GHENT
UNIVERSITY

Lecture Notes

E016350: Artificial Intelligence

Supervised Learning: Linear Regression and Classification

Aleksandra Pizurica

Spring 2024

Contents

1 Theory of learning

In supervised learning, we have a **training set** of N input-output pairs (training examples) $\mathcal{D}_{train} = \{(\mathbf{x}^{(i)}, y^{(i)}); i = 1, \dots N\}$. The input $\mathbf{x}^{(i)}$ is sometimes also called input **features**, and sometimes we rather use a separate notation $\phi(\mathbf{x})$ to stress that features ϕ are extracted from a (raw) input $\mathbf{x}$. The output $y^{(i)}$ is a **target** variable (also called **label**) that we are trying to predict.

When the output y is a number (such as the predicted arrival time), the learning problem is called **regression**. When the output is one of a finite set of values (e.g., land-cover class in a satellite photo: road, building, vegetation or water), the learning problem is called **classification**. In **Boolean** or **binary** classification there are only two values (e.g., classification of online comments to toxic or non-toxic or tissue classification in a medical image to normal or pathologic, etc.).

In the formal (mathematical) theory of learning, we say that given the set of training examples $\{(\mathbf{x}^{(i)}, y^{(i)}) \dots (\mathbf{x}^{(N)}, y^{(N)})\}$, where each $y^{(i)}$ was generated by an unknown function $y = f(\mathbf{x})$, the **goal of supervised learning** is to discover a function h that approximates the true function f.

The function h is a **hypothesis** about the world and is drawn from some **hypothesis space** $\mathcal{H}$. For example, the hypothesis space is the set of all polynomials up to some predefined degree. In different words, we say that h is a **model** of the data drawn from some **model class** $\mathcal{H}$, or, in general, it is a **function** drawn from some **function class**. For **parametric** models, different $h \in \mathcal{H}$ have different parameters $\mathbf{w}$. Here, the hypothesis is a parametrized function $h_{\mathbf{w}}(\mathbf{x})$, and the optimization consists in finding $\mathbf{w}$ that best fits the training data.

Fig. 1 illustrates the principle of supervised learning and Fig. 2 clarifies the notion of the hypthesis space and the functions within it. But how do we choose $\mathcal{H}$ to start with? We might exploit some prior knowledge about the underlying processes that generated the data. We can also perform **exploratory data analysis**: examining the data with statistical tests and visualizations –histograms, scatter plots, box plots – to get a feel for the data, and some insight into what hypothesis space might be appropriate [5]. Often we also try multiple hypothesis spaces (different types of models and/or different variants of the same kind of models) and evaluate which one works best for the given task.

The next question is how to choose a good hypothesis from within the hypothesis space. We say that a hypothesis is good if it correctly predicts the value of y for new examples. We then say that the hypothesis **generalizes well**. Intelligence can be seen as the **ability to predict** (e.g. the next sample) and generalize to unseen scenarios [4].

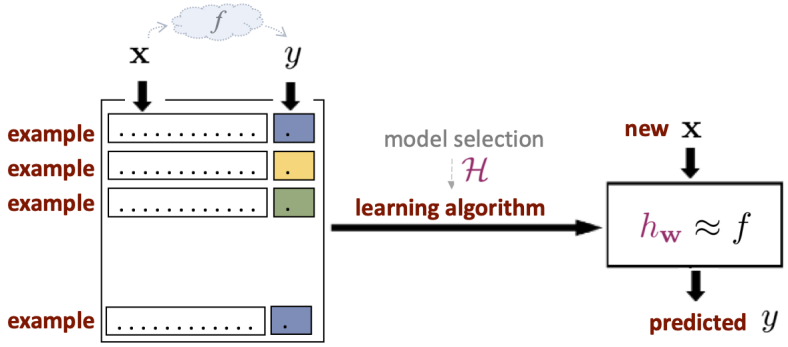

Figure 1: The principle of supervised learning, illustrated for a parameteric approach ($h = h_{\mathbf{w}}$).

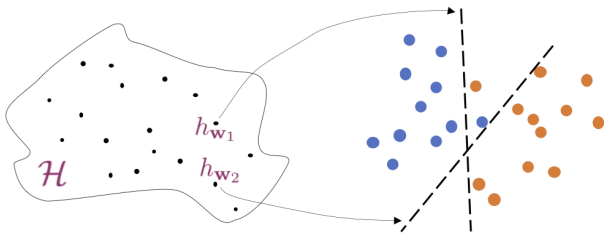

Figure 2: An illustration of the hypothesis space, exemplified with a parameteric model for binary linear classification where different hypotheses $h_{\mathbf{w}_i}$ are characterized with different parameters (weights) $\mathbf{w}_i$. Here, each $h_{\mathbf{w}_i}$ yields a different decision boundary. (e.g., $h_{\mathbf{w}_i}$ yields the predicted label for the input x as $y = 1$ if $w_{i,1}x + w_{i,0} > 0$ and $y = 0$ otherwise).

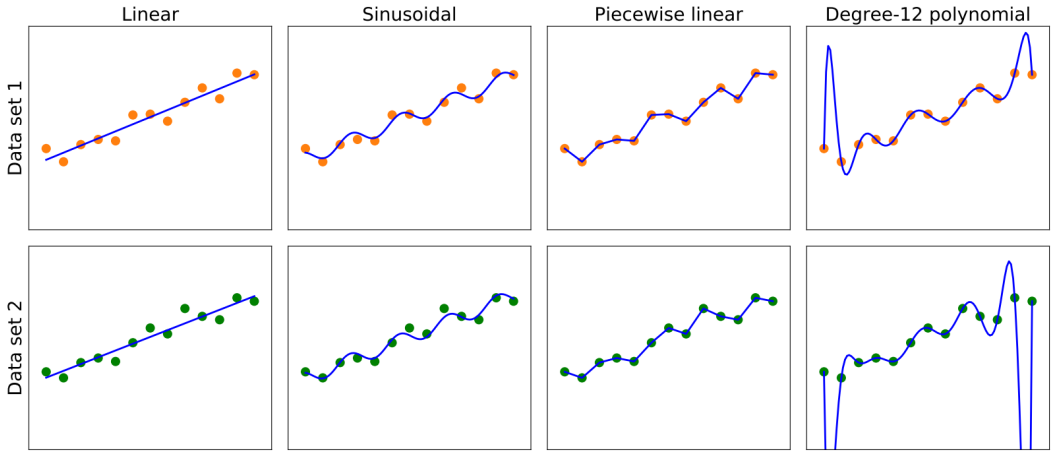

Figure 3: Finding hypotheses to fit data. Top row: four plots of best-fit functions from four different hypothesis spaces trained on data set 1. Bottom row: the same four functions, but trained on a slightly different data set (sampled from the same $f(x)$ function). Taken from [5].

