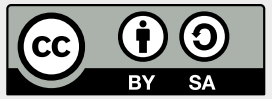
Linear Regression II, SGD, Perceptron

Milan Straka

October 14, 2019

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics



Linear Regression

Given an input value $\boldsymbol{x} \in \mathbb{R}^d$, one of the simplest models to predict a target real value is **linear regression**:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = x_1 w_1 + x_2 w_2 + \ldots + x_D w_D + b = \sum_{i=1}^{d} x_i w_i + b = \boldsymbol{x}^T \boldsymbol{w} + b.$$

The *bias* b can be considered one of the *weights* $\boldsymbol{w}$ if convenient.

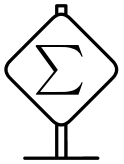
By computing derivatives of a sum of squares error function, we arrived at the following equation for the optimum weights:

$$\boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} = \boldsymbol{X}^T \boldsymbol{t}.$$

If $\boldsymbol{X}^T \boldsymbol{X}$ is regular, we can invert it and compute the weights as $\boldsymbol{w} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{t}$.

Matrix $\boldsymbol{X}^T \boldsymbol{X}$ is regular if and only if $\boldsymbol{X}$ has rank d, which is equivalent to the columns of $\boldsymbol{X}$ being linearly independent.

SVD Solution of Linear Regression



Now consider the case that $\boldsymbol{X}^T\boldsymbol{X}$ is singular. We will show that $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{t}$ is still solvable, but it does not have a unique solution. Our goal in this case will be to find the smallest $\boldsymbol{w}$ fulfilling the equation.

We now consider *singular value decomposition (SVD)* of X, writing $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, where

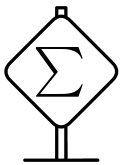
- $\boldsymbol{U} \in \mathbb{R}^{N\times N}$ is an orthogonal matrix, i.e., $\boldsymbol{u}_i^T\boldsymbol{u}_j = [i = j]$,
- $\boldsymbol{\Sigma} \in \mathbb{R}^{N\times D}$ is a diagonal matrix,
- $\boldsymbol{V} \in \mathbb{R}^{D\times D}$ is again an orthogonal matrix.

Assuming the diagonal matrix $\boldsymbol{\Sigma}$ has rank r, we can write it as

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_r & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix},$$

where $\boldsymbol{\Sigma}_r \in \mathbb{R}^{d\times d}$ is a regular diagonal matrix. Denoting $\boldsymbol{U}_r$ and $\boldsymbol{V}_r$ the matrix of first r columns of $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively, we can write $\boldsymbol{X} = \boldsymbol{U}_r\boldsymbol{\Sigma}_r\boldsymbol{V}_r^T$.

SVD Solution of Linear Regression



Using the decomposition $\boldsymbol{X} = \boldsymbol{U}_r \boldsymbol{\Sigma}_r \boldsymbol{V}_r^T$, we can rewrite the goal equation as

$$\boldsymbol{V}_r \boldsymbol{\Sigma}_r^T \boldsymbol{U}_r^T \boldsymbol{U}_r \boldsymbol{\Sigma}_r \boldsymbol{V}_r^T \boldsymbol{w} = \boldsymbol{V}_r \boldsymbol{\Sigma}_r^T \boldsymbol{U}_r^T \boldsymbol{t}.$$

A transposition of an orthogonal matrix is its inverse. Therefore, our submatrix $\boldsymbol{U}_r$ fulfils that $\boldsymbol{U}_r^T \boldsymbol{U}_r = \boldsymbol{I}$, because $\boldsymbol{U}_r^T \boldsymbol{U}_r$ is a top left submatrix of $\boldsymbol{U}^T \boldsymbol{U}$. Analogously, $\boldsymbol{V}_r^T \boldsymbol{V}_r = \boldsymbol{I}$.

