x}) = (1, x_1, x_2, \ldots, x_n)$$

  We don't actually use any mapping here, i.e., the mapping function is actually an identity function (extended with the "dummy one" feature), $\boldsymbol{\phi}(\mathbf{x}) = (1, \mathbf{x})$. With such a mapping function, our linear regression model in fact boils down to linear regression.

- Simple polynomial regression ($n = 1$):

  $$\boldsymbol{\phi}(x) = (x, x^2, \ldots, x^d)$$

  Feature x is being mapped into a polynomial of degree d, thereby projecting from a 1-dimensional input space ($n = 1$) to a d-dimensional feature space ($m = d$).

- Multiple (bivariate) second-degree polynomial regression ($n = 2, d = 2$):

$$\boldsymbol{\phi}(\mathbf{x}) = (1, x_1, x_2, x_1 x_2, x_1^2, x_2^2)$$

Here we map from a two-dimensional input space to a 5-dimensional feature space using a mapping function defined as a second-degree polynomial with an interaction feature. The feature space consists of the original features x_1 and x_2, and well as the interaction feature $x_1 x_2$ and two quadratic features, x_1^2 and x_2^2.

1.3 Optimization procedure

This settles the question of the model. As for the loss function (and, consequently, the error function), it is the same as for linear regression, because the outputs of the models are the same: the quadratic loss function (and the corresponding quadratic error function). What remains is to work out an **optimization procedure**. How does the optimization procedure look like in case of the linear regression model?

The good news is that essentially nothing changes with respect to what we have already derived! This is because the model with the mapping function is still a linear model. The only change is that, instead of performing the optimization using the original design matrix $\mathbf{X}$, we will do it using a design matrix on which we have applied the feature map. We will denote this matrix as $\boldsymbol{\Phi}$.

So, instead of the original design matrix (which is defined in terms of the original features):

$$\mathbf{X} = \begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \dots & x_n^{(2)} \\ \vdots & & & \\ 1 & x_1^{(N)} & x_2^{(N)} \dots & x_n^{(N)} \end{pmatrix}_{N \times (n+1)}$$

we will now be using a design matrix in which we have applied the feature mapping function $\boldsymbol{\phi}(\mathbf{x})$ to each example:

$$\boldsymbol{\Phi} = \begin{pmatrix} 1 & \phi_1(\mathbf{x}^{(1)}) & \dots & \phi_m(\mathbf{x}^{(1)}) \\ 1 & \phi_1(\mathbf{x}^{(2)}) & \dots & \phi_m(\mathbf{x}^{(2)}) \\ \vdots & & & \\ 1 & \phi_1(\mathbf{x}^{(N)}) & \dots & \phi_m(\mathbf{x}^{(N)}) \end{pmatrix}_{N \times (m+1)} = \begin{pmatrix} \phi(\mathbf{x}^{(1)})^{\mathrm{T}} \\ \phi(\mathbf{x}^{(2)})^{\mathrm{T}} \\ \vdots \\ \phi(\mathbf{x}^{(N)})^{\mathrm{T}} \end{pmatrix}_{N \times (m+1)}$$

When doing this, we shouldn't forget about the "dummy one" feature (the first column of the $\boldsymbol{\Phi}$ matrix). To the optimization procedure, of course, it is irrelevant what matrix it works with. Earlier we had:

$$\mathbf{w} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} = \mathbf{X}^{+}\mathbf{y}$$

and now we simply substitute matrix $\boldsymbol{\Phi}$ for matrix $\mathbf{X}$:

$$\mathbf{w} = (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{y} = \boldsymbol{\Phi}^{+}\mathbf{y}$$

where $\boldsymbol{\Phi}^{+} = (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathrm{T}}$ is the pseudoinverse of the design matrix $\boldsymbol{\Phi}$. For the pseudoinverse matrix $\boldsymbol{\Phi}$, of course, the same remarks apply as the ones we made in the last lecture for the pseudoinverse of the design matrix $\mathbf{X}$.

We thus arrived exactly at where wanted: we have one unified regression algorithm, regardless of which mapping function $\boldsymbol{\phi}$ we choose. Isn't that great?! But the question lingering now is: which mapping function to choose? This leads us to the next topic: overfitting.

2 Overfitting

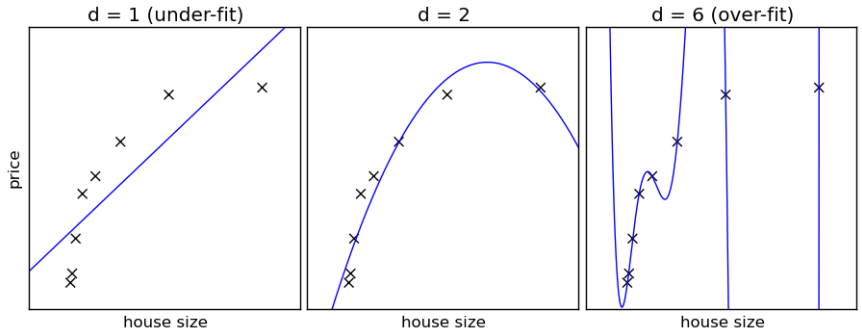
Last week we said that if we have at our disposal a family of models, what we need to do is choose the optimal model for our problem. Otherwise the model will be either **overfitted** or **underfitted**. We also said that different models are indexed by their **hyperparameters**.

