**The Machine Learning Setup**

* Data scientists use supervised machine learning to create computer programs that learn from past data
* To learn from data, you must differentiate between what you know, the *features* (x), and what you would like to infer, the *label* (y)

The purpose of machine learning is to make decisions from data. Following the approach of traditional computer science, one might be tempted to write a carefully designed program that follows some rules to make these decisions. Instead of writing a program in this traditional way, however, data scientists use supervised machine learning to create a computer program that is *learned* from past data.



To learn from data, you must differentiate between what you know, the **features** (xx), and what you would like to infer, the **label** (yy). For example, your features could describe a patient in a hospital (e.g., gender, age, body temperature, various symptoms) and the label could be if the patient is sick or healthy. You can use data from past medical records to learn a **function**(hh**)** that is able to determine a future patient's diagnosis based on their symptoms.

For an incoming patient, when you observe features (xx), you can apply the function (hh) to predict whether this new patient is sick or healthy (yy). The initial stage where you use existing medical records to learn a function is called the *training stage,*and the latter where you apply the function to a new patient is called the *testing stage*.

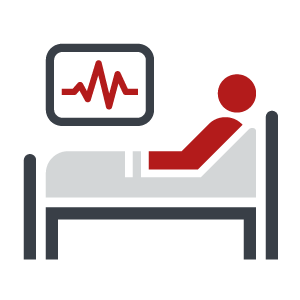
**The Hypothesis of a Problem**

The function, often referred to as *hypothesis* and denoted as hh, is the program that is learned from the data. You differentiate between the *hypothesis class*, which is the set of all possible functions that could be learned by the algorithm, and the*final hypothesis*, which is obtained after learning from our training data. Many possible hypothesis classes exist and they loosely correspond to different types of learning algorithms. All machine learning algorithms function differently and will have a variety of parameters for their hypothesis class. It is your job as a data scientist to identify which algorithm is most suitable for a given learning problem.

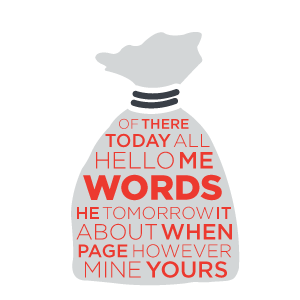
**The Features of a Problem**

* Features are the relevant characteristics or attributes that you believe may contribute to the outcome of an event
* The examples of feature vectors provided are bag-of-words features, pixel features, and heterogeneous features
* How a data instance is encoded into a vector and what data the vector contains will usually influence the outcome of the machine learning process

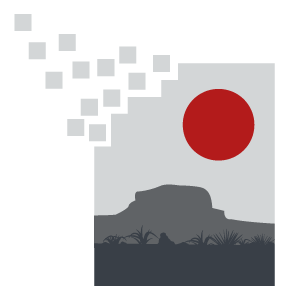
Features are the relevant characteristics or attributes that you believe may contribute to the outcome of an event. You use features to describe the data from which you are trying to make predictions. Throughout this course, you should assume that features are stored as a dd-dimensional vector of feature values. How a data instance is encoded into a vector and what data the vector contains will usually influence the outcome of the machine learning process. Typically, here is where domain expertise can be helpful.



**Heterogeneous features:** Patient data in medical applications typically contain many heterogeneous features. For example, these could include the patient's age in years, blood pressure, and height in centimeters. Here, each feature has its own unit and can vary widely in range and distribution. When working with heterogeneous data, a data scientist has to keep in mind that some feature values can be much larger than others and must choose algorithms that can appropriately cope with such variety in scale. For instance, blood pressure and height are of different scales; a blood pressure reading that is 10 units higher than the average and a height measurement of 10 units higher than the average have very different implications. If this difference in scale is not taken into account, it can easily happen that the algorithm ignores some important features, only because the differences are of very small magnitude.



**Bag-of-words features:** Text documents are often stored as bag-of-words features. This is a method to convert a text document with any number of words into a feature vector of fixed dimensionality. Before learning begins, one agrees on a finite set of possible words of interest, such as the dd =100,000 most common words in the English language. The text document is then scanned for these words and represented as a vector of word counts. In other words, the ithith dimension of the feature vector stores how many times the ithith word appears in this text document. Here, a text document is treated as a set of words whose order is disregarded for representational convenience. As most words in the English language are not present in any given document, the vector will consist mostly of zeros. A feature vector that contains far more zeros than non-zero entries is referred to as “sparse.”