We therefore simplify the goal equation to

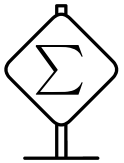
$$\boldsymbol{\Sigma}_r \boldsymbol{\Sigma}_r \boldsymbol{V}_r^T \boldsymbol{w} = \boldsymbol{\Sigma}_r \boldsymbol{U}_r^T \boldsymbol{t}$$

Because the diagonal matrix $\boldsymbol{\Sigma}_r$ is regular, we can divide by it and obtain

$$\boldsymbol{V}_r^T \boldsymbol{w} = \boldsymbol{\Sigma}_r^{-1} \boldsymbol{U}_r^T \boldsymbol{t}.$$

SVD Solution of Linear Regression

We have $\boldsymbol{V}_r^T \boldsymbol{w} = \boldsymbol{\Sigma}_r^{-1} \boldsymbol{U}_r^T \boldsymbol{t}$. If he original matrix $\boldsymbol{X}^T \boldsymbol{X}$ was regular, then $r = d$ and $\boldsymbol{V}_r$ is a square regular orthogonal matrix, in which case



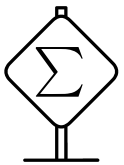
$$\boldsymbol{w} = \boldsymbol{V}_r \boldsymbol{\Sigma}_r^{-1} \boldsymbol{U}_r^T \boldsymbol{t}.$$

If we denote $\boldsymbol{\Sigma}^+ \in \mathbb{R}^{D \times N}$ the diagonal matrix with $\Sigma_{i,i}^{-1}$ on diagonal, we can rewrite to

$$\boldsymbol{w} = \boldsymbol{V} \boldsymbol{\Sigma}^+ \boldsymbol{U}^T \boldsymbol{t}.$$

Now if $r < d$, $\boldsymbol{V}_r^T \boldsymbol{w} = \boldsymbol{y}$ is undetermined and has infinitely many solutions. To find the one with smallest norm $||\boldsymbol{w}||$, consider the full product $\boldsymbol{V}^T \boldsymbol{w}$. Because $\boldsymbol{V}$ is orthogonal, $||\boldsymbol{V}^T \boldsymbol{w}|| = ||\boldsymbol{w}||$, and it is sufficient to find $\boldsymbol{w}$ with smallest $||\boldsymbol{V}^T \boldsymbol{w}||$. We know that the first r elements of $||\boldsymbol{V}^T \boldsymbol{w}||$ are fixed by the above equation – the smallest $||\boldsymbol{V}^T \boldsymbol{w}||$ can be therefore obtained by setting the last $d - r$ elements to zero. Finally, we note that $\boldsymbol{\Sigma}^+ \boldsymbol{U}^T \boldsymbol{t}$ is exactly $\boldsymbol{\Sigma}_r^{-1} \boldsymbol{U}_r^T \boldsymbol{t}$ padded with $d - r$ zeros, obtaining the same solution $\boldsymbol{w} = \boldsymbol{V} \boldsymbol{\Sigma}^+ \boldsymbol{U}^T \boldsymbol{t}$.

SVD Solution of Linear Regression and Pseudoinverses



The solution to a linear regression with sum of squares error function is tightly connected to matrix pseudoinverses. If a matrix $\boldsymbol{X}$ is singular or rectangular, it does not have an exact inverse, and $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{b}$ does not have an exact solution.

However, we can consider the so-called *Moore-Penrose pseudoinverse*

$$\boldsymbol{X}^+ \stackrel{\text{def}}{=} \boldsymbol{V}\boldsymbol{\Sigma}^+\boldsymbol{U}^T$$

to be the closest approximation to an inverse, in the sense that we can find the best solution (with smallest MSE) to the equation $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{b}$ by setting $\boldsymbol{w} = \boldsymbol{X}^+\boldsymbol{b}$.

Alternatively, we can define the pseudoinverse as

$$\boldsymbol{X}^+ = \arg\min_{\boldsymbol{Y}\in\mathbb{R}^{D\times N}} ||\boldsymbol{X}\boldsymbol{Y} - \boldsymbol{I}_N||_F = \arg\min_{\boldsymbol{Y}\in\mathbb{R}^{N\times D}} ||\boldsymbol{Y}\boldsymbol{X} - \boldsymbol{I}_D||_F$$

which can be verified to be the same as our SVD formula.