As we have seen, with a linear regression model we can realize models of different complexities by choosing an appropriate mapping function. What, then, is the hyperparameter of a linear regression model? Well, the hyperparameter is precisely the mapping function $\boldsymbol{\phi}$. Namely, depending on which mapping function we choose, the model will be more or less complex. For example, the second-degree polynomial model is more complex than the linear model. Further, the third-degree polynomial model is even more complex, and so on. At the same time, we need to be aware that the more complex the model, the more parameters it has, which will make it easier to overfit it.

▸ **EXAMPLE**

Let's say we want to train a linear regression model for predicting apartment prices in Boston. For simplicity, consider a simple (i.e., single input) regression, $n = 1$. Let the independent variable be the size of the apartment. We consider three models: a first-degree polynomial (a line), a second-degree polynomial (a parabola), and a sixth-degree polynomial ("a sextic polynomial", in case you were interested). As we already know, these models can easily be defined using an appropriate feature mapping function $\boldsymbol{\phi}(\mathbf{x}) = (x_1, x_1^2, \ldots, x_1^d)$, where d is the desired degree of the polynomial.

Let the set of labeled examples consist of six examples, i.e., $|\mathcal{D}| = 6$. After training the three models, we get the following curves in the input space:



What happened here? The first-degree polynomial does not seem to be very suitable, as the predictions of that model deviate significantly from the target values. That model is underfitted. The second-degree polynomial seems to do quite good on our data set. We might be inclined to say that this model is of optimal complexity for dataset $\mathcal{D}$ (although common sense tells us that the parabola is probably not the best way to model the price of an apartment as a function of its size; the relationship is probably nonlinear, but we do not expect prices for larger apartments to start falling). The sixth degree polynomial is the absolute champion in the sense that the regression curve really passes through each and every point, however, it is obvious that this model is overfitted. While it seems reasonable to assume that the price of an apartment somewhat nonlinearly depends on it size, it is unlikely that this dependence is as hysterical as the hypothesis modeled by a sixth-degree polynomial. The model has learned the noise in the data, resulting in a hypothesis that oscillates heavily and that will obviously generalize poorly to unseen data.

How to prevent the model from overfitting? This is a general question, and whatever answer we come up with, it will apply to the regression algorithm as well as most other machine learning algorithms. The main ways to prevent overfitting are as follows:

- **Use more training examples** – intuitively, the more examples we have, the harder it will be for the regression curve to pass through each of them. When the number of examples is

such that the curve cannot pass through each one of them, the quadratic loss will make sure that the parameters are such that the curve minimizes the sum of squared differences, which means that the curve will oscillate less. So, the more examples we have, the more stable the hypothesis, that is, the less overfitted the model. While this is a straightforward recipe for preventing overfitting, it is generally difficult to apply in practice: the number of labeled examples is typically limited, and obtaining new examples might not be straightforward (either we simply do not have them, or labeling them is expensive, or it's time-consuming);

- **Choose a model by cross-validation** - we already know this. Specifically, in the case of a linear regression model, this would mean that we train models with different feature mapping functions on the training set, apply the trained models on the test set, and then choose the model with the smallest error on the test set, because that model generalizes best to unseen data. If the feature mapping function is defined as a polynomial of degree d, then we can establish a total order among the models based on their complexities, where complexity is determined by the hyperparameter d. The cross-validation then amounts to testing the hyperparameter d from a predetermined interval, e.g., $d \in \{1, 2, \ldots, 6\}$;
- **Select a subset of features** – the more features a model has, the more complex it is. If we reduce the number of features, we get a simpler model. Of course, it wouldn't be wise to remove features that are useful; ideally, we would throw out only those features that are useless. The process of finding the optimal subset of features is called **feature selection** (engl. *odabir značajki*), and we'll talk about it at the very end of the course. Since the feature set actually defines the model, the process of feature selection can be viewed as model selection;
- **Reduce the dimensionality of the input space** – instead of throwing out unnecessary features, we can map the entire input space into a feature space of a lower dimension. The so-obtained features won't match the original features, but sometimes this is not a problem;
- **Regularization** – regularization is a method that implicitly prevents the selection of overly complex models during the procedure of optimizing the model's parameters. Regularization is a universal idea in machine learning, and we will talk about it today;
- **Bayesian regression** (i.e., more generally, **Bayesian model selection**) – Bayesian procedures in statistics and machine learning rely on the idea that model parameters, as well as examples $\mathbf{x}$ and labels y, should all be treated as random variables, for which suitable distributions need to be defined in advance (prior distributions), and learning is then reduced to estimating these parameters and applying the Bayes' theorem to obtain a posterior distribution of labels y for input example $\mathbf{x}$. In this course, we will (unfortunately) not have anything to say about Bayesian methods.

