5. Linear Discriminative Models

Machine Learning 1, UNIZG FER, AY 2022/2023

Jan Šnajder, lectures, v2.3

Last week we covered regression and we talked about the **linear regression model**, which – with an adequate choice of a nonlinear feature mapping functions – actually becomes a nonlinear model. We saw that, regardless of whether the model is linear or nonlinear, the least squares method yields a closed-form solution, which boils down to the computation of the **pseudoinverse**. We also talked about **regularization**, as a means to prevent overfitting, and we have shown that for the L2 regularized linear regression model there also exists a closed-form solution.

Today we won't be dealing with regression anymore, instead we'll talk about **classification**. In fact, the rest of the course will mostly deal with classification. Recall that classification is a process of predicting discrete labels of individual examples: e.g., classification of emails into spam and non-spam, or classification of tweets in positive, negative, and neutral based on sentiment.

Of course, there are several approaches to classification. In the first couple of weeks we'll focus on **linear models**. More specifically, we will deal with the models from the group of **linear discriminative models**. As you will see, this is not a single model, bur rather a whole family of models, some of which are very efficient (for example, the logistic regression and the support vector machine).

Our goal today is to explain the basic idea of a linear discriminative model, then elaborate on the topic in the two weeks after that.

1 Linear discriminative models

Let's start by clarifying the title of today's lecture: what are **linear discriminative** models? Let's look into "linear" first. A model is linear if it models a boundary between classes that is linear: e.g., a line (if the input space is two-dimensional), a plane (if the input space is three-dimensional), or a hyperplane (for spaces with more than three dimensions).

How do we define a linear model? Well, let's start with the regression model:

$$h(\mathbf{x};\mathbf{w}) = \mathbf{w}^\mathrm{T}\mathbf{x}$$

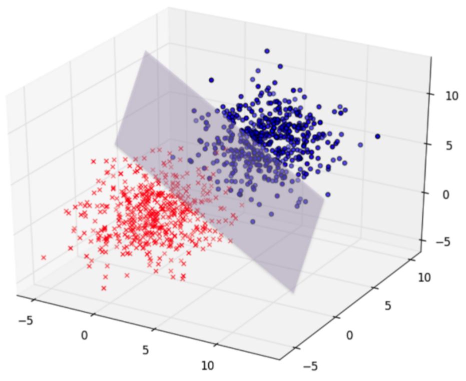
where again included the "dummy one" feature $x_0 = 1$, to be able to use the simpler vector notation. The hypothesis thus defined (i.e., a model with fixed weights $\mathbf{w}$) $h(\mathbf{x};\mathbf{w})$ gives some value from $\mathbb{R}$ for each input vector $\mathbf{x}$ from $\mathbb{R}^n$, i.e., $h(\mathbf{x};\mathbf{w}) : \mathbb{R}^n \to \mathbb{R}$.

Now, how can we conceive of this model as a **binary classification model**? We can do this by saying that this model actually divides the input space into two **half-spaces**: one comprising all points for which $h(\mathbf{x}) \geqslant 0$ and the other comprising all points for which $h(\mathbf{x}) < 0$.

The binary classifier may then be defined as follows:

$$h(\mathbf{x};\mathbf{w}) = \mathbf{1}\{\mathbf{w}^\mathrm{T}\mathbf{x} \geqslant 0\}$$

where $\mathbf{1}\{\cdot\} : \{\bot, \top\} \to \{0, 1\}$ is the indicator function we defined earlier. The boundary is exactly where $h(\mathbf{x}) = 0$ holds (and the points for which this hold also need to be assigned to one of the two half-spaces, it doesn't matter which one). For two-dimensional input space this boundary is a line, for a three-dimensional it is a plane, and in general it is an $(n-1)$-dimensional hyperplane embedded into an n-dimensional input space (the "dummy one" feature x_0 does not count towards the dimension). For example, for $n = 3$ it would look like this:

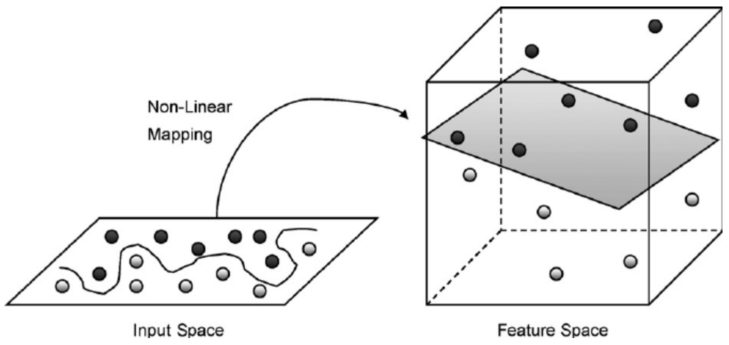


In general, then, the boundary is a **subspace** (hrv. *potprostor*) of dimension $n - 1$ (always one less than the dimension of the input space) which divides the space into two **half-spaces** (engl. *poluprostori*). We call this boundary the **discriminative function** (engl. *diskriminantna funkcija*) or a **decision boundary** or (hrv. *decizijska granica*), or we simply call it the **class boundary**.

But what about nonlinearity? We know that linear boundaries between classes are quite rare; most problems in machine learning are such that the boundary between classes is nonlinear. How can we obtain a nonlinear boundary? Well, if we want a nonlinear boundary, we can use the trick we learned last time: we can use a **nonlinear mapping function** to map our data from the input space to some other space (the feature space), where we will attack them using a linear model. So, by making use of the mapping function $\boldsymbol{\phi} : \mathbb{R}^n \to \mathbb{R}^{m+1}$ we obtain the following model:

$$h(\mathbf{x}; \mathbf{w}) = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})$$

This model still has a linear boundary in the feature space, but – if the mapping $\boldsymbol{\phi}$ is a nonlinear – the boundary in the input space will be nonlinear. Technically, however, it is still **linear model** because it is linear in parameters. Let's recall the example we discussed the last time:



Here we map from an input space of dimension $n = 2$ into a feature space of dimension $m = 3$. What has not been linearly separable in the input space has now become linearly separable in the feature space. The boundary between classes, which is linear in the feature space, is nonlinear in the input space.