A random variable $\mathbf{x}$ is a result of a random process. It can be discrete or continuous.

Probability Distribution

A probability distribution describes how likely are individual values a random variable can take.

The notation $\mathbf{x} \sim P$ stands for a random variable $\mathbf{x}$ having a distribution P.

For discrete variables, the probability that $\mathbf{x}$ takes a value x is denoted as $P(x)$ or explicitly as $P(\mathbf{x} = x)$. All probabilities are non-negative and sum of probabilities of all possible values of $\mathbf{x}$ is $\sum_x P(\mathbf{x} = x) = 1$.

For continuous variables, the probability that the value of $\mathbf{x}$ lies in the interval $[a, b]$ is given by $\int_a^b p(x)\,\mathrm{d}x$.

Expectation

The expectation of a function $f(x)$ with respect to discrete probability distribution $P(x)$ is defined as:

$$\mathbb{E}_{\mathbf{x}\sim P}[f(x)] \stackrel{\text{def}}{=} \sum_x P(x)f(x)$$

For continuous variables it is computed as:

$$\mathbb{E}_{\mathbf{x}\sim p}[f(x)] \stackrel{\text{def}}{=} \int_x p(x)f(x)\,\mathrm{d}x$$

If the random variable is obvious from context, we can write only $\mathbb{E}_P[x]$ of even $\mathbb{E}[x]$.

Expectation is linear, i.e.,

$$\mathbb{E}_{\mathbf{x}}[\alpha f(x) + \beta g(x)] = \alpha\mathbb{E}_{\mathbf{x}}[f(x)] + \beta\mathbb{E}_{\mathbf{x}}[g(x)]$$

Variance

Variance measures how much the values of a random variable differ from its mean $\mu = \mathbb{E}[x]$.

$$\mathrm{Var}(x) \stackrel{\text{def}}{=} \mathbb{E}\left[\left(x - \mathbb{E}[x]\right)^2\right], \text{ or more generally}$$
$$\mathrm{Var}(f(x)) \stackrel{\text{def}}{=} \mathbb{E}\left[\left(f(x) - \mathbb{E}[f(x)]\right)^2\right]$$

It is easy to see that

$$\mathrm{Var}(x) = \mathbb{E}\left[x^2 - 2x\mathbb{E}[x] + \left(\mathbb{E}[x]\right)^2\right] = \mathbb{E}\left[x^2\right] - \left(\mathbb{E}[x]\right)^2,$$

because $\mathbb{E}\left[2x\mathbb{E}[x]\right] = 2(\mathbb{E}[x])^2$.

Variance is connected to $E[x^2]$, a *second moment* of a random variable – it is in fact a *centered* second moment.

Estimators and Bias

An *estimator* is a rule for computing an estimate of a given value, often an expectation of some random value(s).

For example, we might estimate *mean* of random variable by sampling a value according to its probability distribution.

Bias of an estimator is the difference of the expected value of the estimator and the true value being estimated:

$$\mathrm{bias} = \mathbb{E}[\mathrm{estimate}] - \mathrm{true\ estimated\ value}.$$

If the bias is zero, we call the estimator *unbiased*, otherwise we call it *biased*.

Estimators and Bias

If we have a sequence of estimates, it also might happen that the bias converges to zero. Consider the well known sample estimate of variance. Given $\mathbf{x}_1, \ldots, \mathbf{x}_n$ independent and identically distributed random variables, we might estimate mean and variance as

$$\hat{\mu} = \frac{1}{n}\sum\nolimits_i x_i, \quad \hat{\sigma}_2 = \frac{1}{n}\sum\nolimits_i (x_i - \hat{\mu})^2.$$

Such estimate is biased, because $\mathbb{E}[\hat{\sigma}^2] = (1 - \frac{1}{n})\sigma^2$, but the bias converges to zero with increasing n.