The situation with these techniques is as follows. The first technique (shove in more examples) is not always feasible. The next three techniques are often used. They strictly separate the model selection phase from the model training phase. On the other hand, the last two techniques (regularization and Bayesian regression) to some extent combine the two phases.

We will focus on **regularization**, as one very fundamental, very effective, and frequently used technique in machine learning.

3 Regularization

The story about regularization begins from a simple observation: with linear models, the more complex the model, the higher the parameter values $\mathbf{w}$. Take, for example, the second-degree polynomial model:

$$h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

If the weights w_3, w_4, or w_5 are of a large magnitude (either positive or negative), the hypothesis will be highly nonlinear. On the other hand, the closet the weights w_3, w_4, and w_5 to zero, the less nonlinear the hypothesis. In the extreme, if $w_3 = w_4 = w_5 = 0$, the model effectively degenerates to a linear model.

This example indicates that the complexity of the linear regression model directly depends on the feature weights, and that models that are overfitted will have many of its weights set to high values. In order, then, to prevent this, we will, already during model training, **constrain the magnitude of model's parameters**. We will do this by **penalizing** hypotheses with high parameter values. We refer to this mechanism as **regularization**.

In practice, we'll be doing this as follows: we'll start with a relatively complex model, in order to avoid underfitting, but then we use regularization to effectively constrain its complexity. So, we start from a very complex, almost lavish model, but then we restrain it, so as not to end up with too much of an "unbridled" model. If we do this restraint wisely, we will be able to prevent overfitting, that is we will strike a balance between model simplicity and complexity.

The ideal goal of regularization is to push down to zero as many parameters (weights) as possible. Not only will this give as a simpler model from a more complex one, but, if many weights are pushed down exactly to zero, we obtain There is another advantage here: if we tighten the weights to zero, then we will gain so-called **sparse models** (hrv. *rijetki modeli*). In machine learning, we like sparse models because they are: (1) less prone to overfitting, (2) computationally simpler when doing prediction, and (3) more interpretable – we know which features definitely do not affect the output (which might be important in statistical data analysis).

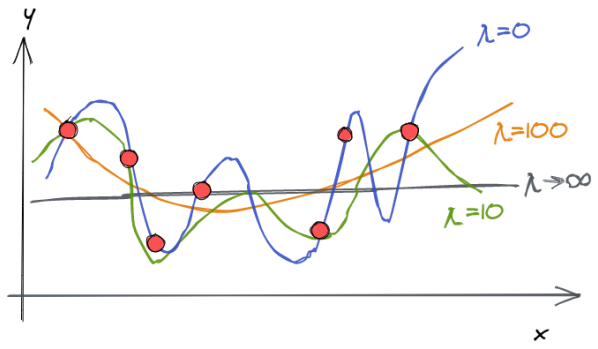
3.1 Regularized regression

Let's look now at how we actually implement regularization. This is quite simple: we extend the error function (which we minimize) so that, in addition to the empirical error, it contains one more term. This term will characterize the complexity of the model as a function of its weights. Doing so we get the **regularized error function**:

$$E_R(\mathbf{w}|\mathcal{D}) = E(\mathbf{w}|\mathcal{D}) + \underbrace{\lambda\Omega(\mathbf{w})}_{\text{reg. term}}$$

In this expression, $\Omega(\mathbf{w})$ is the **regularization term**, while λ is the so-called **regularization factor**. If $\lambda = 0$, then we're back to the unregularized error function. Using higher values for λ will lead to complex models being penalized more and, consequently, will result in a reduction in the effective complexity of the model.

▸ EXAMPLE



We train an L2 regularized simple ($n = 1$) regression model with a mapping function defined as a 10-degree polynomial. Consider four models, differing in the value of the regularization factor λ. If we do not regularize ($\lambda = 0$), the 10-degree polynomial 10 achieves perfect accuracy on our dataset of 7 examples. As we increase the value of the regularization factor λ, we push the weights $\mathbf{w}$ more and more toward zero, resulting in a smoother regression curve in the input space. E.g., a 10-degree polynomial regularized with $\lambda = 100$ could in our case effectively correspond to a second-degree polynomial (a parabola). In the extreme, when $\lambda \to \infty$, all weights except weight w_0 are pushed to zero (we'll explain below why w_0 is exempted), and we effectively obtain a line of slope zero that intersect the y-axis at the mean value of the y labels. Of course, such strong regularization is not what we want in practice.

Let's look now at how to concretely define $\Omega(\mathbf{w})$. The value of this term depends on the weights $\mathbf{w}$. We need some function of the weights $\mathbf{w}$, such that its value is large when the weights are large, and small when the weights are small. Also note that $\mathbf{w}$ is actually a **vector** of weights. What would be a sensible choice for such a function? The most straightforward option is a **vector norm**. Namely, the norm is a function that gives the length of the vector, and its value will increase as the magnitudes of the vector components increase.