The example in Fig. 3 shows how the best-fitted model differs depending on the chosen hypothesis space and it also illustrates the effect of the particular training set. Observe that the linear model has a **large bias** (i.e., a large deviation from the expected value when averaged over different training sets). It is because it allows only functions consisting of straight lines and thus fails to represent any patterns in the data other than the overall slope of a line. We say that such a hypothesis, which fails to find a pattern in the data, is **underfitting**. In the other extreme, a 12-order polynomial has a small bias but it has a **large variance**: a small fluctuation in the training data set translates into a large difference in the hypothesis. It means that at least one of the two found hypotheses must be a poor approximation for the true underlying f from which both datasets were drawn. Such a function that pays too much attention to the particular data set it is trained on is said to be **overfitting** and will perform poorly on unseen data.

Hence, we often face a **bias-variance trade-off**: a choice between more complex, low-bias hypotheses that fit the training data well and simpler, low-variance hypotheses that may generalize better [5]. A general principle, known as **Ockham's razor** tells us that the best models are simple models that fit the data well. In other words, simpler explanations are, other thing s being equal, generally better than more complex ones.

1.1 Model selection and optimization

The task of finding a good hypothesis consists of two main subtasks:

1. **Model selection**: selection of the model class, i.e, selection of the hypothesis space $\mathcal{H}$. This step determines the **hyperparameters** of the learning algorithm. For example, a learning algorithm typically involves a hyperparameter that determines the *size* of the model, like the number of layers in a neural network. If we learn a decision tree, the size could be the number of nodes in the tree; for polynomials, the maximal *degree* of the polynomial.

2. **Optimization**, also called **training**: finding the best hypothesis h within the selected $\mathcal{H}$. It involves a concrete *learning algorithm*, which needs to evaluate how good the predictors are based on some *loss function*.

1.2 Parametric models

For **parametric** models, different $h \in \mathcal{H}$ have different parameters $\mathbf{w}$. Here, the hypothesis is a parametrized function $h_{\mathbf{w}}(\mathbf{x})$, and the optimization consists in finding $\mathbf{w}$ that best fits the training data. Most of the machine learning approaches that we will cover in this course, ranging from the simplest linear regression and logistic regression to deep neural networks are parameteric models. In this chapter, we will deal with parameteric models but we will address nonparameteric machine learning models as well in some of the subsequent chapters.

1.3 Training, validation and test sets

In machine learning, we want to select a hypothesis that will optimally fit some future examples. Here we are implicitly making an assumption that the future examples will behave like the past ones. This is called **stationarity** assumption. Next, we need to define what is the *optimal* fit (i.e., the *best* hypothesis). We will say that it is a hypothesis that minimizes some **error rate**, being the proportion of times that $h(\mathbf{x}) \neq y$ for an $(\mathbf{x}, y)$ example. To estimate the error rate of a hypothesis, we need to test it by measuring its performance on a **test set** of examples. To ensure a fair evaluation, the simplest way is to split the set of all available examples into a **training set** (to create the hypothesis, i.e., to train the model) and a **test set** (to evaluate it). When we are creating only one model these two sets are sufficient. But if we want to compare multiple competing models, which can be entirely different machine learning models or variants of the same approach (with differently adjusted hyperparameters), then we need a third set of examples called the **validation set**. Hence, as a general rule, we need to split the set of all available examples $\mathcal{D}$ into three disjoint sets:

- **Training set** ($\mathcal{D}_{train}$) – to train candidate models
- **Validation set** ($\mathcal{D}_{val}$) – to evaluate the candidate models and choose the best one
- **Test set** ($\mathcal{D}_{test}$) – to do a final unbiased evaluation of the best model

When we don't have enough data to make properly all three of these data sets with sufficient sizes, we can use k-**fold cross-validation**. This technique enables us to "squeeze more" out of the data by allowing each sample to serve double duty – as training and validation data – but not at the same time [5]. We then perform k rounds of learning, each time $1/k$ of the data in $\mathcal{D} \setminus \mathcal{D}_{test}$ is held as a validation set and the remaining examples are used as the training examples. ($\mathcal{D}_{test}$ is always

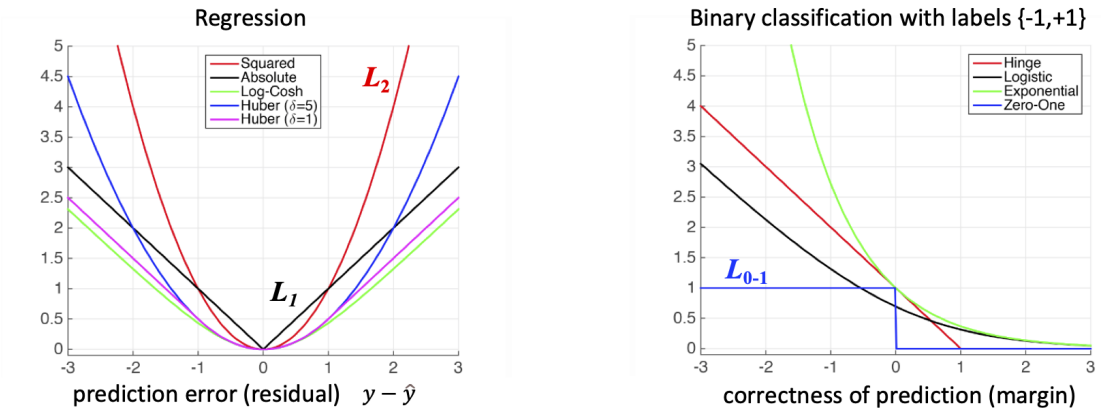

Figure 4: Common loss functions for the regression tasks (left) and binary classification (right).

kept separately and not touched until the testing phase). Popular choices for k are 5 and 10. The extreme case with $k = n$ is known as **leave-one-out cross validation**.