▸ EXAMPLE

The well-known **XOR-problem** (the "exclusive or" problem) is a simple example of how mapping into a feature space of a dimension that is higher than the dimension of the input space can make a problem that is nonlinear in the input space linear in the feature space. This is a binary classification problem defined in the two-dimensional input space $\mathcal{X} = \{-1,+1\}^2$ with the following set of labeled examples:

$$\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^N = \{((-1,-1),0),((+1,+1),0),((-1,+1),1),((+1,-1),1)\}$$

We cannot solve this problem with a linear model. That is, if $\mathcal{H}$ is a linear model (a set of lines), then $\neg\exists h \in H.E(h|\mathcal{D}) = 0$. However, if we map the examples from a two-dimensional input space to a three-dimensional feature space using the following mapping function:

$$\boldsymbol{\phi}(\mathbf{x}) = (1, x_1, x_2, x_1x_2)$$

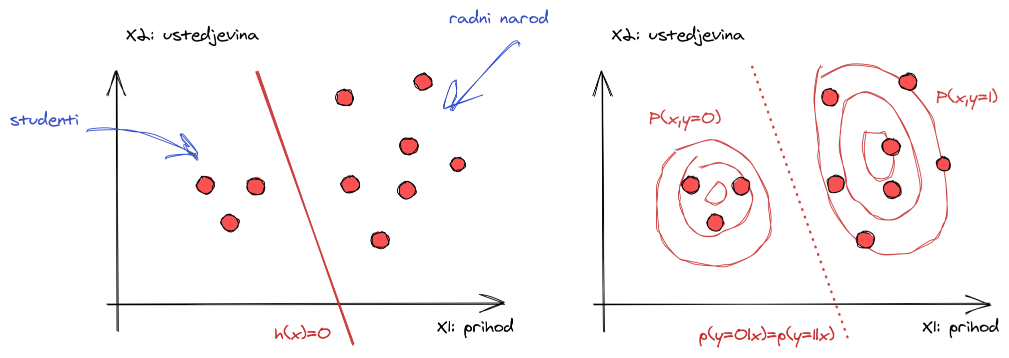
then the examples will be separable in the feature (there exists a plane such that the examples of class $y = 0$ are above that plane, and the examples of class $y = 1$ are below it).

So much for the first part of the title: a "linear model ". Let's now look into what "discriminative" means. **Discriminative models** are models that represent the boundary that discriminates (differentiates) between classes, so that the parameters of the model describe the boundary and nothing more than the boundary. In other words, a discriminative model will have as many parameters as needed to define that boundary.

This may seem obvious, but it becomes more interesting if we consider what the alternative is. Discriminative models stand in opposition to another large family of models, called **generative models**. Generative models model than just the boundary between classes, and in fact they model the boundary rather indirectly. They also tend to have much more parameters than discriminative models. An example will come in handy.

▸ EXAMPLE

We want to classify the creditworthiness of bank customers: the bank needs to decide to whom to grant a credit. As we've already done in the introductory lecture, let's simplify the problem and assume the banks are doing this based on just two features: income and savings. In this case the input space is two-dimensional (x1 → income, x2 → savings), and each bank customer is one two-dimensional vector in that space:



The figure on the left shows the boundary between classes as found by a discriminative model. The boundary will be described by the parameters of that model. The figure on the right shows a generative model. The generative model models the probability densities of each class, and the figure shows the isocountours of the probability density function. One can now compute the class boundary indirectly based on these distributions as all the points for which the densities for the two classes are the same (this boundary is shown as a dotted line; note that it is again linear). That's more information than provided by the discriminative model, but it may be information we don't really need.

We leave a more detailed treatment of generative models for the second part of the course. Today, we focus on linear discriminative models. Now that we have explained the meaning of the terms "linear" and "discriminative", let's take a closer look at how the parameters of a linear discriminant model define the boundary in an input space, which usually is a Euclidean vector space, i.e., let's look at the "geometry" of a linear model.

2 Geometry of the linear model

We have already said that linear discriminative models use **hyperplanes** as class boundaries. For the sake of simplicity, let's first focus on the case of two classes, $K = 2$, to be extended later to multiple cases. Also, with no loss of generality, let's focus on the simplest model without any feature mapping, that is:

$$h(\mathbf{x}; \mathbf{w}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$$

In fact, the derivations will be easier if we pull out the weight w_0 from $\mathbf{w}$ and treat it separately, so that we have:

$$h(\mathbf{x}; w_0, \mathbf{w}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0$$

The boundary between classes is given by the equation $h(\mathbf{x}) = 0$, which defines an $(n-1)$-dimensional hyperplane within an n-dimensional input space. Again with no loss of generality, in the following we will make our lives easier and sketch the case of a two-dimensional input space, $n = 2$, in which the boundary between classes is a line. So, the model is then:

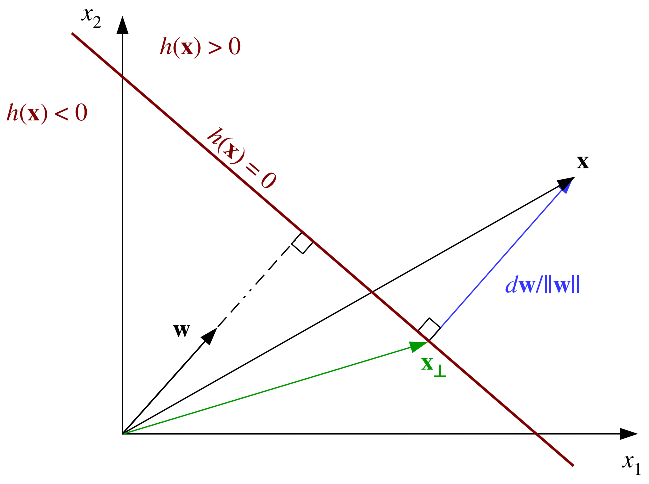
$$h(\mathbf{x}; w_0, \mathbf{w}) = w_1x_1 + w_2x_2 + w_0$$

and the class boundary is:

$$w_1x_1 + w_2x_2 + w_0 = 0$$

which we recognize as the implicit equation for a line. However, below we will continue to speak of a "hyperplane", as our conclusions will be more general.

Our further consideration be based on the following figure:



The figure shows a line (shown in red) as the boundary in a two-dimensional input space $\mathbb{R}^2$ with features x_1 and x_2. That line corresponds to the equation $h(\mathbf{x}; w_0, \mathbf{w}) = 0$, i.e., the boundary is made up of points (x_1, x_2) for which the hypothesis h gives a value of zero. The line divides the two-dimensional space into two half-spaces: one half-space is made up of all points for which $h(\mathbf{x}; w_0, \mathbf{w}) > 0$ and the other of all points for which $h(\mathbf{x}; w_0, \mathbf{w}) < 0$. The normal of the line is the vector $\mathbf{w}$, which is perpendicular to the line, and points in the direction of the half-space for which $h(\mathbf{x}; w_0, \mathbf{w}) > 0$ (which, if we define the binary classifier as $h(\mathbf{x}; w_0, \mathbf{w}) = \mathbf{1}\{\mathbf{w}^{\mathrm{T}}\mathbf{x} \geqslant 0\}$, is the half-space of positive examples). Here the normal vector $\mathbf{w}$ is shown as a vector with the initial point at the origin of the coordinate system (i.e., as a "position vector"), but its initial point is, of course, arbitrary. The figure also shows the vector of an example $\mathbf{x}$, which lies somewhere in the half-space of positive examples. Remember that every example $\mathbf{x}$ is actually a vector, so we can depict it as a position vector (a vector with the initial point at the origin of the coordinate system). The figure further shows the decomposition of vector $\mathbf{x}$ into the sum of two vectors: $d\frac{\mathbf{w}}{\|\mathbf{w}\|}$ (shown in blue) and $\mathbf{x}_\perp$ (shown in green); more on that below.

The main thing that interests us here is which side of the hyperplane an example is lying on and at what distance. It should come at no surprise that we are interested in what side the example is lying on, because this determines how the example is classified. But why do we care about the distance? We are interested in the distance because the distance is indicative of classification **confidence**: how certain we can be that the example is positive or negative. For example, when the example is far from the hyperplane on its positive side, then we can be more certain that the example is positive than if it were very close to the hyperplane.