In math, there is a general class of vector norms called the p-**norm**. Thus, as a general case, we can define the regularization expression $\Omega(\mathbf{w})$ as a **p-norm of the weight vector**:

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_p = \Big(\sum_{j=1}^{m} |w_j|^p \Big)^{\frac{1}{p}}$$

Different p-norms can be visualized as follows:

The figures show where in a 3-dimensional space lie points (or tips of the corresponding vectors from the origin) that are equidistant from the origin (i.e., vectors that have the same norm). In other words, the figures show the shape of the (hyper)surfaces defined by the function $\|\mathbf{w}\|_p = \text{const}$. For the infinite norm this looks like a cube, for the 2-norm it is a sphere, for the 1-norm it is an octahedron, for the 0-norm it's the points on the vertices of the octahedron (lying on the axes). In higher dimensions we would have a hypercube, a hypersphere, a hyperoctahedron, and points at the vertices of the hyperoctahedron, respectively.

In machine learning, we typically use the norms with $p = 2$ and $p = 1$, but we will also mention the norm with $p = 0$. We refer to these as the L2-norm, L1-norm, or L0-norm, respectively. Their definitions follow from the above definition for the general p-norm:

- **L2-norm** ($p = 2$), also known as the **Euclidean norm**:

$$\|\mathbf{w}\|_2 = \sqrt{\sum_{j=1}^{m} w_j^2} = \sqrt{\mathbf{w}^{\mathrm{T}}\mathbf{w}}$$

- **L1-norm** ($p = 1$), also known as **Manhattan distance**:

$$\|\mathbf{w}\|_1 = \sum_{j=1}^{m} |w_j|$$

- **L0-norm** ($p = 0$), which is in fact equal to the number of non-zero elements of the vector (i.e., the number of features that are not removed):

$$\|\mathbf{w}\|_0 = \sum_{j=1}^{m} \mathbf{1}\{w_j \neq 0\}$$

Note – and this is very important – that weight w_0 is not regularized. Why? Because w_0 determines the y-intercept of the line (or generally of the hyperplane). If we predict the foot size with respect to age, then we certainly don't want to have $w_0 = 0$ because that would mean foot size of a newborn baby is zero.

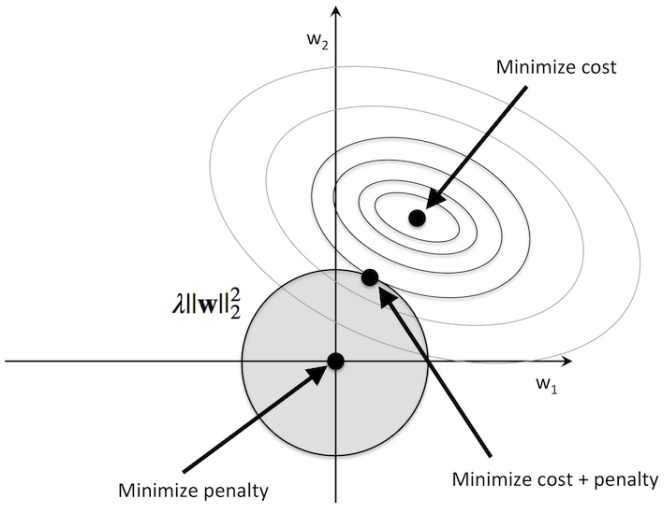
3.2 Regularized linear regression model

Let's now look at what regularization looks like specifically for regression. **L2 regularization** (or Tikhonov regularization) gives us the so-called **ridge regression** (hrv. *hrbatna regresija*):

$$E_R(\mathbf{w}|\mathcal{D}) = \frac{1}{2}\sum_{i=1}^{N}\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

Notice that here we use the squared L2 norm and $\lambda/2$. These two choices, together with $1/2$ in front of the sum, are for later mathematical convenience. L2 regularization penalizes a hypothesis proportionate to the extent to which its weights deviate from zeros in terms of quadratic differences. This means that larger weights will be penalized more than smaller weights; this insight will prove useful a bit later.

L2 regularization has a fairly intuitive geometric interpretation. As we know hypotheses are functions defined by parameters $\mathbf{w}$. Consequently, the error $E(\mathbf{w}|\mathcal{D})$ is also a function of parameters $\mathbf{w}$. We can sketch how that function looks like in parameter space:



The figure shows the isocontours (a set of points at which the function assumes a constant value) of the error function $E(\mathbf{w}|\mathcal{D})$ in the parameter space $w_1 \times w_2$ (parameter w_0 is omitted for clarity). The elliptical contours in the upper right part of the figure correspond to the isocontours of the unregularized error function $E(\mathbf{w}|\mathcal{D})$ defined as the sum of squares residuals. This function is convex, and its minimum is at the center of the isocontour. The circle in the lower left part of the figure corresponds to the isocontour of the L2 regularization term, the minimum of which is at the origin (because there the norm of the weight vector is equal to zero). The regularized error function $E_R(\mathbf{w}|D)$ is obtained by summing these two curves. The sum of two convex functions gives again a convex function. The isocontours of this convex

function are not shown in the figure, but its minimizer is shown, as a point positioned between the minimizer of the unregularized error function and the minimizer of the L2 regularization term (the origin). We can therefore view regularization as an attraction force that pulls the minimizer of the regularized error function closer to the origin of the parameter space.