**Pixel features:** Images are typically stored as pixels. These can be represented as a vector by simply “vectorizing” the image in one long chain of numbers. If an image has six megapixels, and each pixel has three numbers (one for red, green, blue) this yields an 18 million dimensional vector. Colors are typically stored by their saturation, ranging from zero to 255, so all feature values are non-negative and within that range. More specialized algorithms (such as convolutional neural networks) will preserve the 2D grid structure of an image and represent images as three saturation matrices, one for each color channel.

**The Labels of a Problem**

The label (yy) is what you want to predict for a given data instance. Labels can come in many different forms, but throughout this course we will only distinguish three different cases:

|  |  |
| --- | --- |
|  | **Binary**  **There are only two possible label values.** For example, with spam email classification, an email is either spam or not spam. Spam could be mapped to "+1" and email not considered spam to "-1." |
|  | **Multiclass**  **There are multiple distinct label values.**For example, in a facial recognition application, you would distinguish each individual as a separate class, such as:   * class1=”Bill Gates” * class2=”Steve Jobs” * class3=”Linus Torvalds” |
|  | **Regression**  **There are infinite possible label values.** For example, if you want to predict for how much a particular house will sell, the label (the sale price) could be any non-negative value. |

**Regression vs. Multiclass Classification**

Some problems could be cast as either regression or multiclass; however, there is usually a most natural choice.

For example, assume you are trying to predict the height (label) of a personbased on other data (features), such as their gender, weight, age, etc. If you cast this as a multiclass classification problem, you might define each class as a rounded incremental value of 1cm. If we try to classify someone who is actually 183cm tall and return a label of 182cm, we are just as wrong as if we had predicted 140cm. Most people would consider a predicted height of 182cm to be much closer to the truth (183cm) than 140cm is, but since each height is its own class and there is typically no assumption that one class is more similar to another in classification problems, we’ve simply failed to predict the height correctly. For this example, we should instead choose regression. In regression, we typically assume that, because the label can be any real value, you will almost never hit it exactly but hope to get very close.

An example problem where multiclass classification would be the best choice is facial recognition. Bill Gates, Steve Jobs, and Linus Torvalds would each be their own class. We see here that each class is separate and distinct; selecting the wrong class is simply an incorrect label. There is no notion of similarity between classes since you are either right or wrong. For image classification problems, multiclass classification is the appropriate choice.

## Formalizing the Machine Learning Setup

To formalize the supervised machine learning setup, let's first look at how we express features and labels for our training data. Our training data comes in pairs of inputs and labels (x,y)(x,y), where x∈Rdx∈Rd is the input instance and yy is its label. The data is denoted as D={(x1,y1),…,(xn,yn)}⊆Rd×CD={(x1,y1),…,(xn,yn)}⊆Rd×C where:

xixi: input vector of the ithith sample (e.g., we refer to the athath feature dimension of xixi as [xi]a[xi]a)

yiyi: label of the ithith sample

RdRd: d-dimensional feature space

CC: label space

### ****Examples of label space:****

|  |  |  |
| --- | --- | --- |
| **Binary classification** | C={0,1}C={0,1} or  C={−1,+1}C={−1,+1} | Example: spam filtering. An email is either spam (+1) or not spam (-1). |
| **Multiclass classification** | C={0,1,...,K−1}C={0,1,...,K−1} or  C={1,2,⋯,K}C={1,2,⋯,K}  (K≥2)(K≥2) | Example: facial recognition. A person can be exactly one of KK identities (e.g., 1="Bill Gates", 2="Steve Jobs", etc.). |
| **Regression** | C=R+C=R+ | Example: future sale price of a house. A house could sell for any positive value. |

Given a training data set D of label and feature pairs, the ultimate goal of supervised machine learning is to find a function h:Rd→Ch:Rd→C, such that it can predict the label of data points that are not in the training set (we call this testing data):