So then, aiming to determine the side and the distance of an example, we will consider two things: (1) what is the weight vector $\mathbf{w}$ (without w_0) actually corresponding to and (2) how to calculate the distance of an example from the hyperplane.

First, what is the weight vector $\mathbf{w}$ actually corresponding two? Consider two examples, i.e., two points $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$, that lie on the hyperplane. The following holds:

$$h(\mathbf{x}^{(1)}) = h(\mathbf{x}^{(2)}) = 0$$

If we plug in the model definition for $h(\mathbf{x})$, we get:

$$\begin{aligned}
\mathbf{w}^\mathrm{T}\mathbf{x}^{(1)} + w_0 &= \mathbf{w}^\mathrm{T}\mathbf{x}^{(2)} + w_0 \\
\mathbf{w}^\mathrm{T}(\mathbf{x}^{(1)} - \mathbf{x}^{(2)}) + w_0 - w_0 &= 0 \\
\mathbf{w}^\mathrm{T}(\mathbf{x}^{(1)} - \mathbf{x}^{(2)}) &= 0
\end{aligned}$$

Note that $(\mathbf{x}^{(1)} - \mathbf{x}^{(2)})$ is the difference of two vectors, which is again a vector, and that this vector lies exactly on the hyperplane. Since we have just derived $\mathbf{w}^\mathrm{T}(\mathbf{x}^{(1)} - \mathbf{x}^{(2)}) = 0$, and knowing that the scalar product of the two vectors vanishes when these vectors are **perpendicular** to each other, this implies that the vector $\mathbf{w}$ is perpendicular to all vectors lying on the hyperplane, so we conclude that $\mathbf{w}$ is the **hyperplane's normal vector**. In our figure above, the $\mathbf{w}$ is a normal of a line, as we are dealing with a two-dimensional input space.

As we said, another thing that interests is the **distance** of an example from the hyperplane. Let us therefore consider a point $\mathbf{x}$, which lies at some distance from the hyperplane. This point has a projection on the hyperplane – let us denote that point by $\mathbf{x}_\perp$. The vector $\mathbf{x}$ can now be decomposed into the sum of two vectors:

$$\mathbf{x} = \mathbf{x}_\perp + d\frac{\mathbf{w}}{\|\mathbf{w}\|}$$

This second vector is the unit vector of the normal vector, $\mathbf{w}/\|\mathbf{w}\|$, multiplied by d. Hence, d is actually the distance of the point $\mathbf{x}$ from the hyperplane, and that's what we are after.

And now for a bit of algebraic magic. Let's multiply both sides of the equation by $\mathbf{w}^T$ and add w_0 to both sides:

$$\begin{aligned}
\underbrace{\mathbf{w}^\mathrm{T}\mathbf{x} + w_0}_{h(\mathbf{x})} &= \underbrace{\mathbf{w}^\mathrm{T}\mathbf{x}_\perp + w_0}_{=h(\mathbf{x}_\perp)=0} + d\frac{\mathbf{w}^\mathrm{T}\mathbf{w}}{\|\mathbf{w}\|} \\
h(\mathbf{x}) &= d\frac{\mathbf{w}^\mathrm{T}\mathbf{w}}{\|\mathbf{w}\|} = d\|\mathbf{w}\|
\end{aligned}$$

where we used $\mathbf{w}^{\mathrm{T}}\mathbf{w} = \|\mathbf{w}\|^2$. From this we get that the distance d of an example $\mathbf{x}$ from the hyperplane with the normal vector $\mathbf{w}$ is equal to:

$$d = \frac{h(\mathbf{x})}{\|\mathbf{w}\|}$$

Note that this distance can be positive or negative, which is why we will call it **signed distance** (hrv. *predznačena udaljenost*). Specifically, we have:

- $d > 0 \ \Rightarrow\ \mathbf{x}$ is on the side of the hyperplane in the direction pointed to by the normal $\mathbf{w}$
- $d < 0 \ \Rightarrow\ \mathbf{x}$ is on the opposite side of the hyperplane
- $d = 0 \ \Rightarrow\ \mathbf{x}$ is exactly on the hyperplane

All these considerations of ours also apply to spaces of dimension greater than two, $n > 2$. Thus, the vector $\mathbf{w}$ (without w_0) is the normal of an $(n-1)$-dimensional plane in space $\mathbb{R}^n$, and it points in the direction of the half-space for which $h(\mathbf{x}) > 0$. The signed distance of example $\mathbf{w}$ from the hyperplane $h(\mathbf{x}) = 0$ is equal to $h(\mathbf{x})/\|\mathbf{w}\|$. It can also be shown, although we'll skip this, that the distance of the hyperplane from the origin is equal to $-w_0/\|\mathbf{w}\|$.

We should also notice at this point that one and the same hyperplane can be defined with different weights $\mathbf{w}$: more precisely, there are infinitely many weight vectors $(w_0, \mathbf{w})$ that define an identical hyperplane. This is because if we multiply the vector $(w_0, \mathbf{w})$ by some factor α, then the vector of normal $\mathbf{w}$ will be α-times bigger, but its direction will not change. Also, because the weight w_0 will be α times larger, neither the distance of the hyperplane from the origin nor the distance of the example from the hyperplane will change. This insight is of no particular use to us now, but it will be in two weeks, when we talk about the SVM model.

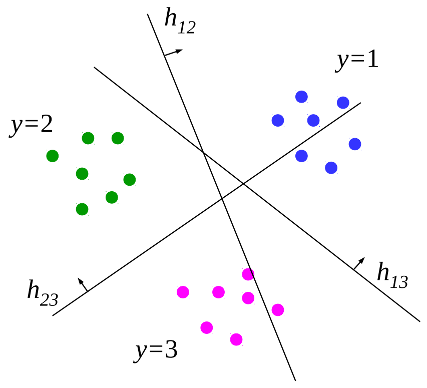
3 Multiclass classification

So far, we have dealt with binary classification, i.e., classification in two classes. But not all classification problems we'll encounter will be binary. Sometimes we need to classify the examples into more than two classes. For example, classification of posts from Twitter into positive, negative, or neutral. Or classification of newspaper articles into sections: sports, culture, crime reports, politics, etc.

The problem with multiclass classification is that many linear discriminative models are inherently **binary classifiers**. This rises the question of how we can leverage such models when we have $K > 2$ classes? We would very much like not to mess up with the model but instead to change the problem. Remember that we pulled a similar trick earlier when we mapped the examples to the feature space, so that we can apply a linear model but have a nonlinear boundary. We are now about to do something similar: we will decompose a multiclass classification problem into a set of binary classification problems, and then simply apply the binary classifier multiple times. In particular, we have two options for how to do this: a one-vs-one scheme and a one-vs-rest scheme. Let's look at them one after the other.

The **one-versus-one (OVO)** scheme (hrv. *jedan-naspram-jedan*) reduces a multiclass problem to $\binom{K}{2}$ independent binary classification problems, one for each pair of classes. We train a model h_{ij} so that it learns to separate the examples of class $y = i$ from the examples of class $y = j$, thereby ignoring the examples from all other classes.