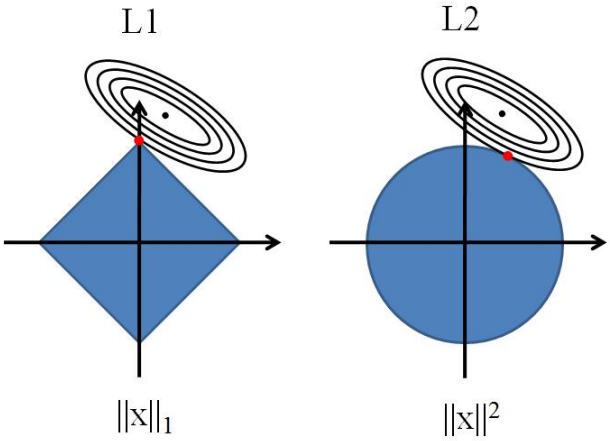
Instead of L2 regularization, we can choose to do **L1 regularization** or **LASSO regularization** (least absolute shrinkage and selection operator):

$$E_R(\mathbf{w}|\mathcal{D}) = \frac{1}{2}\sum_{i=1}^{N}\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\|\mathbf{w}\|_1$$

L1 regularization penalized the hypothesis to the extent that the absolute value of its weights deviate from zeros.

What is the difference between L2 and L1 regularization? The difference shows when the weights are small, i.e., close to zero. L1 regularization doesn't square the difference, hence it will penalize these small weights more than L2 regularization would. L2 regularization, on the other hand, will only slightly penalize the small weights, which will actually make it very difficult to push them all the way down to zero. Therefore, L1 regularization will generally produce models where more weights are pushed to zero, that is, L1 regularization will result in **sparser models** than L2 regularization.

The difference between L2 and L1 regularization can also be explained graphically, in the parameter space:



The figure depicts L1 regularization (left) and L2 regularization (right). The isocontours of the L1 regularization term are generally a hyperoctahedron, which in a two-dimensional parameter space is a square. In contrast, the isocontours of the L2 regularization term are hyperspheres, which in a two-dimensional parameter space is a circle. The figure shows that, for L2 regularization, because the isocontours of the regularization term are not rounded but have vertices, it is easier for the minimizer of the regularized error function to be found on one of the coordinate axes of the parameter space, i.e., in one of the vertices of the square (because the vertices are on the axes). Furthermore, if the minimizer is on one of the axes of the parameter space, then this means that the other coordinate (i.e., the weight) is equal to zero. In a three-dimensional parameter space, the isocontour is an octahedron, and if the minimum is found in one of the vertices of the octahedron, then the other two weights will be equal to zero. If the minimum is found on an edge of the octahedron, then one weight will be equal to zero. In multidimensional space, each time the minimum is found in the vertex or on edge of a hyperoctahedron, a certain number of weights will be equal to zero. In a multidimensional space, the likelihood of this happening increases with the number of dimensions, as the number of vertices and edges of an hyperoacteader increase as well. From this we then conclude that L1 regularization more easily leads to sparse models (many weights are pushed down to zero) than L2 regularization.

Finally, we can also consider **L0 regularization**, where the regularized error is defined as follows:

$$E_R(\mathbf{w}|\mathcal{D}) = \frac{1}{2}\sum_{i=1}^{N}\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{m}\mathbf{1}\{w_j \neq 0\}$$

L0 regularization penalized the hypothesis proportionate to the number of non-zero features. This means that it treats each feature as being either included or excluded, on or off. This then means that L0 regularization effectively selects the best subset of features, which means that it performs **feature selection** (hrv. *odabir značajki*).

Now that we have introduced L2, L1, and L0 regularization, a natural question to ask is: which one is the best? The answer is that in principle, we prefer regularization that gives models that are as sparse as possible. Thus, we prefer L0 over L1, and L1 over L2. However, there is also the issue of computational complexity. Namely, L0 regularization is an **NP-complete problem**: it has no tractable solution (all 2^m combinations of features would have to be tested). On the other hand, L1 regularization is tractable, but admits **no closed-form solution**. It turns out that, for the linear regression model, only the L2 regularization has a closed-form solution.

In what follows, we'll look into L2 regularized regression (i.e., ridge regression), for the very reason that its optimization procedure has an analytical solution. Ridge regression is widely used in practice. Also used in practice are the L1 regularization (which does not have an analytical solution, so the optimization has to be done iteratively) and a combination of L1 regularization and L2 regularization called the **elastic net** (hrv. *elastična mreža*). But we won't talk about this.

3.3 Ridge regression: optimization

As we have said, L2 regularized regression, or ridge regression, does have a solution in closed form. And the solution can be easily derived. Let's see how.