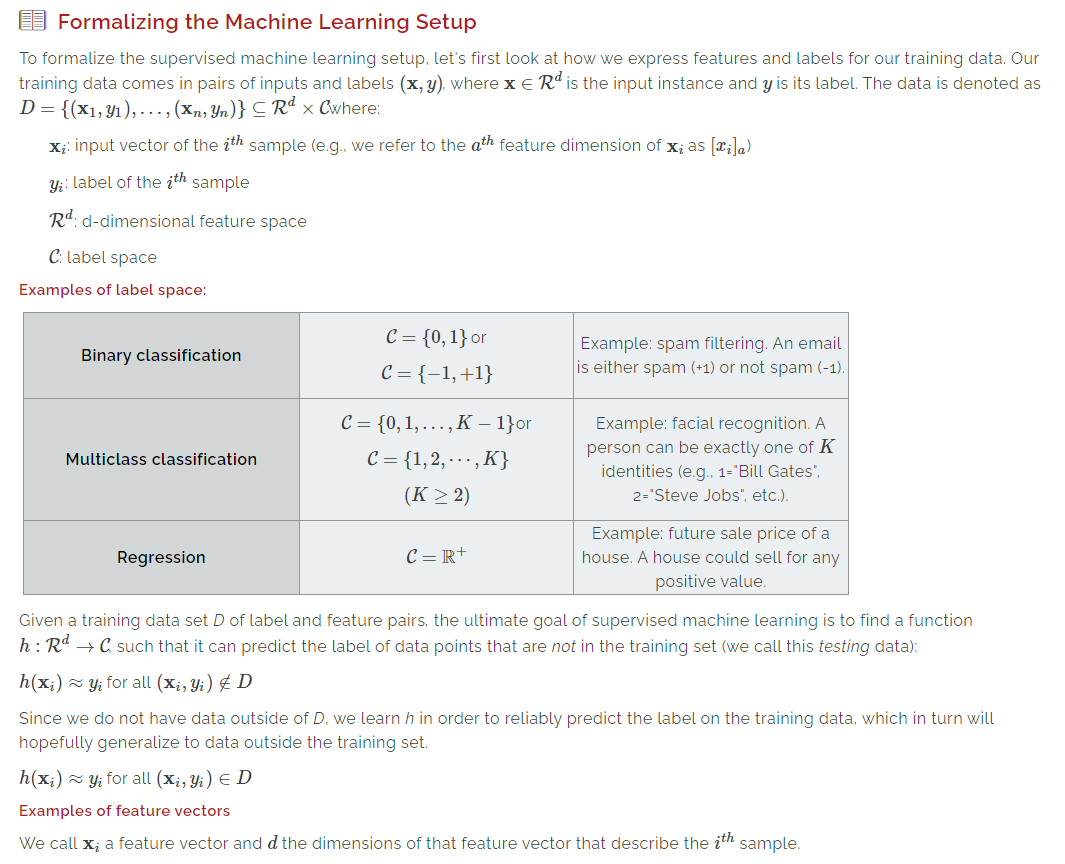
h(xi)≈yih(xi)≈yi for all (xi,yi)∉D(xi,yi)∉D

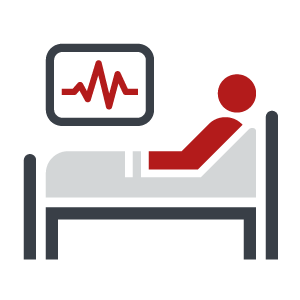
Since we do not have data outside of D, we learn h in order to reliably predict the label on the training data, which in turn will hopefully generalize to data outside the training set.

h(xi)≈yih(xi)≈yi for all (xi,yi)∈D(xi,yi)∈D

### Examples of feature vectors

We call xixi a feature vector and dd the dimensions of that feature vector that describe the ithith sample