Also, an unbiased estimator does not necessarily have small variance – in some cases it can have large variance, so a biased estimator with smaller variance might be preferred.

Gradient Descent

Sometimes it is more practical to search for the best model weights in an iterative/incremental/sequential fashion. Either because there is too much data, or the direct optimization is not feasible.

Assuming we are minimizing an error function

$$\arg\min_{\boldsymbol{w}} E(\boldsymbol{w}),$$

we may use *gradient descent*:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} E(\boldsymbol{w})$$

The constant α is called a *learning rate* and specifies the "length" of a step we perform in every iteration of the gradient descent.

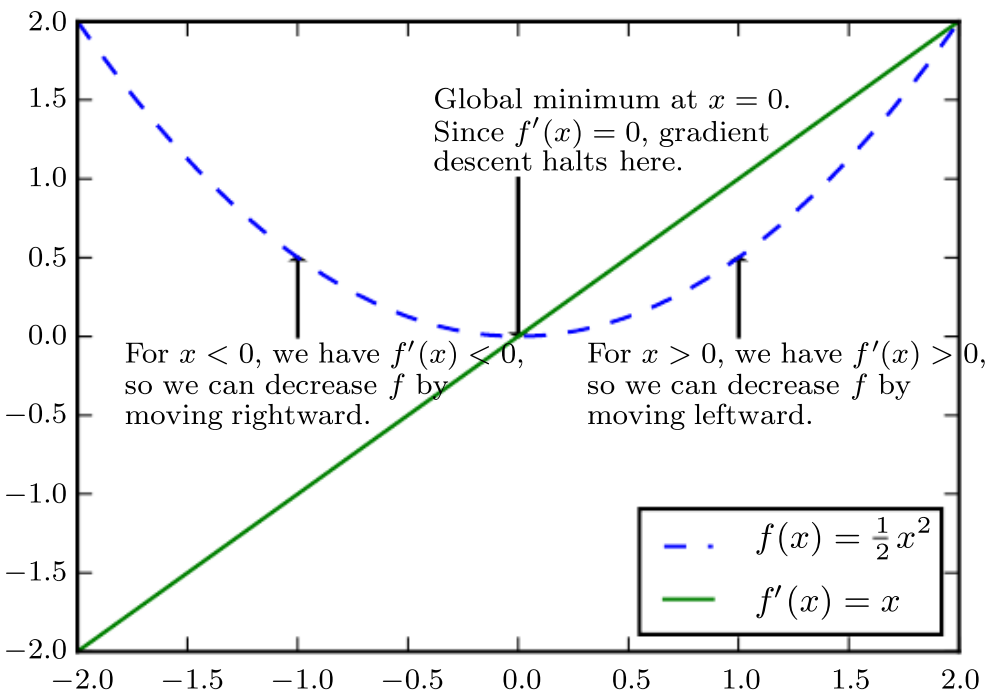


Figure 4.1, page 83 of Deep Learning Book, http://deeplearningbook.org

Gradient Descent Variants

Consider an error function computed as an expectation over the dataset:

$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) = \nabla_{\boldsymbol{w}} \mathbb{E}_{(\boldsymbol{x},t)\sim\hat{p}_{\text{data}}} L\big(f(\boldsymbol{x};\boldsymbol{w}),t\big).$$

- **(Regular) Gradient Descent**: We use all training data to compute $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$ exactly.
- **Online (or Stochastic) Gradient Descent**: We estimate $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$ using a single random example from the training data. Such an estimate is unbiased, but very noisy.

$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) \approx \nabla_{\boldsymbol{w}} L\big(f(\boldsymbol{x};\boldsymbol{w}),t\big) \text{ for randomly chosen } (\boldsymbol{x},t) \text{ from } \hat{p}_{\text{data}}.$$

- **Minibatch SGD**: The minibatch SGD is a trade-off between gradient descent and SGD – the expectation in $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$ is estimated using m random independent examples from the training data.