For example, for $K = 3$ classes in a two-dimensional input space, this would look like this:



Since here there are three classes, in the OVO scheme we need $\binom{3}{2} = 3$ binary classifiers. The hypotheses corresponding to these three classifiers are denoted as h_{12}, h_{23}, and h_{13}, where h_{ij} separates the examples of the class labeled $y = i$ from the examples of classes labeled $y = j$. In the figure, the arrows point in the direction of the positive orientation of the hyperplane, i.e., the direction where h_{ij} classifies as positive the examples from the class $y = i$. We see that each of the three hypotheses always separates only two classes, while completely ignoring the third class. Thus, for example, the plane corresponding to hypothesis $h_{23}(\mathbf{x}) = 0$ separates classes $y = 2$ and $y = 3$, with an orientation towards examples from class $y = 2$, while passing through examples from class $y = 1$, which it ignores. We achieve this by training each of these binary classifiers h_{ij} on a subset of labeled examples $\mathcal{D}$ that contains only those examples that are from classes $y = i$ and $y = j$.

The decision into which class to classify the example in is now made by a majority vote. In other words, we define a **multiclass OVO model** as follows:

$$h(\mathbf{x}) = \underset{i}{\operatorname{argmax}} \sum_{i \neq j} \operatorname{sgn}\big(h_{ij}(\mathbf{x})\big)$$

where

$$h_{ij}(\mathbf{x}) = -h_{ji}(\mathbf{x})$$

Convince yourself that this definition really corresponds to taking the majority decision of the binary classifiers. Let's look at an example.

▸ EXAMPLE

Look at the figure above and consider the classification of an example $\mathbf{x}$ from class $y = 2$. For this example we have:

$$\begin{aligned} h_{12}(\mathbf{x}) &= -1 &\Rightarrow\quad h_{21}(\mathbf{x}) &= 1 \\ h_{13}(\mathbf{x}) &= -1 &\Rightarrow\quad h_{31}(\mathbf{x}) &= 1 \\ h_{23}(\mathbf{x}) &= 1 &\Rightarrow\quad h_{32}(\mathbf{x}) &= -1 \end{aligned}$$

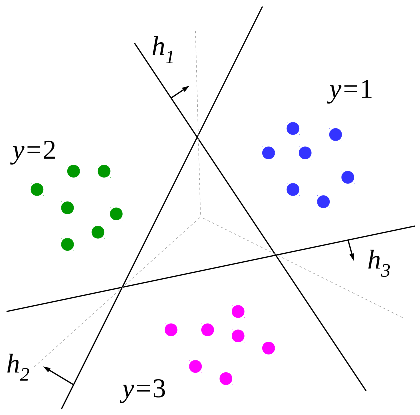
The voices accumulated for the each of the three classes are as follows:

$$\begin{aligned} (i=1):&\quad \operatorname{sgn}(h_{12}(\mathbf{x})) + \operatorname{sgn}(h_{13}(\mathbf{x})) = (-1) + (-1) = -2 \\ (i=2):&\quad \operatorname{sgn}(h_{21}(\mathbf{x})) + \operatorname{sgn}(h_{23}(\mathbf{x})) = 1 + 1 = 2 \\ (i=3):&\quad \operatorname{sgn}(h_{31}(\mathbf{x})) + \operatorname{sgn}(h_{32}(\mathbf{x})) = 1 + -1 = 0 \end{aligned}$$

so the class labeled $y = 2$ wins the majority vote, which is consistent with the true label of example $\mathbf{x}$.

The OVO scheme is simple, but we one can notice that there may be situations when the classification decision will be ambiguous. This will be the case with every example that falls into the portion of the input space (or feature space, if we use mapping) where there are ties in the number of votes. In the figure above, this is the triangular region in the middle of the input space. In practice, however, the chances for a perfect tie are relatively slim. Even if does occur, one can resolve the tie by taking into account the confidence of classification decisions (based on the signed distance of the example from the hyperplane, as we explained above).

The other multiclass classification scheme that decomposes the problem into binary classifications **one-versus-rest (OVR)** scheme (hrv. *jedan-naspram-ostali*) (sometimes incorrectly called *one-vs-all*). This scheme uses K independent binary classifiers, one for each class. Each classifier h_i is trained to separate the examples of class $y = i$ from examples of all other classes $y \neq i$, i.e., the classifier is trained to recognize whether the example comes from class $y = i$ or not. This is how the above example would look like, when cast in the OVR scheme:



Since we have $K = 3$ classes, we need 3 binary classifiers for the OVR scheme. The hypotheses corresponding to these three classifiers are h_1, h_2, and h_3, where hypothesis h_i separates examples of a class labeled $y = i$ from examples of all other classes.

The multiclass model in the OVR scheme makes a decision based on the confidences of the individual classifiers, assigning the example to the most confident class. This is easily defined as follows:

$$h(\mathbf{x}) = \underset{j}{\operatorname{argmax}}\ h_j(\mathbf{x})$$

hence the example is classified in the class corresponding to the binary classifier that is most confident in its decisions. A model defined in this way actually implicitly defines a boundary between two adjacent classes $y = i$ and $y = j$ at the points where $h_i(\mathbf{x}) = h_j(\mathbf{x})$ holds. In our figure, such boundaries correspond to the bisectors of the angles (shown as dashed lines).

Note that with the OVR scheme, unlike with the OVO scheme, there will be much fewer regions where the classification decision is ambiguous (admittedly, an example may fall right at the point where confidences of two or more most confident classifiers are tied, but this just means you're cursed and you better stay away from machine learning and all who practice it).

Let us now compare these two schemes. An obvious advantage of OVR over OVO is that there are fewer models to train: K versus $\binom{K}{2}$, which is a linear versus quadratic dependence on the number of classes. In this example of ours, we only had three classes so the difference didn't come to the fore. But the difference will be substantial if we have a lot of classes and a whole lot of examples, because then training a large number of models will take a very long time, and also the time it takes to make a prediction may not be negligible. For example, if we have ten classes (e.g., for the digit recognition problem), we will need 45 classifiers for the OVO scheme, whereas for the OVR scheme we will need only ten.

On the other hand, the problem with the OVR scheme is that it easily results in an **unbalanced** number of examples between a pair of classes for which we train a model. Why? Because for each model h_j there will in principle be many more negative examples (all those that do not belong to class j) than positive examples. This is already evident in our example. The three classes each have seven examples (see previous figure). If we train a binary classifier to separate examples of one class from the examples of the other two classes, then each such classifier will be trained on 7 positive and 14 negative examples. The situation becomes worse with the number of classes. For ten classes, the ratio of positive and negative examples will be 1:9. In general, if the ratios of the K classes are balanced in $\mathcal{D}$, i.e., if we have an equal number of examples for each class, the ratio of positive versus negative examples for the individual binary classification problems in the OVR scheme will be $1 : (K - 1)$. (If initial ratios are not balanced, then the ratio for some binary problems will be more favorable, only to be worse for some others.) This is problematic because linear discriminative models are are bad in dealing with situations where there are many more examples from one class than examples from another class. What typically happens in such situations is that the optimization algorithm, in the legitimate effort to reduce the empirical error, simply does so at the expense of examples from the smaller class, by assigning all these examples to the majority class. This **class imbalance problem** (hrv. *problem neuravnoteženosti klasa*) is a well-known and a well-studied problem in machine learning. The problem is very common in practice and some remedies have been proposed, but we will not talk about them here.