1.4 Loss function

Since in AI the decision making (and optimal action) maximizes some form of **expected utility**, we can define the **loss function** in general terms as the amount of utility lost by replacing the correct answer $f(\mathbf{x}) = y$ by a hypothesis $h(\mathbf{x}) = \hat{y}$. This is the most general formulation. We will write the loss function as $L(y, h(\mathbf{x}))$ and often simply as $L(y, \hat{y})$. In the case of *parameteric* learning $\hat{y} = h_{\mathbf{w}}(x)$ we will use interchangeably $L(y, h_{\mathbf{w}}(\mathbf{x}))$ and $L(\mathbf{x}, y, \mathbf{w})$.

Fig. 4 illustrates some common loss functions for the regression tasks and for the binary classification tasks. For the regression task, the loss function needs to increase with the difference between the true and the predicted output. This increase is linear for the **absolute value loss** $L_1(y, \hat{y}) = |y - \hat{y}|$ and quadratic for the **squared-error loss**: $L_2(y, \hat{y}) = (y - \hat{y})^2$. The **Huber** loss function behaves as quadratic for small prediction errors and liner for large prediction errors:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

This makes it less sensitive to outliers in data than the squared error loss. Its parameter δ allows adjusting the transition between the two regions and the slope of the linear part.

For classification, we want to penalize the cases that will yield $\hat{y} \neq y$, but now it makes no sense to let the loss depend on the value of the difference $y - \hat{y}$ (think why). Rather, we now can reason how *confident* the model was when making a certain prediction and let the loss depend on the *correctness of prediction*, which is larger when the model is more confident about its correct prediction, and, conversely, is smaller when the model is more confident about its wrong prediction. Fig. 4 shows some common classification loss functions. For the **zero-one loss** $L_{0-1}(y, \hat{y})$ the penalty is 0 if $\hat{y} = y$ and 1 if $\hat{y} \neq y$ regardless of how confident the model was in predicting a particular value of $\hat{y}$. Other, more nuanced loss functions shown in the diagram on the right of Fig. 4 take this confidence into account.

1.5 Training loss

The **expected generalization loss** for a hypothesis h, with respect to loss function L is the mathematical expectation of the loss:

$$GenLoss_L(h) = \sum_{(\mathbf{x},y)\in\mathcal{E}} L(y, h(\mathbf{x}))P(\mathbf{x}, y) \tag{1}$$

where $\mathcal{E}$ denotes the set of all possible input-output pairs, and $P(\mathbf{x}, y)$ the joint probability of $\mathbf{x}$ and y. The **best hypothesis** is the one that yields the minimum expected generalization loss:

$$h^* = \arg\min_{h\in\mathcal{H}} GenLoss_L(h) \tag{2}$$

In reality, the true distribution $P(\mathbf{x}, y)$ is not known, so the learning agent can only estimate the generalization loss with an **empirical loss** on a set of available examples E:

$$EmpLoss_{L,E}(h) = \frac{1}{|E|} \sum_{(\mathbf{x},y)\in E} L(y, h(\mathbf{x})) \tag{3}$$

By minimizing the empirical loss, we obtain the **estimated best hypothesis**:

$$\hat{h}^* = \arg\min_{h\in\mathcal{H}} EmpLoss_{L,E}(h) \tag{4}$$

We define the **training loss** as the empirical loss over the set of training examples $\mathcal{D}_{train}$:

$$TrainLoss_{L,\mathcal{D}_{train}}(h) = \frac{1}{|\mathcal{D}_{train}|} \sum_{(\mathbf{x},y)\in\mathcal{D}_{train}} L(y, h(\mathbf{x})) \tag{5}$$

For compactness, we will suppress the subscripts L and $\mathcal{D}_{train}$, and for parameteric models $h_{\mathbf{w}}$, the loss function can be written both as $L(y, h_{\mathbf{w}}(\mathbf{x}))$ and $L(\mathbf{x}, y, \mathbf{w})$, so we also write:

$$TrainLoss(\mathbf{w}) = \frac{1}{|\mathcal{D}_{train}|} \sum_{(\mathbf{x},y)\in\mathcal{D}_{train}} L(\mathbf{x}, y, \mathbf{w}) \tag{6}$$

Sometimes we will find it more convenient to express the training set explicitly as N examples: $\mathcal{D}_{train} = \{(\mathbf{x}^{(i)}, y^{(i)}); i = 1, \dots N\}$, and then write the training loss as

$$TrainLoss(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, h_{\mathbf{w}}(\mathbf{x}^{(i)})) = \frac{1}{N} \sum_{i=1}^{N} L(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}) \tag{7}$$

which is equivalent to (6). Finally, we will sometimes use **explicit regularization** of a machine learning model, meaning that we will add a penalty term that penalizes the complexity of the solution. In other words, we will directly minimize a weighted sum of the empirical loss and the complexity of the hypothesis, which is also called the **total cost**:

$$Cost(h) = EmpLoss(h) + \lambda Complexity(h) \tag{8}$$

Here, $\lambda \geq 0$ is a parameter, often determined by cross-validation. This process explicitly penalizes complex hypotheses, promoting thus more 'regular' functions as solutions and is therefore called regularization. In practice, our training objective will become:

$$\min_{\mathbf{w}} \sum_{i=1}^{N} L(y^{(i)}, h_{\mathbf{w}}(\mathbf{x}^{(i)})) + \lambda Reg(\mathbf{w}) \tag{9}$$

where $Reg(\mathbf{w})$ is some regularization function imposed on the weights, e.g., ℓ_1-regularization: $Reg(\mathbf{w}) = |\mathbf{w}|$ or ℓ_2-regularization: $Reg(\mathbf{w}) = \mathbf{w}^2$. We address regularization more concretely in Section 2.2.