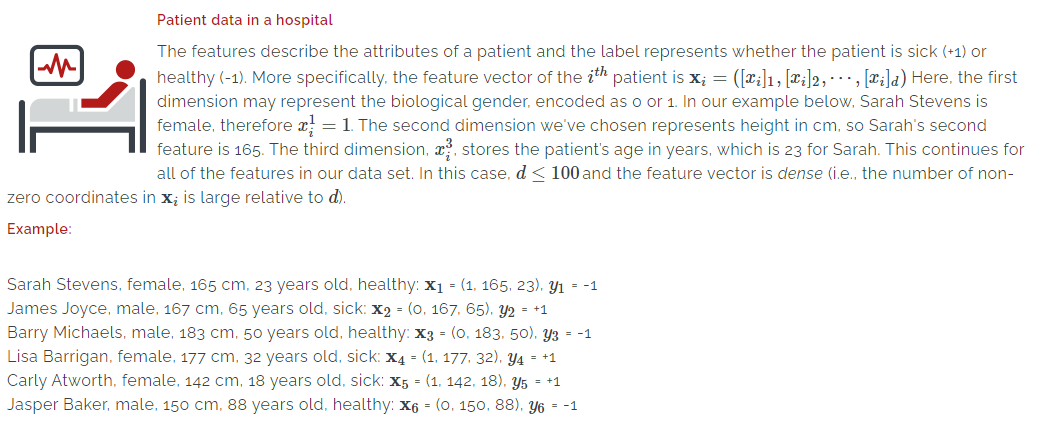


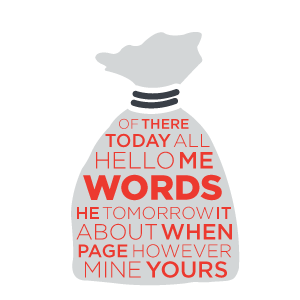
#### **Patient data in a hospital**

The features describe the attributes of a patient and the label represents whether the patient is sick (+1) or healthy (-1). More specifically, the feature vector of the ithith patient is xi=([xi]1,[xi]2,⋯,[xi]d)xi=([xi]1,[xi]2,⋯,[xi]d). Here, the first dimension may represent the biological gender, encoded as 0 or 1. In our example below, Sarah Stevens is female, therefore x1i=1xi1=1. The second dimension we've chosen represents height in cm, so Sarah's second feature is 165. The third dimension, x3ixi3, stores the patient’s age in years, which is 23 for Sarah. This continues for all of the features in our data set. In this case, d≤100d≤100 and the feature vector is *dense* (i.e., the number of non-zero coordinates in xixi is large relative to dd).

#### Example:

Sarah Stevens, female, 165 cm, 23 years old, healthy: x1x1 = (1, 165, 23), y1y1 = -1  
James Joyce, male, 167 cm, 65 years old, sick: x2x2 = (0, 167, 65), y2y2 = +1  
Barry Michaels, male, 183 cm, 50 years old, healthy: x3x3 = (0, 183, 50), y3y3 = -1  
Lisa Barrigan, female, 177 cm, 32 years old, sick: x4x4 = (1, 177, 32), y4y4 = +1  
Carly Atworth, female, 142 cm, 18 years old, sick: x5x5 = (1, 142, 18), y5y5 = +1  
Jasper Baker, male, 150 cm, 88 years old, healthy: x6x6 = (0, 150, 88), y6y6 = -1





#### **Text document in bag-of-words format**

The feature vector represents a text document — an article, for example — and the label corresponds to a category (e.g., sports, politics, entertainment*)*. The ithith document is represented as the vector xi=([xi]1,[xi]2,⋯,[xi]d)xi=([xi]1,[xi]2,⋯,[xi]d). Each dimension represents the exact number of occurrences of one particular word (e.g., xjixij is the number of occurrences of the jthjth word in document ii). In this case, d∼100000−10Md∼100000−10M and the feature vector is sparse (i.e., xixi consists mostly of zeros).