Thus, the choice between OVO and OVR multiclass schemes in practice boils down to a compromise between the number of classifiers on the one hand and class imbalances on the other. If the classes are not too many and if time complexity isn't too much of an issue, OVO is probably a better option. better option.

4 Classification using regression

So far we have not yet introduced any classification algorithm, so it's high time we finally do it. Actually, instead of introducing a new algorithm, we'll try to take advantage of what we already know. Namely, last week we talked about the **linear regression model**. A reasonable question is whether we can somehow use that algorithm for classification. It will turn out that we can't, but let's try anyway, because we'll learn an important lesson.

4.1 The naive approach

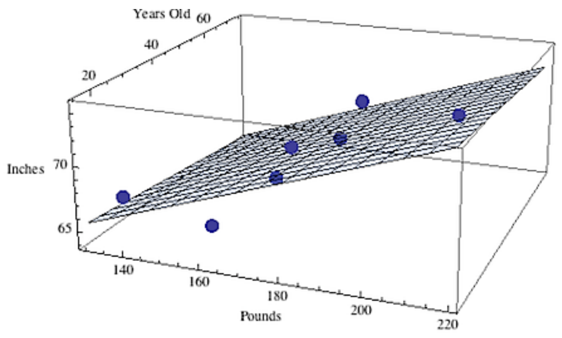
Recall, we have this for the error function (empirical expectation of the quadratic loss) of a linear regression model:

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

The error minimizer is:

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

As an example, consider a regression with two features ($n = 2$). So, the input space is two-dimensional and the regression function is plane:



This plane assigns one number to each example $(x_1, x_2) \in \mathbb{R}^2$, ranging from minus infinity to plus infinity. The question is: how could we use this for classification? Well, the simplest thing we can do is to treat classification as regression: to assign numbers from $\{0, 1\}$ to the examples from the two classes, and to train the regression model on such a set. In other words, we want to learn the regression model $h(\mathbf{x})$ that will output $h(\mathbf{x}) = 1$ for examples from class $y = 1$, and $h(\mathbf{x}) = 0$ for examples from class $y = 0$. Then, when doing prediction, we simply look at whether $h(\mathbf{x})$ is greater than or less than 0.5. In the former case, the example is classified into the positive class, otherwise into the negative class. That is:

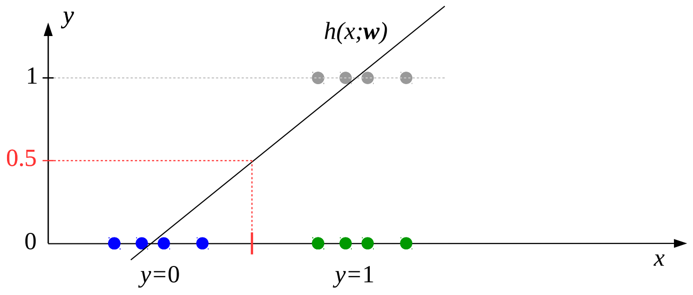
$$h(\mathbf{x}; \mathbf{w}) = \mathbf{1}\{\mathbf{w}^{\mathrm{T}}\mathbf{x} \geqslant 0.5\}$$

Thus, the boundary between the classes $y = 1$ and $y = 0$ is formed by the line defined by the equation $h(\mathbf{x}) = 0.5$. Alternatively, we can train the model so that for examples from the negative class the target value is $y = -1$. The boundary between the classes will then be defined as $h(\mathbf{x}) = 0$.

Let's look at an example. .

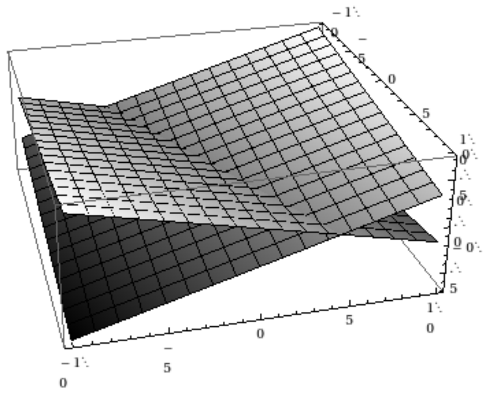
▸ EXAMPLE

The simplest example is for a one-dimensional input space ($n = 1$). It's an unrealistic example, but one that will serve to explain the idea. Let's say we have 4 examples coming from the positive class ($y = 1$) and 4 examples coming from the negative class ($y = 0$), and that one can separated the two classes in the one-dimensional input space (i.e, on the x-axis). We train a regression model to predict $y = 1$ for examples from class $y = 1$ and to predict $y = 0$ for examples from class $y = 0$. This is how this looks like:

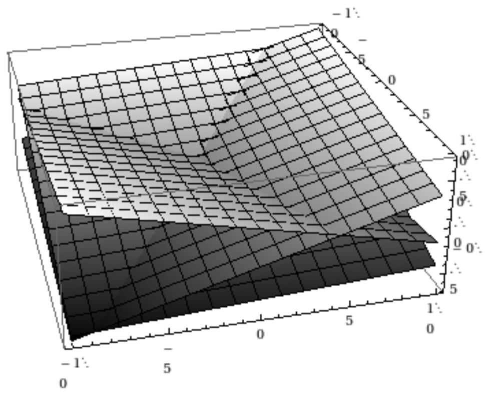


We obtain a regression line $h(x; \mathbf{w})$ that, as always, minimizes the sum of the squared differences between the predicted values and the target values. A point on the x axis for which $h(x; \mathbf{w}) = 0.5$ represents the boundary between the classes in the one-dimensional input space. For the examples to the left of that point, we have $h(x; \mathbf{w}) < 0.5$, so we classify them as negatives, while for the examples to the right of that point, we have $h(x; \mathbf{w}) \geqslant 0.5$), so we classify these examples as positives. So far this looks like it might work, but we're about to see there's a problem. (Maybe you already see it?)

Instead of training one model for the two classes, we can train two models, one for each class, and in this case the class boundary would consist of points for which $h_i(\mathbf{x}) = h_j(\mathbf{x})$. For example, in a two-dimensional input space:



If, on the other hand, we want to classify into more than two classes, we can, for example, apply the OVR scheme, and train one model for each of the K classes. The boundary between adjacent classes will consist of the points for which $h_i(\mathbf{x}) = h_j(\mathbf{x})$, where h_i and h_j are two hypotheses with the largest values for $\mathbf{x}$. For example, for $n = 2$ and $K = 3$:



4.2 Problems

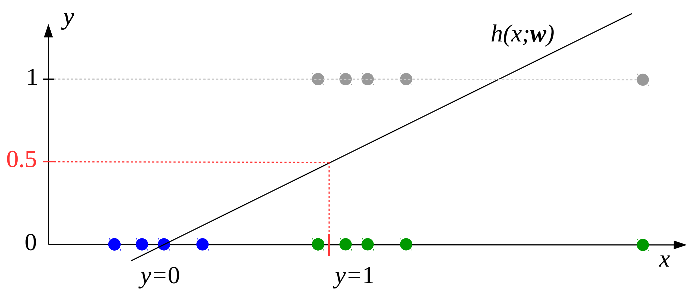
At first glance, this looks great! We seem to have managed to solve the classification problem using a linear regression model. The advantages of this approach are its simplicity and the existence of a closed-form solution. But the look is deceiving. There are shortcomings, some quite grave, of using a linear regression model for classification. Namely:

1. The model outputs have no probabilistic interpretation, since the domain of hypothesis $h(\mathbf{x})$ is not limited to the $[0, 1]$ interval;
2. The model is not robust, that is, it is very sensitive to outliers. In particular, the algorithm penalizes examples that are classified "too correctly" and therefore, in some cases, misclassifies examples even when they are linearly separable.