First recall how we derived the solution for weights $\mathbf{w}$ for the unregularized linear regression model:

$$\begin{aligned} E(\mathbf{w}|\mathcal{D}) &= \frac{1}{2}(\boldsymbol{\Phi}\mathbf{w} - \mathbf{y})^{\mathrm{T}}(\boldsymbol{\Phi}\mathbf{w} - \mathbf{y}) \\ &= \frac{1}{2}(\mathbf{w}^{\mathrm{T}}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\mathbf{w} - 2\mathbf{y}^{\mathrm{T}}\boldsymbol{\Phi}\mathbf{w} + \mathbf{y}^{\mathrm{T}}\mathbf{y}) \end{aligned}$$

Next, to find the minimum, we calculated the gradient of this function, equated it with zero, and expressed $\mathbf{w}$:

$$\begin{aligned} \nabla_{\mathbf{w}} E &= \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\mathbf{w} - \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{y} \\ \mathbf{w} &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{y} \end{aligned}$$

We will now do the exact same thing as before, only with the regularization factor (shown in red) added to the error function:

$$\begin{aligned} E_R(\mathbf{w}|\mathcal{D}) &= \frac{1}{2}(\boldsymbol{\Phi}\mathbf{w} - \mathbf{y})^{\mathrm{T}}(\boldsymbol{\Phi}\mathbf{w} - \mathbf{y}) + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \\ &= \frac{1}{2}(\mathbf{w}^{\mathrm{T}}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\mathbf{w} - 2\mathbf{y}^{\mathrm{T}}\boldsymbol{\Phi}\mathbf{w} + \mathbf{y}^{\mathrm{T}}\mathbf{y} + \lambda\mathbf{w}^{\mathrm{T}}\mathbf{w}) \end{aligned}$$

Now we calculate the gradient of the function, equate it with zero, and express $\mathbf{w}$:

$$\begin{aligned} \nabla_{\mathbf{w}} E_R &= \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\mathbf{w} - \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{y} + \lambda\mathbf{w} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\mathbf{I})\mathbf{w} - \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{y} = 0 \\ \mathbf{w} &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\mathbf{I})^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{y} \end{aligned}$$

Here we made use of the equivalence $\frac{\mathrm{d}}{\mathrm{d}\mathbf{x}}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} = \mathbf{x}^{\mathrm{T}}(\mathbf{A} + \mathbf{A}^{\mathrm{T}})$ from matrix differential calculus, with $\mathbf{A} = \mathbf{I}$, and also the equivalence $\frac{\mathrm{d}}{\mathrm{d}\mathbf{w}}\mathbf{w}^{\mathrm{T}}\mathbf{w} = 2\mathbf{w}$.

The $\lambda\mathbf{I}$ is a matrix of dimensions $(m+1) \times (m+1)$ with λ on the diagonal and zeros off the diagonal. By adding that matrix to the Gram matrix, we effectively achieve regularization. However, let us recall that we want to exclude weight w_0 from regularization. This means that the matrix $\lambda\mathbf{I}$ is actually defined so that, besides the off-diagonal elements, the upper-left element also equals zero, i.e., $\lambda\mathbf{I} = \mathrm{diag}(0, \lambda, \ldots, \lambda)$.

We can see that regularization has "embedded" itself quite neatly in the least squares solution. The solution, however, is not directly a pseudoinverse of the Gram matrix $\mathbf{\Phi}$ anymore, instead it should be computed as the inverse of the sum of the Gram matrix and the matrix $\lambda\mathbf{I}$. From a computational perspective, this extension is trivial. However, the fact that regularization has boiled down to augmenting the diagonal values of the Gram matrix is interesting. And here we have an interesting story to tell ...

4 Regularization and multicollinearity

Let's go back a bit to computing the inverse of the design matrix $\mathbf{\Phi}$, which we talked about briefly last time. We said that we cannot compute the inverse directly, because it can easily happen that the number of examples does not match the number of parameters, which would render the equation system **overdetermined or underdetermined**, that is, the design matrix would not be square and hence would have no inverse. Moreover, we said that even if the design matrix miraculously happens to be square, it can still be the case that it has no inverse, namely if the corresponding system of equations is inconsistent or has multiple solutions. The solution emerged in the form of the least squares procedure, which brought us very quickly to the **pseudoinverse** of the design matrix:

$$\mathbf{w} = (\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi})^{-1}\mathbf{\Phi}^{\mathrm{T}}\mathbf{y}$$

We also said that the pseudoinverse always exists. Thus, even when we cannot compute the pseudoinverse of the Gram matrix by means of matrix inverse, we can compute it using singular value decomposition (SVD).

However, this is not always what we really want. Allow me to explain.

First, let's look at when it is possible to compute the pseudoinverse of the Gram matrix using matrix inverse. The Gram matrix is of dimensions $(m+1) \times (m+1)$, i.e., it certainly is square. But to be invertible, it also must be of full rank. When will the Gram matrix be of full rank? As we mentioned last time, the rank of the Gram matrix equals the rank of the design matrix. This means that the Gram matrix will be of full rank if and only the rank of the design matrix is $(m+1)$, i.e., if the design matrix has $(m+1)$ linearly independent columns, or, otherwise said, if all features are **linearly independent**. Unfortunately, a frequent problem in practice is that the columns happen to be linearly dependent. Why is this the case? Because a dataset often contains **redundant features**. Let's look at an example.