#### Example

Consider this corpus, a portion of Charles Dickens' "A Tale of Two Cities." For this example, let's imagine that each line is a distinct document:

It was the best of times,  
it was the worst of times,  
it was the age of wisdom,  
it was the age of foolishness,

The vocabulary is every unique word in the corpus:

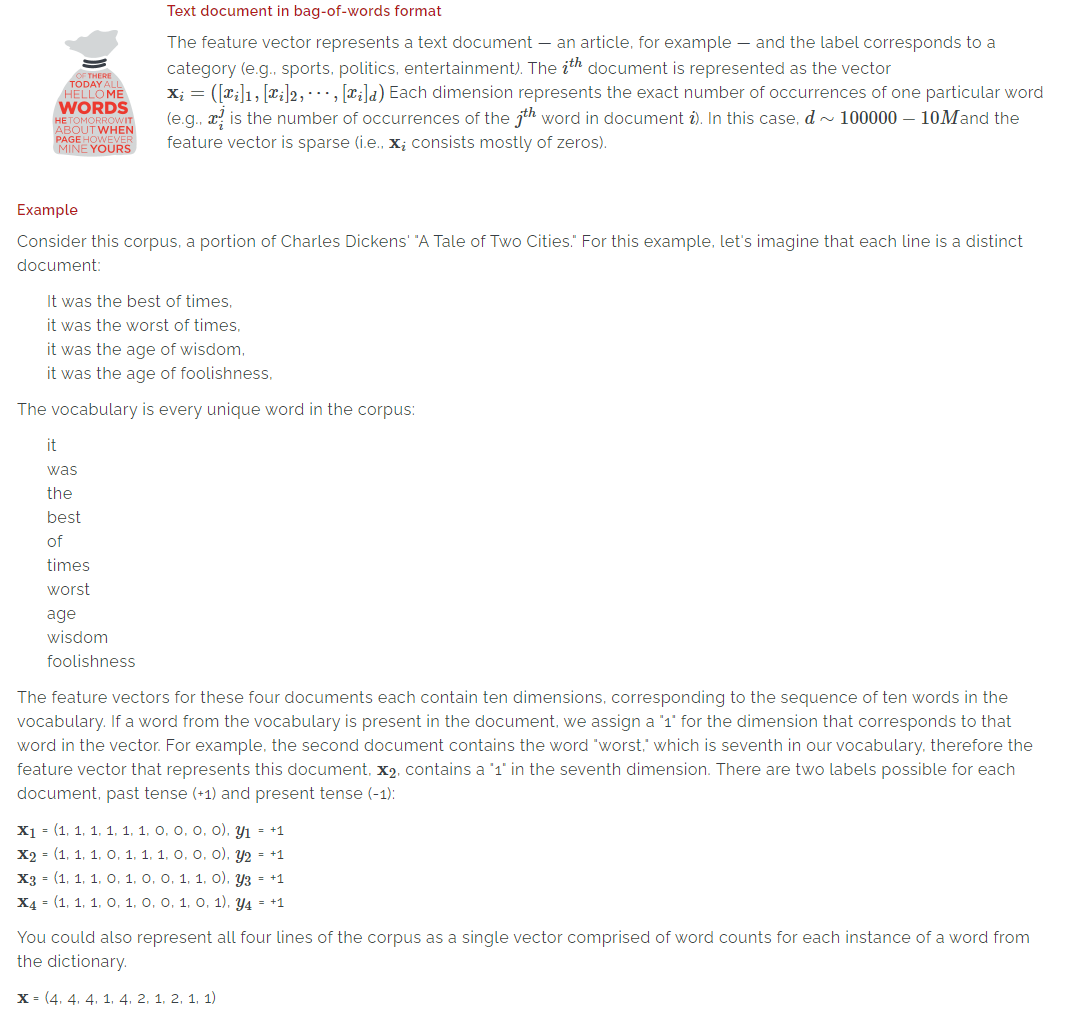
it  
was  
the  
best  
of  
times  
worst  
age  
wisdom  
foolishness

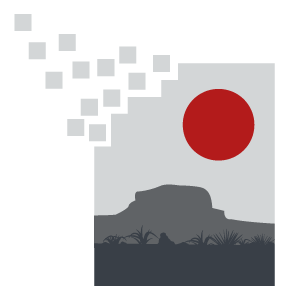
The feature vectors for these four documents each contain ten dimensions, corresponding to the sequence of ten words in the vocabulary. If a word from the vocabulary is present in the document, we assign a "1" for the dimension that corresponds to that word in the vector. For example, the second document contains the word "worst," which is seventh in our vocabulary, therefore the feature vector that represents this document, x2x2, contains a "1" in the seventh dimension. There are two labels possible for each document, past tense (+1) and present tense (-1):

x1x1 = (1, 1, 1, 1, 1, 1, 0, 0, 0, 0), y1y1 = +1  
x2x2 = (1, 1, 1, 0, 1, 1, 1, 0, 0, 0), y2y2 = +1  
x3x3 = (1, 1, 1, 0, 1, 0, 0, 1, 1, 0), y3y3 = +1  
x4x4 = (1, 1, 1, 0, 1, 0, 0, 1, 0, 1), y4y4 = +1