$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) \approx \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{w}} L\big(f(\boldsymbol{x}_i;\boldsymbol{w}),t_i\big) \text{ for randomly chosen } (\boldsymbol{x}_i,t_i) \text{ from } \hat{p}_{\text{data}}.$$

Gradient Descent Convergence

Assume that we perform a stochastic gradient descent, using a sequence of learning rates α_i, and using a noisy estimate $J(\boldsymbol{w})$ of the real gradient $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$:

$$\boldsymbol{w}_{i+1} \leftarrow \boldsymbol{w}_i - \alpha_i J(\boldsymbol{w}_i).$$

It can be proven (under some reasonable conditions; see Robbins and Monro algorithm, 1951) that if the loss function L is convex and continuous, then SGD converges to the unique optimum almost surely if the sequence of learning rates α_i fulfills the following conditions:

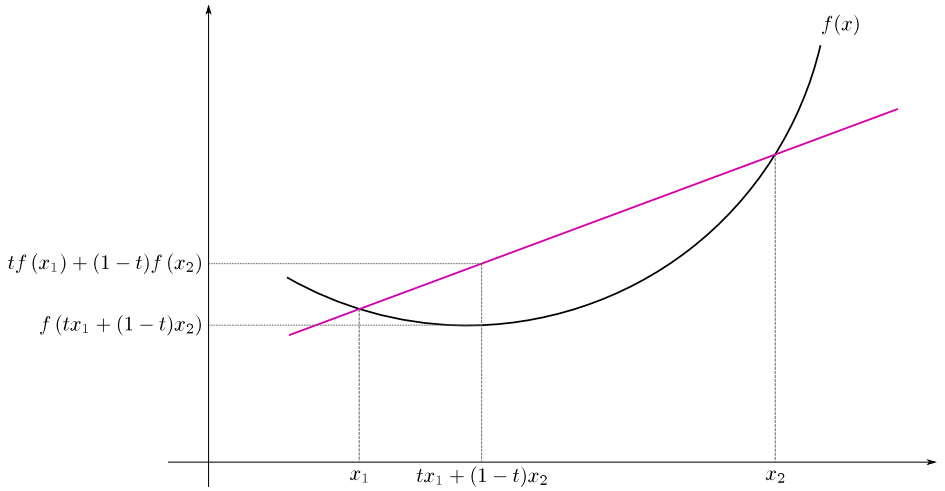
$$\alpha_i \to 0, \quad \sum_i \alpha_i = \infty, \quad \sum_i \alpha_i^2 < \infty.$$

For non-convex loss functions, we can get guarantees of converging to a *local* optimum only. However, note that finding a global minimum of an arbitrary function is *at least NP-hard*.

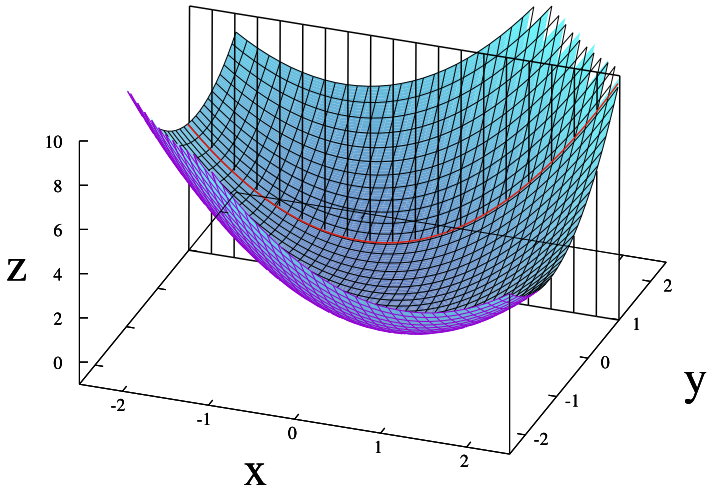
Gradient Descent Convergence

Convex functions mentioned on a previous slide are such that for x_1, x_2 and real $0 \leq t \leq 1$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2).$$



https://upload.wikimedia.org/wikipedia/commons/c/c7/ConvexFunction.svg



https://commons.wikimedia.org/wiki/File:Partial_func_eg.svg

A twice-differentiable function is convex iff its second derivative is always non-negative.