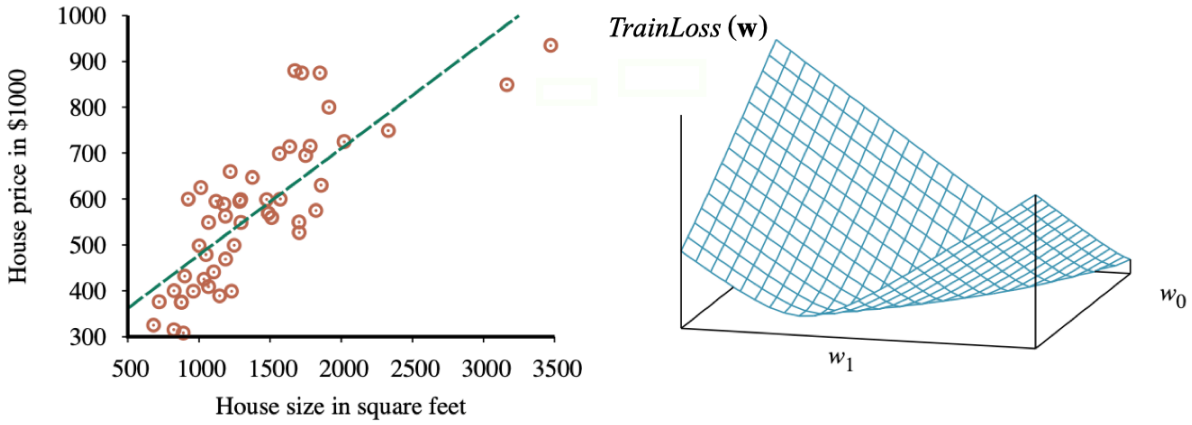

Figure 5: Left: data points and the fitted line under the squared error loss. Right: Plot of the training loss $\sum_i (y^{(i)} - (w_1 x^{(i)} + w_0))^2$ for various values of w_0, w_1. Observe that the training loss is a *convex* function. This is true for *every* linear regression problem with an L_2 loss function [5].

2 Linear regression

The task of linear regression is fitting a linear model through the training data. The hypothesis space is the space of all **linear functions** of continuous-valued inputs. The learning algorithm seeks to find the parameters $\mathbf{w}$, which characterize the best fitted line. The produced model $h_{\mathbf{w}}$ in the context of regression is called a **predictor**.

2.1 Univariate linear regression

In univariate linear regression, the inputs are one-dimensional (real numbers) x. The goal is to *fit a straight line*, i.e., to learn the coefficients w_0 and w_1 of a **univariate linear function**:

$$y = w_1 x + w_0 \tag{10}$$

The coefficients w_0 and w_1 can be seen as **weights**: the value of y is changed by changing the relative weight of one term or another.

Denoting the vector of weights by $\mathbf{w} = [w_0\ w_1]^\top$, the predictor is

$$h_{\mathbf{w}}(x) = w_1 x + w_0 \tag{11}$$

and the task is to find $\mathbf{w}$ such that $h_{\mathbf{w}}(x)$ fits best the training data.

Typically, squared error loss function L_2 is used, which is then called **least squares linear regression**. An example is shown in Fig. 5.

A historical note: Carl Friedrich Gauss showed that when the noise in the outputs y is normally distributed, the most likely values of the weights are obtained using the L_2 loss, i.e., minimizing the sum of the squared errors [1]. It is believed that Gauss used this method in 1801 to model the orbit of Ceres (a dwarf planet) and to predict its location. The Ceres dataset consisted of 19 observations of the Ceres's locations, acquired over 42 days, where each data point consisted of a time stamp and the location on the sky. Ceres was located within 1/2 degree of Gauss's prediction, which was a far better prediction than made by other astronomers at the time. The least squares method was published by the French mathematician Adrien-Marie Legendre in 1805, and is attributed to Legendre but usually also co-credited to Gauss.

The training loss for the linear regression is thus commonly defined with the L_2 loss function. If the training set consists of N examples: $\mathcal{D}_{train} = \{(x^{(i)}, y^{(i)}); i = 1, \dots N\}$, the training loss is

$$TrainLoss(\mathbf{w}) = \sum_{i=1}^{N} (y^{(i)} - (w_1 x^{(i)} + w_0))^2 \tag{12}$$

We find the weights w_0 and w_1 by setting to zero the corresponding partial derivatives of the training loss, i.e.,

$$\frac{\partial}{\partial w_0} \sum_{i=1}^{N} (y^{(i)} - (w_1 x^{(i)} + w_0))^2 = 0; \qquad \frac{\partial}{\partial w_1} \sum_{i=1}^{N} (y^{(i)} - (w_1 x^{(i)} + w_0))^2 = 0 \tag{13}$$

This yields:

$$w_1 = \frac{N \sum_i x^{(i)} y^{(i)} - \sum_i x^{(i)} \sum_i y^{(i)}}{N \sum_i (x^{(i)})^2 - \left(\sum_i x^{(i)}\right)^2}; \qquad w_0 = \left(\sum_i y^{(i)} - w_1 \sum_i x^{(i)}\right)/N \tag{14}$$

Although in this case we have a closed form solution, we will need a more general method for determining the weights that does not rely on solving to find the zeroes of the derivatives of the entire training loss. This will be needed for various reasons, e.g., when using other than L_2 loss functions and/or when the training examples arrive sequentially.

One such method that can be applied to any loss function – no matter how complex it is – is the **gradient descent**. It searches through a continuous weight space by incrementally modifying the parameters. Fig. 6 gives the pseodo-code and illustrates its operation. The parameter α, which is called the **step size** is usually called the **learning rate** when we are trying to minimize a loss in a learning problem. It can be a fixed constant, or chosen to decay as the learning process proceeds [5]. For linear regression with the L_2 loss function, the training loss is always **convex** and thus gradient descent will find the optimal solution.