You could also represent all four lines of the corpus as a single vector comprised of word counts for each instance of a word from the dictionary.

xx = (4, 4, 4, 1, 4, 2, 1, 2, 1, 1)





#### **Images in pixel space**

The features correspond to images in raw pixel format and the label can correspond to an image category. For example, the image may be of a person and the label may be the person's name (e.g., “Bill Gates”). Here, every input dimension corresponds to a color of a particular pixel: xi=([xi]1,[xi]2,⋯,[xi]3k)xi=([xi]1,[xi]2,⋯,[xi]3k), where x3j−2ixi3j−2, x3j−1ixi3j−1, and x3jixi3j refer to the red, green, and blue values of the jjth pixel in the image.

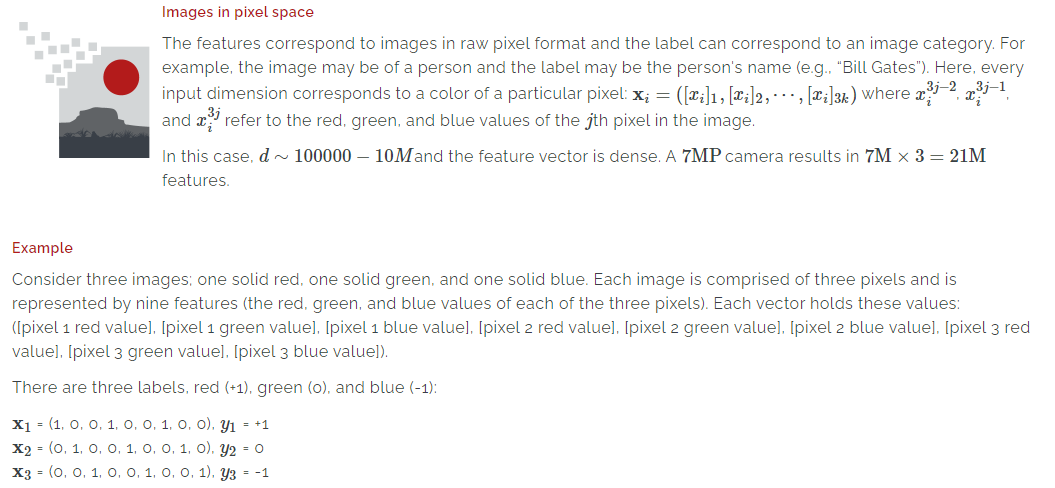
In this case, d∼100000−10Md∼100000−10M and the feature vector is dense. A 7MP7MP camera results in 7M×3=21M7M×3=21M features.

#### Example

Consider three images; one solid red, one solid green, and one solid blue. Each image is comprised of three pixels and is represented by nine features (the red, green, and blue values of each of the three pixels). Each vector holds these values:  
([pixel 1 red value], [pixel 1 green value], [pixel 1 blue value], [pixel 2 red value], [pixel 2 green value], [pixel 2 blue value], [pixel 3 red value], [pixel 3 green value], [pixel 3 blue value]).

There are three labels, red (+1), green (0), and blue (-1):

x1x1 = (1, 0, 0, 1, 0, 0, 1, 0, 0), y1y1 = +1  
x2x2 = (0, 1, 0, 0, 1, 0, 0, 1, 0), y2y2 = 0  
x3x3 = (0, 0, 1, 0, 0, 1, 0, 0, 1), y3y3 = -1



**Selecting a Loss Function**

* There are typically two steps in learning a hypothesis function: selecting the algorithm and finding the best function within the class of possible functions
* A loss function evaluates a hypothesis on our data and tells us how good or bad it is, helping us choose the best function
* Three examples of loss function are zero-one, squared, and absolute losses

There are typically two steps involved in learning a hypothesis function:

1. Selecting the appropriate algorithm for the problem
2. Finding the best function from all possible functions

**Select the Appropriate Algorithm**

First, we select the type of machine learning algorithm that we think is appropriate for this particular learning problem. This defines the hypothesis class HH (i.e., the type of function we would like to learn).