A local minimum of a convex function is always the unique global minimum.

Well-known examples of convex functions are x^2, e^x and $-\log x$.

Gradient Descent of Linear Regression

For linear regression and sum of squares, using online gradient descent we can update the weights as

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} E(\boldsymbol{w}) = \boldsymbol{w} - \alpha(\boldsymbol{x}^T \boldsymbol{w} - t)\boldsymbol{x}.$$

Input: Dataset ($\boldsymbol{X} \in \mathbb{R}^{N \times D}$, $\boldsymbol{t} \in \mathbb{R}^N$), learning rate $\alpha \in \mathbb{R}^+$.
Output: Weights $\boldsymbol{w} \in \mathbb{R}^D$ which hopefully minimize MSE of linear regression.

- $\boldsymbol{w} \leftarrow 0$
- repeat until convergence:
  - for $i = 1, \ldots, n$:
    - $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha(\boldsymbol{x}_i^T \boldsymbol{w} - t_i)\boldsymbol{x}_i$.

Features

Note that until now, we did not explicitly distinguished *input* instance values and instance *features*.

The *input* instance values are usually the raw observations and are given. However, we might extend them suitably before running a machine learning algorithm, especially if the algorithm is linear or otherwise limited and cannot represent arbitrary function.

We already saw this in the example from the previous lecture, where even if our training examples were x and t, we performed the linear regression using features $(x^0, x^1, \ldots, x^M)$:

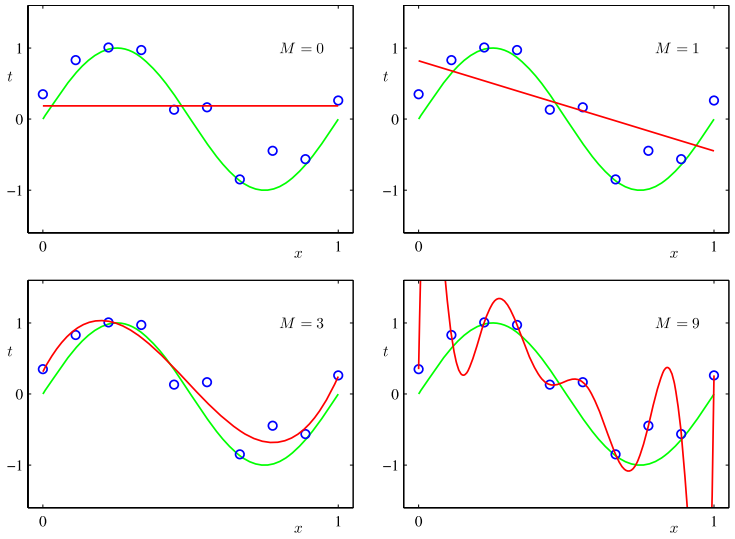


Figure 1.4 of Pattern Recognition and Machine Learning.

Features

Generally, it would be best if we have machine learning algorithms processing only the raw inputs. However, many algorithms are capable of representing only a limited set of functions (for example linear ones), and in that case, *feature engineering* plays a major part in the final model performance. Feature engineering is a process of constructing features from raw inputs.

Commonly used features are:

- **polynomial features** of degree p: Given features $(x_1, x_2, \ldots, x_D)$, we might consider *all* products of p input values. Therefore, polynomial features of degree 2 would consist of $x_i^2 \forall i$ and of $x_i x_j \forall i \neq j$.
- **categorical one-hot features**: Assume for example that a day in a week is represented on the input as an integer value of 1 to 7, or a breed of a dog is expressed as an integer value of 0 to 366. Using these integral values as input to linear regression makes little sense – instead it might be better to learn weights for individual days in a week or for individual dog breeds. We might therefore represent input classes by binary indicators for every class, giving rise to **one-hot** representation, where input integral value $1 \leq v \leq L$ is represented as L binary values, which are all zero except for the v-th one, which is one.