▸ **EXAMPLE**

We do personal income prediction, using as features the person's age in days and, as another feature, the person's age in years. But, these two features are almost perfectly linearly correlated. In this case, the design matrix $\mathbf{\Phi}$ may look as follows:

$$\mathbf{\Phi} = \begin{pmatrix} 1 & 9511 & 26 \\ 1 & 11340 & 31 \\ 1 & 8201 & 22 \\ 1 & 18022 & 49 \end{pmatrix}$$

The first column is the "dummy one" feature $x_0 = 1$, the second column is the feature x_1 corresponding to person's age in days, while the third column is the feature x_2 corresponding to person's age in years. The second and third columns are linearly dependent: we can obtain the third column from the second one by dividing the second column by 365.25 and rounding it down to an integer. This is not a perfect linear relationship (because we're rounding, and thus the number of days gives a more accurate information than the number of years), but it's close enough to a perfect linear relationship to be problematic in practice. If we don't do rounding, but represent years as a real numbers, we get a perfect linear dependence between the second and the third column.

We call this phenomenon **multicollinearity**: two or more input variables are correlated, making it possible to very accurately predict one variable as a linear combination of one or more other variables. The extreme case of multicollinearity is **perfect multicollinearity**, as in the previous example (had we not rounded down the number of years). In practice, however, perfect multicollinearity is rare. In contrast, (non-perfect) multicollinearity happens quite often. For example, if we do personal income prediction based on person's age and years of service, then these two variables might easily come out as highly correlated, giving rise to multicollinearity.

Let's now go back to our design matrix. What happens if there exist a feature that is a perfect linear combination of one or more other features? In this case, the design matrix $\boldsymbol{\Phi}$ will not be of rank $m + 1$. Furthermore, since the Gram matrix (the matrix of scalar products) $\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$ has the same rank as the design matrix, which means it will not have a full rank. This then means that the Gram matrix will have no inverse, i.e., it will be **singular**. If we really want to compute the pseudoinverse of $\boldsymbol{\Phi}$, we can do this using SVD, but the problem in this case will that the solution we get will be very **unstable**. What does that mean? It means that the solution for $\mathbf{w}$ will be **very sensitive to changes in the values of input variables**. A slight change in the input can give a very large change in the weights.

What if the features are **multicollinear** but not perfectly multicollinear? In this case, the design matrix will be full rank, as will the Gram matrix, so it won't be singular and we can compute its inverse, that is, we can compute the pseudoinverse using matrix inverse. However, we need to be aware of the fact that in this case the Gram matrix, albeit not singular, will be very close to being singular, which is why solution will be very again **unstable**.

Such unstable solutions are what we typically observe in **overfitted hypotheses**. In regression, this means that small changes in labeled data will produce radically different outputs. For example, if we use a 10-degree polynomial, small changes in the input will yield very different hypotheses. Take another look at the example predicting apartment prices in Boston. There we had a polynomial of degree 6, whose curve will go very wild for small changes in input data. What this means is that small changes in the design matrix will give hypotheses $h(x; \mathbf{w})$ have very different parameters $\mathbf{w}$. This is a sign of overfitting.

Luckily, there is way to detect instability of a solution. In numerical mathematics we talk about the **condition number** of a matrix. We don't have time to go into details here, but it's enough for us to know that when that number is large, then our solution is unstable and the model is most certainly overfitted. We say such a matrix is **ill-conditioned** (hrv. *loše kondicionirana*).

Overfitting the model is something we definitely want to avoid. Note that if the model is complex but the data is few, i.e., $m \geqslant N$, we will likely see the model overfitting. In this case the design matrix will be **"wider than it is tall"**, the rank of the matrix will certainly be less than $(m + 1)$, that is, it will be at most N. The Gram matrix will be of the same rank, hence it will not be of full rank, and we certainly have multicollinearity!

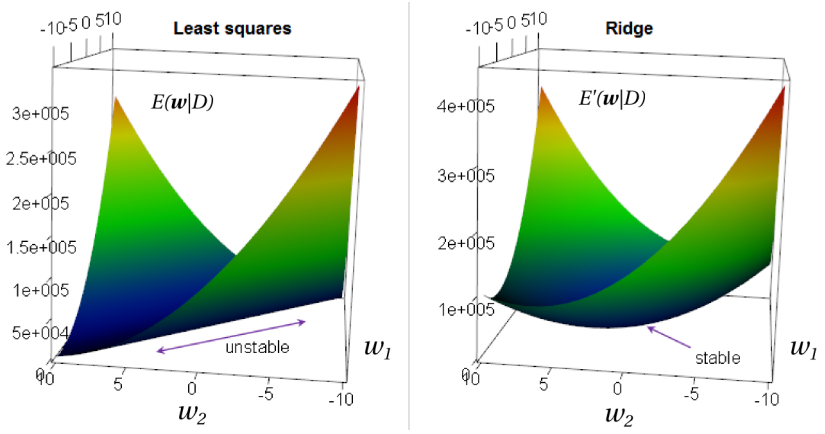
How can multicollinearity be avoided? A sensible first thing to do is to analyze the correlations between pairs of features before training the model and throw out the ones that are redundant. So that's one option. The alternative option is something we are already familiar with: **regularization**! Now that we know that the overfitting problem is due to the ill-conditioning of

the Gram matrix, let's look at how regularization solves this problem. Specifically, let's look at what regularization actually does to the Gram matrix: matrices:

$$\mathbf{w} = (\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi} + \lambda\mathbf{I})^{-1}\mathbf{\Phi}^{\mathrm{T}}\mathbf{y}$$