**Find the Best Function**

Second, we find the best function within the hypothesis class. This second step is the actual learning process and often — but not always — involves an optimization problem. Essentially, we try to find a function h∈Hh∈H within the hypothesis class that makes the fewest mistakes on our training data.

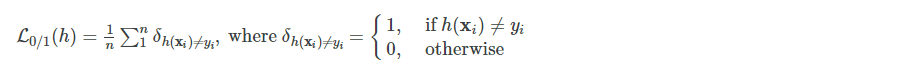
How can we find the best function? For this, we need some way to evaluate what it means for one function to be *better* than another. This is where the **loss function** comes in. A loss function evaluates a hypothesis on our training data and tells us how bad it is. The higher the loss, the worse it is — a loss of zero means it makes perfect predictions. It is common practice to divide the loss by the total number of training samples, *n*, so that the output can be interpreted as the average loss per sample (and is independent of *n*).

**Examples of Loss Functions**

**Zero-One Loss**

The simplest loss function is the zero–one loss. It literally counts how many mistakes a hypothesis function makes on a particular data set. For every single example that is predicted incorrectly, it suffers a loss of 1. The normalized zero–one loss returns the fraction of misclassified training samples, also referred to as the training error. The zero-one loss is often used to evaluate classifiers in multiclass/binary classification settings but rarely useful to guide optimization procedures because the function is non-differentiable and non-continuous.

Formally, the zero-one loss can be stated as:



**Squared Loss**

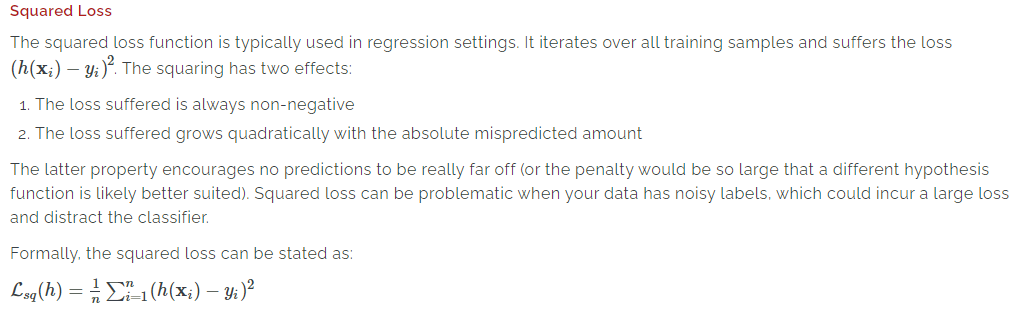
The squared loss function is typically used in regression settings. It iterates over all training samples and suffers the loss (h(xi)−yi)2(h(xi)−yi)2. The squaring has two effects:

1. The loss suffered is always non-negative
2. The loss suffered grows quadratically with the absolute mispredicted amount

The latter property encourages no predictions to be really far off (or the penalty would be so large that a different hypothesis function is likely better suited). Squared loss can be problematic when your data has noisy labels, which could incur a large loss and distract the classifier.

Formally, the squared loss can be stated as:

Lsq(h)=1n∑ni=1(h(xi)−yi)2Lsq(h)=1n∑i=1n(h(xi)−yi)2

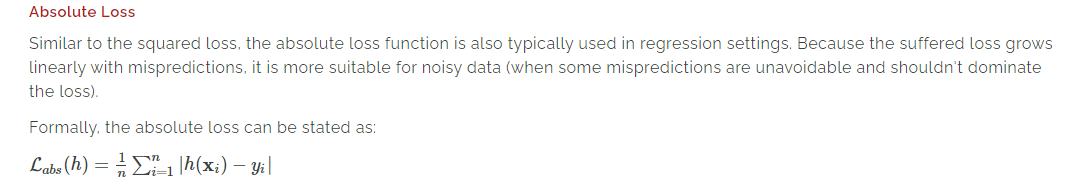


**Absolute Loss**

Similar to the squared loss, the absolute loss function is also typically used in regression settings. Because the suffered loss grows linearly with mispredictions, it is more suitable for noisy data (when some mispredictions are unavoidable and shouldn't dominate the loss).