Let's take a closer look at what we mean when we say the model is not robust.

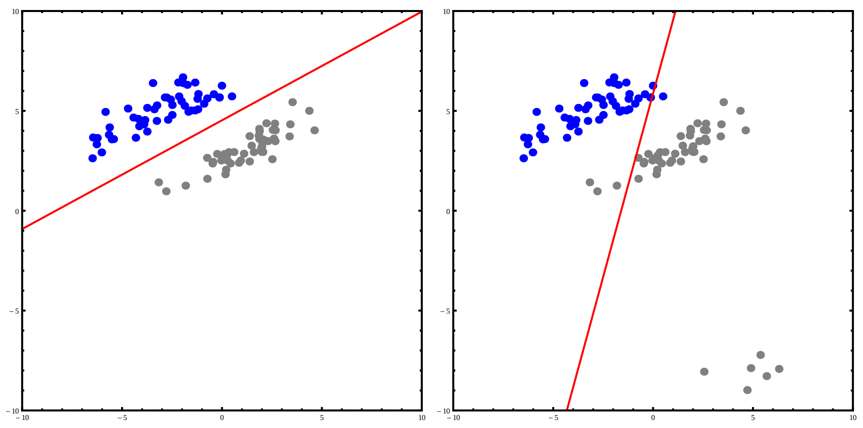
▸ **EXAMPLE**

Consider the same situation as in the previous example, i.e., classification in a one-dimensional input space, but the addition of one positive example ($y = 1$). Let this example be "deep" in the region of positive examples, so, on the very right of the x axis. Like this:



How will this affect the regression line? Since the regression algorithm minimizes the squared differences, adding a new example will make the line pass closer to that example, because otherwise the square of the residual for that example would become very large. The $h(x;\mathbf{w})$ line shown in the figure is approximately the line that minimizes the squared differences (actually, the line should be slightly steeper, but you get the point). Because the line's slope has changed after adding a new example, so did the value for which $h(x;\mathbf{w}) = 0.5$, which determines the boundary between the classes. We see that the boundary between classes has shifted to the right, and now one positive example is being misclassified (a false negative). So, what happened is that the addition of a single example, which, mind you, already was on the correct side of the boundary, led to the boundary moving closer to that example and the misclassification of another example which was close to the boundary.

The same problem occurs when the input space is of a larger dimension. For example, for $n = 2$:



The figure on the left shows a two-dimensional input space with examples from two classes (blue and gray dots) and the boundary between them obtained as $h(\mathbf{x};\mathbf{w}) = 0.5$ (red line). This boundary was obtained by training a regression model that assigns a value of 1 to examples from one class (e.g., the blue examples) and a value of 0 to examples from the other call (e.g., the gray examples). The hypothesis corresponds to a plane in a three-dimensional space $\mathbb{R}^2 \times \mathbb{R}$, and the place where that plane intersects the $x_1 \times x_2$ plane is precisely the line defined by the equation $h(\mathbf{x};\mathbf{w}) = 0.5$. In the figure on the right, we see what happens when we add to the training set a couple of examples that are far from the boundary (the gray-colored examples at the bottom right). Since the model is now trained so that these examples also need to be assigned a value of 0, the least squares optimization procedure will find a new plane that minimizes the empirical error. As can be seen in the figure, the boundary we get as $h(\mathbf{x};\mathbf{w} = 0.5)$ is now very much tilted toward the newly added examples, which makes the model misclassify some examples from the positive class as well as the negative class. Similarly to the previous example, although the problem is still linearly separable, examples that are far from the boundary have a large impact on its position and reduce the accuracy of the classifier.

The problem we just identified – that classification doesn't work well if some examples are far from the boundary – reveals that the linear regression model is not a good classifier. If the examples are linearly separable in input space, we expect the machine learning algorithm to give us a hypothesis that perfectly accurately classifies all the examples. However, with linear regression this is clearly not the case. This is a big problem and, if classification is what we want, we have to solve it somehow.

4.3 Who's to blame?

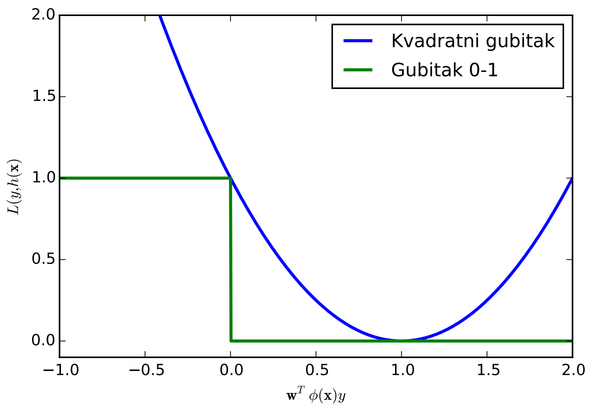
What exactly is the problem with classification using regression? Let us recall that each machine learning algorithm consists of three components: a model, a loss function (from which we derive the empirical error function), and an optimization algorithm. If there is a problem, it lies in one of these three components. Which one? The optimization algorithm is definitely clear of guilt, because that algorithm is merely doing its job with the model and the loss function we give it.

The problem, then, has to be in the model or the loss function. In fact, we can say that the problem is *either* in the model *or* in the loss function. Namely, on the one hand the problem is in the model because for the model gives a very high output for examples that are far from the boundary, which, in turn, leads to large values of squared differences. On the other hand, the problem is in the loss function, because the loss function is defined as the squared difference, which penalizes even the correctly classified examples.