Let us now derive the learning rule for the linear regression with the common L_2 loss. We start from a simplified case with one training example (x, y):

$$\frac{\partial}{\partial w_j} TrainLoss(\mathbf{w}) = \frac{\partial}{\partial w_j} (y - h_{\mathbf{w}}(x))^2 \tag{15}$$

Applying to both w_0 and w_1, we get

$$\frac{\partial}{\partial w_0} TrainLoss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)); \qquad \frac{\partial}{\partial w_1} TrainLoss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x))x; \tag{16}$$

w ← any point in the parameter space
while not converged **do**
For each w_i **in w do**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} TrainLoss(w)$$

step size (learning rate)

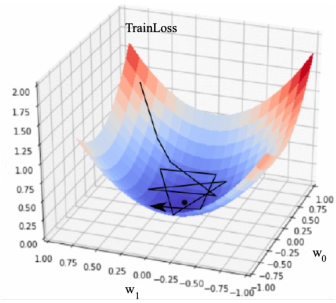

Figure 6: Left: Pseudo-code of the gradient descent algorithm. Right: An illustration of its operation (image from [1]).

Plugging this into the update rule of the gradient descent and folding the constant 2 into the unspecified learning rate[1] α, we obtain

$$w_0 \leftarrow w_0 + \alpha(y - h_{\mathbf{w}}(x)); \qquad w_1 \leftarrow w_1 + \alpha(y - h_{\mathbf{w}}(x))x; \tag{17}$$

These learning rules are intuitive: if $h_{\mathbf{w}}(x) = y$, the prediction is correct so don't change anything (keep the weights as they are). If $h_{\mathbf{w}}(x) > y$, i.e., the output of the hypothesis is too large, reduce w_0 a bit and reduce w_1 if x is positive but increase w_1 if x was negative. The opposite holds when $h_{\mathbf{w}}(x) < y$. While these updates were obtained for one training example, we can simply extend them to the case with N training examples. What changes is that in Eq (15) we will have the partial derivative of the sum $\sum_i (y^{(i)} - h_{\mathbf{w}}(x^{(i)}))^2$ instead of the partial derivative of a single element $(y - h_{\mathbf{w}}(x))^2$. Since the derivative of a sum is the sum of the derivatives, the expressions in Eq (17) generalize straightforwardly to

$$w_0 \leftarrow w_0 + \alpha \sum_{i=1}^{N} (y^{(i)} - h_{\mathbf{w}}(x^{(i)})); \qquad w_1 \leftarrow w_1 + \alpha \sum_{i=1}^{N} (y^{(i)} - h_{\mathbf{w}}(x^{(i)}))x^{(i)}; \tag{18}$$

Note that here we are summing over all the N training examples in each step. This is the so-called **batch gradient descent** learning rule for univariate linear regression. Since the loss function is convex, convergence to the optimum is guaranteed unless the learning rate is chosen too large so that it "overshoots" (see the following notes). A faster variant, called **stochastic gradient descent** makes in each step updates based on one randomly selected sample, according to Eq (17). Most commonly, the updates are made on a **minibatch** of m out of total N examples, and the resulting learning rule is known as the **minibatch gradient descent**. We return to these optimization aspects in the following notes.

2.2 Multivariate linear regression

In **multivariate** linear regression the input is a vector $\mathbf{x} = [x_1, \ldots, x_n]^\top$. Some authors use the terms multivariate, **multivariable** and **multiple** linear regression interchangeably. In some cases, a differentiation is made in the sense that the term *multivariate* denotes a more general case where the output is also a vector (this is the terminology, e.g., in [5]). We will here consider that the output is always a single number y.

[1]Note that we could have written this update rule with some learning rate $\alpha' = 2\alpha$ but since the learning rate is not specified, we simply denote the new constant with some generic α, and we could have chosen any other symbol.

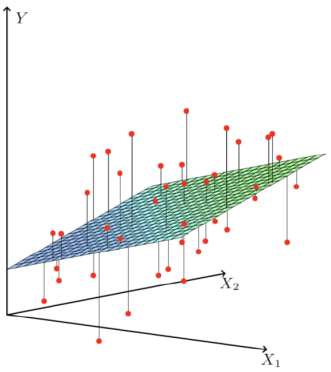

Figure 7: An illustration of the multivariate linear regression taken from [3].

We generalize the univariate linear regression to the case where each input $\mathbf{x}$ is a vector as:

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \cdots + w_n x_n = w_0 + \sum_j w_j x_j \tag{19}$$

In order to treat the intercept term w_0 in the same way as the others, we introduce a dummy input attribute x_0 , which is always 1. Then we can write compactly

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} = \mathbf{w}^\top \mathbf{x} = \sum_j w_j x_j \tag{20}$$

The inner product $\mathbf{w} \cdot \mathbf{x}$ is in the machine learning literature often called the **score**.

The optimal weights can be obtained analytically, using the tools of linear algebra and vector calculus. Let $\mathbf{X}$ be the **data matrix** defined as the matrix of inputs, where each raw is one n-dimensional input example: $\mathbf{X}(:, i) = (\mathbf{x}^{(i)})^\top = [x_1^{(i)}, \ldots, x_n^{(i)}]$. One can show that

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_i L_2(y^{(i)}, \mathbf{w} \cdot \mathbf{x}^{(i)}) = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = \underbrace{(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top}_{pseudoinverse} \mathbf{y} \tag{21}$$

So, we can obtain an analytical solution for the optimal (in the mean squared error sense) weights as the **pseudoinverse** $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ of the data matrix $\mathbf{X}$. In practice, it may be difficult to calculate the pseudoinverse of a large data matrix.

Instead, we can employ the gradient descent, which updates the weights as

$$w_j \leftarrow w_j + \alpha \sum_i (y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)})) x_j^{(i)} \tag{22}$$

Gradient descent reaches the **unique minimum** because the training loss remains convex also for multivariate regression with the squared loss function.