Cross-Validation

We already talked about a **train set** and a **test set**. Given that the main goal of machine learning is to perform well on unseen data, the test set must not be used during training nor hyperparameter selection. Ideally, it is hidden to us altogether.

Therefore, to evaluate a machine learning model (for example to select model architecture, input features, or hyperparameter value), we normally need the **validation** or a **development** set.

However, using a single development set might give us noisy results. To obtain less noisy results (i.e., with smaller variance), we can use **cross-validation**.

In cross-validation, we choose multiple validation sets from the training data, and for every one, we train a model on the rest of the training data and evaluate on the chosen validation sets. A commonly used strategy to choose the validation sets is called **k-fold cross-validation**. Here the training set is partitioned into k subsets of approximately the same size, and each subset takes turn to play a role of a validation set.

https://commons.wikimedia.org/wiki/File:K-fold_cross_validation_EN.svg

Binary Classification

Binary classification is a classification in two classes.

To extend linear regression to binary classification, we might seek a *threshold* and the classify an input as negative/positive depending whether $\boldsymbol{x}^T \boldsymbol{w}$ is smaller/larger than a given threshold.

Zero value is usually used as the threshold, both because it is symmetric and also because the *bias* parameter acts as a trainable threshold anyway.

Perceptron

The perceptron algorithm is probably the oldest one for training weights of a binary classification. Assuming the target value $t \in \{-1, +1\}$, the goal is to find weights $\boldsymbol{w}$ such that for all train data

$$\operatorname{sign}(\boldsymbol{w}^T \boldsymbol{x}_i) = t_i,$$

or equivalently

$$t_i \boldsymbol{w}^T \boldsymbol{x}_i > 0.$$

Note that a set is called **linearly separable**, if there exist a weight vector $\boldsymbol{w}$ such that the above equation holds.

Perceptron

The perceptron algorithm was invented by Rosenblat in 1958.

Input: Linearly separable dataset ($\boldsymbol{X} \in \mathbb{R}^{N \times D}$, $\boldsymbol{t} \in \{-1, +1\}$).
Output: Weights $\boldsymbol{w} \in \mathbb{R}^D$ such that $t_i \boldsymbol{x}_i^T \boldsymbol{w} > 0$ for all i.

- $\boldsymbol{w} \leftarrow 0$
- until all examples are classified correctly:
  - for i in $1, \ldots, N$:
    - $y \leftarrow \boldsymbol{w}^T \boldsymbol{x}_i$
    - if $t_i y \leq 0$ (incorrectly classified example):
      - $\boldsymbol{w} \leftarrow \boldsymbol{w} + t_i \boldsymbol{x}_i$

We will prove that the algorithm always arrives at some correct set of weights $\boldsymbol{w}$ if the training set is linearly separable.

Perceptron as SGD

Consider the main part of the perceptron algorithm:

- $y \leftarrow \boldsymbol{w}^T \boldsymbol{x}_i$
- if $t_i y \leq 0$ (incorrectly classified example):
  - $\boldsymbol{w} \leftarrow \boldsymbol{w} + t_i \boldsymbol{x}_i$

We can derive the algorithm using on-line gradient descent, using the following loss function

$$L(f(\boldsymbol{x}; \boldsymbol{w}), t) \stackrel{\text{def}}{=} \begin{cases} -t\boldsymbol{x}^T \boldsymbol{w} & \text{if } t\boldsymbol{x}^T \boldsymbol{w} \leq 0 \\ 0 & \text{otherwise} \end{cases} = \max(0, -t\boldsymbol{x}^T \boldsymbol{w}) = \mathrm{ReLU}(-t\boldsymbol{x}^T \boldsymbol{w}).$$

In this specific case, the value of the learning rate does not actually matter, because multiplying $\boldsymbol{w}$ by a constant does not change a prediction.