Formally, the absolute loss can be stated as:

Labs(h)=1n∑ni=1|h(xi)−yi|



## Splitting Data Sets

To avoid overfitting to the training set, you will usually split the data DD Into three mutually exclusive subsets:



DTRDTR - training data (80%)

DVADVA - validation data (10%)

DTEDTE - test data (10%)

A common choice is to split the data 80/10/10, respectively. You then choose a function based on the training data, improve it using the validation data, and evaluate it on the test data.

### How Do You Split a Data Set?

One has to be careful when splitting the training data into the three sets. All three sets should be drawn from the same distribution. A good rule of thumb is to make sure that the data sets simulate the real-life settings in which the algorithm will operate. The most common way of splitting the data is uniformly at random. However, if the data has a temporal component and changes over time, you should split the data by time to make sure that you never predict the past from the future and always the future from the past.

An example of such a data set with a temporal component is email spam. Spam emails change over time, as spammers adapt their emails to get past spam filters. What makes spam filtering so hard is that you don’t know what clever tricks the spammers will deploy tomorrow. To address this problem, you could collect data for a few weeks to train your function. Then, collect data for a few days to use as your DVADVA data set to validate your function. Finally, collect data for a few more days to generate your DTEDTE data set to test your function. This process is essentially simulating the real-world setting that a spam filter trained today should catch the spam emails of tomorrow.

### Why Is Validation Data Important?

A common scenario is that the first machine learning algorithm that is trained on DTRDTR does not perform well enough on DTEDTE and needs further refinement. For example, imagine you are building an email spam filter with the objective to catch at least 99% of all spam, with at most 1/1000 false positives. If the initial classifier is not accurate enough for your needs, you may have to use a different algorithm or change its hyperparameters. The problem is that you cannot tweak your algorithm to perform well on DTEDTE, otherwise you would be overfitting your function to this specific data set and not necessarily improving its accuracy on new data. The error obtained on DTEDTE is only an unbiased estimate of the true generalization error of the model if the model was trained independently of this test set. The moment you look at the test data once and make changes to the algorithm, it is no longer independent. This is where the validation data set comes in.

The validation data set is a proxy for the test set. In practice, you train your algorithm on the training set and evaluate it on the validation set. If your function is not satisfactorily accurate, you continue tweaking it until the validation error improves to an acceptable level. Then, complete a single and final evaluation of your function on the test set DTEDTE to find the unbiased estimate of the generalization error of your final model. Often, the final model is re-trained on the union of DTRDTR and DVADVA so as to not waste the data points in the validation set.

### Formalized Process of Training and Evaluation

#### Training the Function

Choose a function (hh) to minimize the training loss:

h∗(⋅)=argminh∈HϵTR(h)h∗(⋅)=argminh∈HϵTR(h) ϵTR(h)=1|DTR|∑(x,y)∈DTRℓ(x,y|h(⋅))ϵTR(h)=1|DTR|∑(x,y)∈DTRℓ(x,y|h(⋅))

Here, HH is the hypothetical class (i.e., the set of all possible classifiers h(⋅)h(⋅)). In other words, you are trying to find a hypothesis hh which would have resulted in the lowest possible value (argminargmin) of the loss function (ℓℓ) averaged over all the points in the training set.

#### Evaluating the Function

Evaluate the function through the testing loss:

ϵTE(h)=1|DTE|∑(x,y)∈DTEℓ(x,y|h∗(⋅))ϵTE(h)=1|DTE|∑(x,y)∈DTEℓ(x,y|h∗(⋅))

The function ϵϵ is the average of the loss function over all points in the test set.

#### Generalization

The testing loss is an unbiased estimator (i.e., an approximation) of the **generalization loss** — the loss over unseen data. Minimizing the generalization loss is our true objective but cannot be done directly:

ϵ(h)=E(x,y)∼P[ℓ(x,y|h∗(⋅))]