We will concentrate below on the loss function as the culprit for this problem. Our goal is to show why the loss function we use in the regression algorithm is not good for classification. Recall that regression uses a **quadratic loss function**:

$$L(y, h(\mathbf{x})) = \big(y - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})\big)^2$$

The loss function L is of type $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_0^+$, i.e., it is a function of two variables. It will, however, be somewhat easier if we analyze and plot this function if we somehow redefine it as a function of one variable. Fortunately, we can easily be done. First, let's switch from the label set $y \in \{0, 1\}$ to the label set $y \in \{-1, +1\}$. Next, note that if an example $(\mathbf{x}, y)$ is classified correctly, then the the sign of the scalar product $\mathbf{w}^{\mathrm{T}}\mathbf{x}$ will be equal to the sign of the y label, regardless of whether the example is positive or negative. This means that the product $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) \cdot y$ is always positive for correctly classified examples and negative for incorrectly classified examples (the case $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})y = 0$ can be included in the correct classification, if $h(\mathbf{x}) = \mathbf{1}\{\mathbf{w}^{\mathrm{T}}\mathbf{x} \geqslant 0\}$). The idea is then to plot the quadratic loss function L as a function of one argument, $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})y$. Here's what such a graph would look like for **loss 0-1** and for the **quadratic loss**:



This graph is important, so let's explain it in detail. On the x-axis of the graph represents the value $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})y$, which, as we have just established, will be positive for correct classification and negative for incorrect classification. This value can be understood as a "measure of classification accuracy". Thus, the positive part of the x-axis corresponds to cases when the classification is correct, and the negative part of the x-axis corresponds to cases when the classification is incorrect, and the farther to the right or to the left we are, the more accurate or incorrect the classification, respectively. The y-axis is for the value of the loss function L. The 0-1 loss function is shown in green. As can be seen from the graph, this function has a value of 1 if classification is incorrect ($\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})y < 0$) and a value of 0 if classification is correct ($\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})y \geqslant 0$). Let us now look at what the graph of the quadratic loss function looks like. The loss will be zero when the predicted value is equal to y, will will be the case when $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})y = 1$. When the predicted value is zero, the the loss equal to 1. Also, when the predicted value is equal to 2 or -2, the loss will again be 1. So we have a parabola, which is show in blue in the graph.

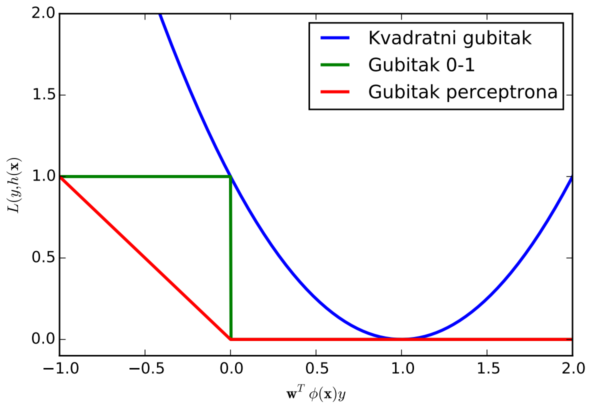
Let's see what we got here. We see that the quadratic loss is equal to zero only for examples for which $\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x})y = 1$, i.e., only for those examples for which the output of the hypothesis is equal to exactly 1 (for positive examples) or exactly -1 (for negative examples). In all other cases the quadratic loss is larger than zero. Misclassified examples (the negative part of the x-axis) inflict a loss that increases quadratically with the distance of the example from the boundary. However, we see that the quadratic loss also **penalizes the correctly classified examples** (the positive part of the x-axis), especially examples which are very, very correctly classified: examples that lie deep in the region of the correct class will be severely penalized. It is, in fact, what causes the non-robustness of the solution: if there exists even a single example that is deep in the correct region (on either side), the loss will be huge and the optimization algorithm will try to reduce it by moving the hyperplane accordingly.

Obviously, if we want to fix this – and we want to – we need a different loss function. But, what loss function would do? Ideally, we want to penalize only incorrect classifications, and we want that penalty to be the same for all incorrect classification, no matter how incorrect. In other words, we want a **loss 0-1** (shown in green in the above graph).

Of course, the problem is that we can't just change the loss function and do linear regression again, as that would no longer be linear regression. That is, the quadratic loss function is one of the components of the regression algorithm. If we were to replace it with some other loss function, then the optimization procedure would have to be changed as well (because the least squares procedure works only with quadratic loss), and that would no longer be the linear regression algorithm. So, in order to do classification, we will after all need a different algorithm. We can't just like that use regression for classification!

The idea that immediately comes to mind is to devise a classification algorithm that uses the 0-1 loss as its loss function. That's a good idea, but unfortunately it won't work. That is to say, the error function defined as the expectation of the 0-1 loss function cannot be used for optimization. Why? For two reasons. First, this function is not differentiable, thus it will not yield a closed-form solution. Second, if we try to optimize in some other way, e.g., using gradient descent, the problem with this function is that it is mostly constant (except at zero), so there is no slope to descend on.

But here is an alternative idea: instead of using the 0-1 loss, why not use a function that is as similar to it as possible, but is differentiable and does have a slope? For example, we can define the loss function so that the loss is zero for examples that are classified correctly, while for the for the incorrectly classified examples it is proportional to how inaccurate the classification actually is. In our graph of the loss functions, such a loss function would look like this (red line):



Thus, the loss function on the positive part of the x-axis is zero, while on the negative part of the x-axis it increases linearly. That part of the function, when the examples are classified incorrectly, has a non-zero derivative, which means that we could use it for optimization, e.g., by an iterative method, specifically, e.g., the **gradient descent**.

It turns out there is already a machine learning algorithm that has just this kind of loss. This is the **perceptron** algorithm, which many of you are probably already familiar with. Let's look at this algorithm in a bit more detail, focusing specifically on the loss function.

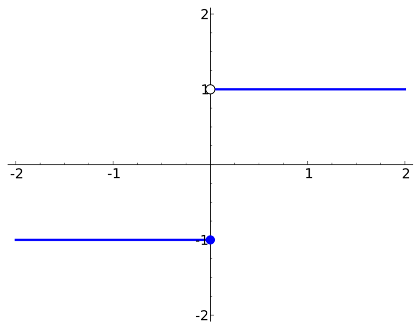
5 Perceptron

The perceptron algorithm is one of the earliest machine learning algorithm and the first hope of the connectionist (neural) approach to artificial intelligence. Basically, it is an algorithm for binary classification. Let's start with its model. The main difference with respect to the linear regression model is that the model outputs are restricted to the values -1 and $+1$. This is accomplished by wrapping the scalar product of the weight vector and the feature vector into a suitably defined function:

$$h(\mathbf{x};\mathbf{w}) = f\big(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})\big)$$

where f is defined as a **step function**:

$$f(\alpha) = \begin{cases} +1 & \text{if } \alpha \geqslant 0 \\ -1 & \text{otherwise} \end{cases}$$



so the boundary between the classes $y = 1$ and $y = -1$ is a line defined by $h(\mathbf{x};\mathbf{w}) = 0$. In this context (and, more generally, in the context of neural networks) the f function is referred to as the **activation function** (hrv. *aktivacijska funkcija*) or **transfer function** (hrv. *prijenosna funkcija*).