2.3 Regularization

With multivariable linear functions it is common to use some form of **regularization** to avoid overfitting. This is because in high-dimensional spaces the data are more sparse and the chance is bigger that some dimensions that are in fact irrelevant are given more importance only because by chance they appeared to be useful.

In Eq (8), we defined a total cost by adding explicitly a regularization term to the optimization problem. For linear regression, we commonly specify the complexity of the hypthesis in terms of its weights, and particularly we consider the so-called ℓ_p family[2] of regularization functions:

$$Complexity(h_{\mathbf{w}}) = \ell_p(\mathbf{w}) = \sum_j |w_j|^p \tag{23}$$

Here, $\ell_p(\mathbf{w}) = \|w\|_p^p$, where $\|w\|_p$ is the L_p-norm:

$$\|\mathbf{w}\|_p = \Big(\sum_j w_j^p\Big)^{1/p} \tag{24}$$

With the squared error loss and ℓ_p-regularization, the total cost, being now the training loss, is

$$TrainLoss(\mathbf{w}) = Cost(h_{\mathbf{w}}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \ell_p(\mathbf{w}) \tag{25}$$

where $\lambda > 0$ is a regularization parameter. The optimal parameters follow as before from the minimization of the resulting loss:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \ell_p(\mathbf{w}) \tag{26}$$

Two special cases are of particular interest: $p = 2$ and $p = 1$. For $p = 2$, the resulting optimization problem is known as the **Ridge regression** or **Tikhonov regularization**:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2 = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \Big(y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)})\Big)^2 + \lambda \sum_j w_j^2 \tag{27}$$

For $p = 1$, the problem is known as the Least Absolute Shrinkage and Selection Operator (LASSO) regression:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_1 = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \Big(y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)})\Big)^2 + \lambda \sum_j |w_j| \tag{28}$$

LASSO regression promotes sparse solutions. Fig. 8 explains this pictorially and illustrates the effect of both of these two regularization strategies. Note that we are minimizing the sum of two terms $Loss(\mathbf{w}) + \lambda Complexity(\mathbf{w})$, which is equivalent to minimizing $Loss(\mathbf{w})$ subject to the constraint that $Complexity(\mathbf{w}) \leq c$, for some constant c that is related to λ [5]. The shaded regions

[2] In some cases, which are beyond the scope of this course, $\ell_{p,q}$ -regularization is used. For example, to impose 'structured sparsity', the coefficients $\mathbf{w}$ are structured into r groups $\mathbf{w} = [\mathbf{w}_{\mathcal{G}_1}^\top, \ldots, \mathbf{w}_{\mathcal{G}_r}^\top]^\top$ and different types of behaviour are promoted within and across the groups, which is expressed by adding an $\ell_{p,q}$-regularization term $\|\mathbf{w}\|_{p,q}^q$, where $\|\mathbf{w}\|_{p,q} = (\sum_i^r \|\mathbf{w}_{\mathcal{G}_i}\|_p^q)^{1/q}$ and $\mathcal{G}_i$ is the index set of the i-th group of coefficients. Particularly, $\ell_{2,1}$ regularization is popular in group sparse optimization. For more details, see [2].

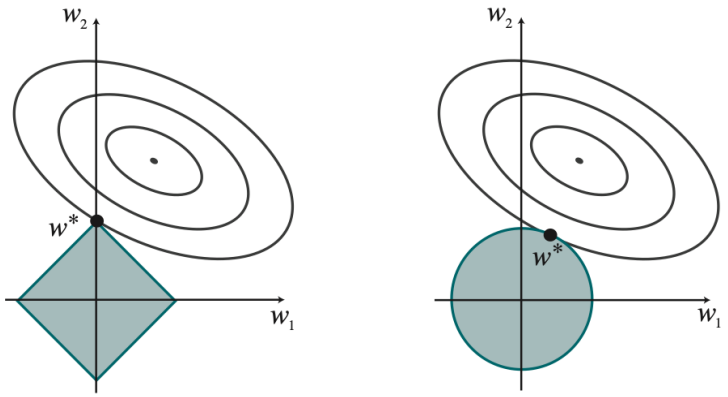

Figure 8: An illustration of the effects of ℓ_1 (left) and ℓ_2 (right) regularization. The concentric ovals are the contours of the loss function without regularization (minimum in the middle) and the shaded areas are the constraints for the corresponding regularization – the solution has to be within the shaded area. Observe that for ℓ_1 the solution will likely be on an axis (meaning other coefficients zero $\rightarrow$ sparse solution). Illustration from [5].

in the figure (a diamond-shape for ℓ_1 and a circle for ℓ_2) represent the set of points that satisfy the constraint. For both LASSO and ridge regression $Loss(\mathbf{w})$ is squared-error loss, represented in the figure with concentric contours, with the minimum (smallest achievable loss) being the point in the middle. The optimal solution is where the shaded area touches the loss contour closest to the minimum. We can see that with ℓ_1 this will likely be along some of the axes, simply because the constraint area is pointy. It means that at least some of the components of this solution $\mathbf{w}^*$ will be zero, hence, the solution will be a **sparse** vector.

Why do we want to impose a sparsity constraint? Firstly, sparsity allows us to model naturally phenomena that are often appearing in real-world signals and images. Secondly, it also brings various practical (modelling and computational) advantages, as we will explain in the following.

In natural signals, sparsity refers to the phenomenon where only a small number of components or features in the signal are significant or carry essential information, while the majority are negligible or redundant. For example, in neuroscience, brain signals recorded from electrodes often exhibit sparsity because only certain neural events or activities are relevant to a particular cognitive process or behavior. In audio signals, such as speech or music, sparsity occurs because most sounds can be represented using a relatively small number of frequency components.

Incorporating a sparsity constraint in a computational model allows for more efficient representation and processing of the signals by focusing computational resources on the most relevant components while ignoring or compressing the redundant ones. When dealing with hight dimensional signal, sparsity provides also a way of mitigating the **curse of dimensionality**[3]

[3]The term *curse of dimensionality* refers to various challenges and phenomena that arise when working with high-dimensional data spaces. As the number of dimensions increases, the amount of data required to effectively cover the space increases exponentially. This leads to various issues, including an increased risk of overfitting, the need to gather and label large amounts of data, which can be costly and time-consuming, and processing of such data requires often huge (or even prohibitive) computation complexity.