We see that regularization adds a λ to a different row in each column of the Gram matrix. Thus, if the columns were linearly dependent or almost linearly dependent, they will now be less so. The larger the value of λ, the closer the matrix to the diagonal matrix and the farther away from a singular matrix, and, consequently, the lower the multicollinearity. Ultimately, when the matrix is almost diagonal, there will certainly be no multicollinearity. Therefore, what regularization actually does is it improves the condition of the Gram matrix (reduces its condition number). We say we have **reconditioned** the matrix.

Finally, it is interesting to look at the effect of matrix reconditioning on the shape of the **error function** in parameter space:



The figures show a graph of the regression error function $E(\mathbf{w}|\mathcal{D})$ (which, recall, is defined via the quadratic loss function) in the parameter space $w_1 \times w_2$ (again, for clarity, i.e., to be able to depict the graph, we omitted the weight w_0, but we know it is also being minimized over, although it is excluded from regularization). As always, we are trying to find a weight vector $\mathbf{w}$ that minimizes the error. The left figure shows the unregularized empirical error, i.e, the function being minimized by the ordinary least squares procedure, while the right figure shows the regularized empirical error, i.e, the function being minimized by ridge regression (which corresponds to the least squares criterion plus a regularization term). In the case of the unregularized error, we see that the point of minimum is attained at a point in a long valley. In fact, all points at the bottom of that valley, which in this figure make up a line, have the same function value, so we have an infinite number of points where the error function takes a minimum. We call this valley the **"ridge"**: an area of unstable solution, where infinitely many solutions are possible. The figure on the right shows the regularized error function. The effect of regularization, i.e., the conditioning of the design matrix, is that the error function becomes "more convex". This reduces the instability of the solution, and we actually end up in a situation where there is only one minimum. Also, it is intuitively clear that the more convex the error function, i.e., the stronger the regularization, the more stable the solution will be. And that's exactly why we refer to L2 regularized regression as **ridge regression**: because it can work with "ridges" (or valleys).

What is the point of this story? A concrete point is that we can conceive of regularization as a way of "fixing" the Gram matrix which takes us away from unstable solutions (overfitted hypotheses). A more general point is that in machine learning we can often look at one and the same thing from a variety of interrelated perspectives. One of the main purposes of this course it make you aware of this.

5 Notes

We'll finish with a few general remarks regarding the linear regression model.

- The magnitude of w_i parameter corresponds to **feature importance**, and the sign indicates how it affects (positively or negatively) the output value.
- However, if the model is overfitted (there is multicollinearity), the magnitude of the parameter is meaningless. In that case, one should regularize the model.
- Regularization **prevents overfitting** by reducing model complexity vie damping the weights of the individual features, or effectively eliminating them (when $w_j \to 0$).
- If the model is nonlinear, regularization will reduce the nonlinearity.
- The weight w_0 should be excluded from regularization (because it defines the intercept) or the data should be centered so that $\overline{y} = 0$, because then $w_0 \to 0$.
- L2 regularization penalizes weights in proportion to their magnitude (large weights are penalized more and small weights are penalized less). This makes it difficult to push down the weights all the way down to zero, and this is why L2 regularization does not result in sparse models.
- L1 regularized regression results in sparse models, but it has no closed-form solution (one can, however, use iterative optimization methods).
- Regularization is useful for models with a lot of parameters, because such models overfit easy.
- Regularization reduces the possibility of overfitting and multicollinearity, but there remains the problem of choosing the value for the λ hyperparameter. As you can probably guess, this selection is most often done via **cross-validation**. What would we obtain as the optimal value for λ if we were to optimize it on the training set? The answer is that we would definitely choose $\lambda = 0$, because doing so would give us the most complex model and one that would have the smallest error on the training set. And this, of course, would not be good. So we have to optimize λ using cross-validation.

Summary

- The **Linear regression model** is linear in its parameters
- The nonlinearity of the regression function is achieved by mapping from the input space to a **feature space** using nonlinear **base functions**
- The parameters of a linear model with a quadratic loss function have a closed-form solution in the form of **the pseudoinverse of the design matrix**, regardless of what feature mapping we use
- **Regularization reduces overfitting** by extending the error function with an additional term that penalizes model complexity
- **L2 regularized regression** has a closed-form solution and effectively **reconditions** the design matrix, which is tantamount to **removing multicollinearity**