This, then, is the perceptron's model. Let us now look at its loss function. We have already shown the perceptron loss function in our loss functions graph (see above, red line). We can define this function as follows:

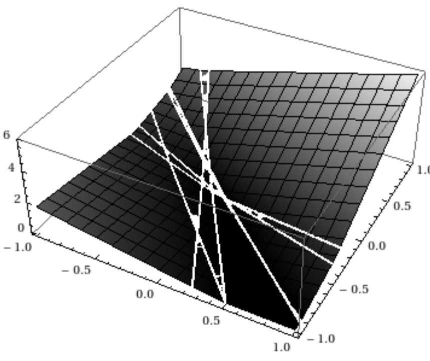
$$\max\left(0, -\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})y\right)$$

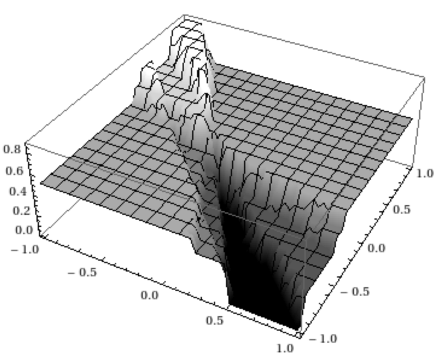
Note that we use the expression $\max(0, \cdot)$ to lower bound the value of the loss function at zero, as we do not want the loss to be negative (a negative loss would mean a reward for the hypothesis, and in supervised machine learning we don't want to reward anyone). The error function is the expectation of the loss function, that is, the empirical error function is the average value of the loss function on the training set, but here we will ignore the $1/N$ term (Why? Because we can!), so we have:

$$E(\mathbf{w}|\mathcal{D}) = \sum_{i=1}^{N} \max\left(0, -\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}^{(i)})y^{(i)}\right) = -\sum_{i\,:\,f(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}^{(i)}))\neq y^{(i)}} \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}^{(i)})y^{(i)}$$

The first expression is simply the sum of the losses over all examples from $\mathcal{D}$. The second expression is an alternative way to define the same, but without the max function: we sum only over examples that are classified incorrectly. At any rate, we penalize only incorrectly classified examples, and we do this in proportion to the amount of misclassification.

What would be interesting now is to see what the error function $E(\mathbf{w}|\mathcal{D})$ looks like in the parameter space $\mathbf{w}$ (i.e. in the space of the weights). For a two-dimensional parameter space ($\mathbf{w}_1 \times w_2$), the error function looks like this (left figure):





We see that the function is piecewise linear and convex. The function must be convex because it is defined as the sum of convex loss functions. In this graph, each point (w_1, w_2) corresponds to one specific hypothesis, i.e., one line defined by the parameters $\mathbf{w}$ as $h(\mathbf{x}; \mathbf{w}) = 0$. This line accumulates a certain error, which we calculate by summing the loss function for each example from $\mathcal{D}$. By changing the parameters (w_1, w_2) we obtain different lines. When a line passes over an example from the set $\mathcal{D}$, the classification of that example changes from $+1$ to -1, or vice versa, so at that point there is a discrete change in the value of the error function. This is why the surface of the error function is piecewise linear. The figure on the right shows the error function that we would get with a 0-1 loss, which corresponds to the proportion of misclassified examples. We see that the problem with such an error function is that it is almost always constant. There is a sharp drop only at the points where the line passes over an example, which changes the proportion of misclassified examples. We cannot minimize such a function neither analytically nor numerically. The perceptron error, on the other hand, is mostly not constant, i.e., there is a slope that leads us to the minimum. We can also see that the perceptron error function is actually an approximation of the proportion of misclassified examples (that is, the function on the left figure is an approximation of the function on the right figure).

And, finally, let's look at optimization. Our definition of the error function is quite nice, but there is a price we have to pay for using this new loss function, namely the fact that there is no closed-form for the function's minimum, because the function is not continuous. The alternative is to apply **gradient descent**,by calculating the derivative of the loss function at each example which is incorrectly classified, and then at each such example we will update the weights in the direction opposite to the increase in loss. This will gradually lead us to the minimum of the empirical error.

The gradient of the loss for incorrectly classified examples is the following:

$$\nabla_{\mathbf{w}}\big(-\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})y\big) = -\boldsymbol{\phi}(\mathbf{x})y$$

Note that we are only interested in the gradient of the loss when the example is misclassified. When the example is classified correctly, the gradient is zero. This is how we bypassed the need to calculate the gradient of the piecewise linear function $\max(0, -\boldsymbol{\phi}(\mathbf{x})y)$.

We obtained a vector pointing in the direction of loss increase (specifically for example $\mathbf{x}$). If we want to reduce the loss, we need to go in the opposite direction. That then gives us the following **weight update rule** (engl. *pravilo ažuriranja težina*):

$$\mathbf{w} \leftarrow \mathbf{w} + \eta\boldsymbol{\phi}(\mathbf{x})y$$

where the value η determines the so-called **learning rate** (engl. *stopa učenja*), i.e., the size of the steps by which we descend to the minimum. This rule is called the **Widrow-Hoff delta-rule**.

Combining all this gives a very simple algorithm – the perceptron algorithm:

▸ **Perceptron algorithm**

```
1: initialize $\mathbf{w} \leftarrow (0, \dots, 0)$
2: repeat until convergence
3:     for $i = 1, \dots, N$
4:         if $f\big(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}^{(i)})\big) \neq y^{(i)}$ then $\mathbf{w} \leftarrow \mathbf{w} + \eta\boldsymbol{\phi}(\mathbf{x}^{(i)})y^{(i)}$
```

One can prove that, if the examples are linearly separable, then the perceptron algorithm will find a solution (the weights that correspond to a hypothesis that perfectly classifies every example from $\mathcal{D}$) in a finite number of steps. However, if the examples are not linear separable - and typically they aren't, the mapping function $\boldsymbol{\phi}(\mathbf{x})$ notwithstanding – then the perceptron algorithm will not converge.

Let's summarize the strengths and weaknesses of the perceptron algorithm. Perceptron's strength is that the algorithm is more robust than the linear regression model, in the sense that the correctly classified examples that lie far from the class boundary will not at all affect the boundary (if the example is classified correctly, the weights $\mathbf{w}$ are not updated). Another strength is the obvious simplicity of the algorithm (the optimization procedure is simpler than, say, computing the pseudoinverse of the design matrix). But perceptron also has a number of grave weaknesses. First, similarly as in regression, model outputs have no probabilistic interpretation ($h(\mathbf{x}^{(i)})$ is not constrained to the $[0,1]$ interval). Another problem is that the result (the hypothesis) depends on the initial weights and the order in which the weights are updated. The third and the most serious problem is that the perceptron algorithm does not converge if the examples are not linearly separable.

Recall that the main problem with classification using regression is the **non-robustness**: "penalizing" the very correctly classified examples. The perceptron does not have this problem. However, the perceptron does not converge if examples they are not linearly separable. Can we somehow combine these two methods, and get best of both? We'd like to have both robustness and convergence. Can we also have probabilistic output on top of that? It turns out, we can. But we'll leave that for next time, when we look into the **logistic regression** algorithm.

Summary

- **Discriminative linear models** give a linear boundary between classes

- The boundary between classes is a $(n-1)$-dimensional **hyperplane** in a n-dimensional space, defined by the weight vector $(w_0, \mathbf{w})$
- Every multiclass classification problem can be decomposed into a set of binary problems using the OVO or the OVR scheme, each with its advantages and disadvantages
- **Classification using regression** is simple, but not robust, so we won't use it
- The **perceptron** is a linear classification model that uses gradient descent to minimize an error function that approximates the number of incorrect classifications
- The perceptron **does not converge** for linearly inseparable problems, while for linearly separable ones the solution depends on weight initialization and the order of the examples
- Neither regression nor perceptron have a probabilistic output