Additional insight (optional reading): Imposing sparse constraints in optimization and machine learning can be highly beneficial for several reasons:

- **Interpretability**: Sparse models are often more interpretable because they focus on a small subset of features that are deemed most relevant for the task at hand. For example, in LASSO regression, the non-zero coefficients indicate which features are most predictive of the outcome.
- **Dimensionality Reduction**: Sparsity helps in reducing the dimensionality of the problem by selecting only a subset of the available features. This can mitigate the curse of dimensionality, making the models more tractable and less prone to overfitting, especially in high-dimensional spaces.
- **Improved Generalization**: By focusing on the most informative features, sparse models often generalize better to unseen data. They are less likely to overfit to noise or irrelevant features present in the training data, leading to better performance on test or validation sets.
- **Robustness to Noise**: Sparse regularization can enhance the robustness of models by filtering out noisy or irrelevant features. This can help in improving the model's performance in the presence of noisy data or outliers.
- **Memory Efficiency**: Sparse representations require less memory storage compared to dense representations, especially when dealing with large datasets. This makes sparse models more scalable and feasible for deployment in resource-constrained environments.
- **Feature Selection**: Sparse regularization techniques naturally perform feature selection by encouraging many feature weights to be exactly zero. This automatic feature selection mechanism simplifies the model and can eliminate irrelevant or redundant features, leading to simpler and more efficient models.

We have seen various advantages of sparse optimization. Tikhonov regularization shares some of these (it also mitigate overfitting and improves generalization) and has its distinctive advantages as well, especially in terms of stability, ease of implementation and computational efficiency (see the additional insight parts for more details).

Additional insight (optional reading): Imposing Tikhonov (ℓ_2) regularization in optimization and machine learning has the following advantages:

- **Interpretability**: Tikhonov regularization has a well-understood theoretical foundation, especially in the context of linear regression. The solution obtained often has a closed-form expression, allowing for easier interpretation of model parameters and their significance.
- **Improved Generalization**: Tikhonov regularization helps prevent overfitting by penalizing large parameter values. The regularization term encourages the model to capture the underlying structure of the data rather than fitting the noise present in the training set. By controlling overfitting, it enables improved generalization performance.
- **Less Susceptible to Small Variations**: Tikhonov regularization tends to be less sensitive to small variations in the input data compared to LASSO. This is because the penalty term in Tikhonov regularization is proportional to the square of the parameter values, which provides more stability against Gaussian noise.
- **Continuous Solution**: Tikhonov regularization typically results in a solution with non-zero values for all parameters, albeit some may be very small. This continuous shrinkage of parameter values can be advantageous when the problem domain requires a smooth and continuous solution.
- **Handles Multicollinearity Better**: Tikhonov regularization performs well in the presence of multicollinearity, where predictor variables are highly correlated. It tends to distribute the penalty evenly among correlated variables, whereas LASSO may arbitrarily select one of them and shrink the others to zero.
- **Simple Implementation**: Tikhonov regularization is easy to implement and can be incorporated into various machine learning algorithms using standard techniques such as gradient descent or closed-form solutions. The additional computational cost is usually minimal compared to the benefits gained in terms of performance and stability.
- **Efficient Computation**: The optimization problem associated with Tikhonov regularization often has a closed-form solution or can be solved efficiently using standard optimization techniques. This makes Tikhonov regularization computationally less demanding compared to LASSO, especially for large-scale problems.

The choice between Tikhonov regularization and LASSO depends on the specific characteristics of the dataset and the goals of the modeling task. LASSO may be preferred when feature selection or sparsity of the solution is desired, while Tikhonov regularization may be more suitable for problems where a continuous solution with less sensitivity to noise is required. Techniques such as **Elastic net regularization** combine the strengths of both Tikhonov regularization and LASSO, offering a more flexible regularization approach.

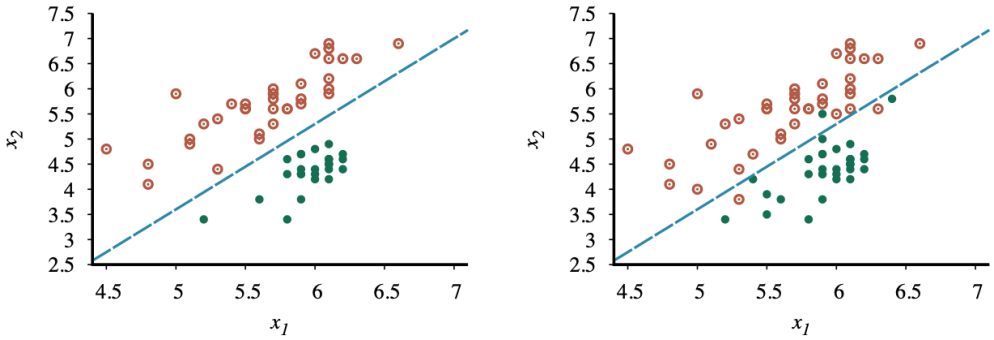

Figure 9: Left: A linearly separable dataset consisting of two classes and a linear decision boundary. x_1 and x_2 are two seismic parameters measured for earthquakes (orange circles) and explosions (green dots). Right: The same domain with more data, no longer linearly separable. This situation is common in real world. The image is taken from [5].

3 Linear Classification

Now we turn to the linear **classification framework**. As before, we are given training data, which consists of a set of examples $(\mathbf{x}, y)$, but y's are now some discrete class labels. We will focus first on the **binary classification** problem in which y can take only two values, 0 and 1. An example of this problem is shown in Fig. 9, where the input $\mathbf{x}$ consists of two components x_1 and x_2. The task of classification is to learn a model h that we call a **classifier** and that will for a new input $\mathbf{x}$ return the class label 0 or 1. A line (or surface, in higher dimensions) that separates the two classes is called a **decision boundary**. A linear decision boundary is called a **linear separator** and a dataset that can be ideally separated by at least one linear decision boundary is **linearly separable**.

3.1 Linear classification with a hard threshold

Consider the linearly separable case in Fig. 9 on the left. The depicted linear separator is:

$$x_2 = 1.7x_1 - 4.9 \quad \text{or} \quad -4.9 + 1.7x_1 - x_2 = 0 \tag{29}$$

We want to classify the explosions (green dots) with value 1. For these points $-4.9+1.7x_1-x_2 > 0$, while for earthquakes (orange circles) it holds $-4.9 + 1.7x_1 - x_2 < 0$. Introducing a dummy input $x_0 = 1$, we can write this compactly in vector form, with $\mathbf{w} = [-4.9, 1.7, -1]^\top$, as $\mathbf{w} \cdot \mathbf{x} > 0$ in one case and $\mathbf{w} \cdot \mathbf{x} < 0$ in the other. Thus, we can write the classification hypothesis as follows:

$$h_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{30}$$

We can think of this classifier as the result of passing $\mathbf{w} \cdot \mathbf{x}$ through a threshold function:

$$h_{\mathbf{w}}(\mathbf{x}) = Threshold(\mathbf{w} \cdot \mathbf{x}), \quad \text{where} \quad Threshold(z) = \begin{cases} 1 & z \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{31}$$

The question is now how to choose the weights $\mathbf{w}$ to minimize the loss. For linear regression, we were able to determine the weights that minimize the squared error loss both analytically and using

gradient descent algorithm. Here we cannot do that because with $h_{\mathbf{w}}(\mathbf{x})$ being a step function the gradient of the training loss is zero almost everywhere in the weight space, except at the transition $\mathbf{w} \cdot \mathbf{x} = 0$ where it is not defined.

However, it can be shown that the simple update rule called the **perceptron learning rule**

$$w_j \leftarrow w_j + \alpha(y - h_{\mathbf{w}}(\mathbf{x}))x_j \tag{32}$$

converges to the perfect linear separator (provided that data are linearly separable). Observe the following behaviour of this update:

- If the output is correct, i.e., $y = h_{\mathbf{w}}(x)$, the weights are not changed
- If $y = 1$ but $h_{\mathbf{w}}(\mathbf{x}) = 0$, then w_j is *increased* when x_j is positive and is *decreased* when x_j is negative. This is because we want to make $\mathbf{w} \cdot \mathbf{x}$ bigger so that $h_{\mathbf{w}}(x)$ outputs 1
- If $y = 0$ but $h_{\mathbf{w}}(\mathbf{x}) = 1$, then w_j is *decreased* when x_j is positive and is *increased* when x_j is negative. This way we make $\mathbf{w} \cdot \mathbf{x}$ smaller so that $h_{\mathbf{w}}(x)$ outputs 0

3.2 Soft classification with a logistic function

The hard nature of the classification threshold causes some problems. The fact that the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable and is a discontinuous function of its inputs and its weights, makes learning with the perception rule very unpredictable [5]. Furthermore, the linear classifier always announces a "completely confident" prediction 0 or 1, while we often need more graduated predictions. These problems are alleviated by **softening** the threshold function: approximating a hard threshold with a continuous, differentiable function.

A widely used soft-threshold function is the **logistic** function, also called **sigmoid** function:

$$Logistic(z) = \frac{1}{1 + e^{-z}} \tag{33}$$

Replacing the hard threshold with the logistic function, the classification hypothesis becomes

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}} \tag{34}$$

Logistic regression is the process of fitting the weights of this model to minimize loss on a data set [5]. Here we will show how the update rule is derived for the logistic regression under the L_2 loss. In the next note, we will address in more detail logistic regression, focusing on maximum likelihood estimation.

3.3 Least-square error logistic regression

We first derive the update rule for the logistic regression under L_2 loss. Let g denote the logistic function and g' its derivative. As we did for linear regression, we will use the chain rule for the derivatives: $\partial g(f(x))/\partial x = g'(f(x))(\partial f(x)/\partial x)$

We start again from a simplified case with *one* training example $(\mathbf{x}, y)$. The derivation is similar as for the linear regression but now $h_{\mathbf{w}}(\mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})$, so we have:

$$\begin{aligned}\frac{\partial}{\partial w_j} TrainLoss(\mathbf{w}) &= \frac{\partial}{\partial w_j}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\ &= 2(y - h_{\mathbf{w}}(\mathbf{x}))\frac{\partial}{\partial w_j}(y - h_{\mathbf{w}}(\mathbf{x})) \\ &= -2(y - h_{\mathbf{w}}(\mathbf{x}))g'(\mathbf{w}\cdot\mathbf{x})\frac{\partial}{\partial w_j}(\mathbf{w}\cdot\mathbf{x}) \\ &= -2(y - h_{\mathbf{w}}(\mathbf{x}))g'(\mathbf{w}\cdot\mathbf{x})x_j\end{aligned}$$

The derivative of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w}\cdot\mathbf{x}) = g(\mathbf{w}\cdot\mathbf{x})(1 - g(\mathbf{w}\cdot\mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \tag{35}$$

and the weight update for minimizing the loss is

$$w_j \leftarrow w_j + \alpha(y - h_{\mathbf{w}}(\mathbf{x}))h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))x_j \tag{36}$$

Note that this rule was derived for *one* training example (or for the stochastic gradient descent). You should know how to generalize it to the update rule based on N examples!

References

[1] M. Charikar and S. Koyejo. (CS221): Artificial intelligence: Principles and techniques. *Stanford*.

[2] Y. Hu, C. Li, Meng K., J. Qin, and X. Yang. Group sparse optimization via $\ell_{p,q}$ regularization. *Journal of Machine Learning Research*, 18, 2017.

[3] N. Narasimhan. Multiple linear regression-an intuitive approach. *Medium*, 2020.

[4] T Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the AMS*, 50(5):537–544, 2003.

[5] S Russel and Norvig. P. *Artificial Intelligence: A Modern Approach, 4th Edition*. Pearson, 2021.