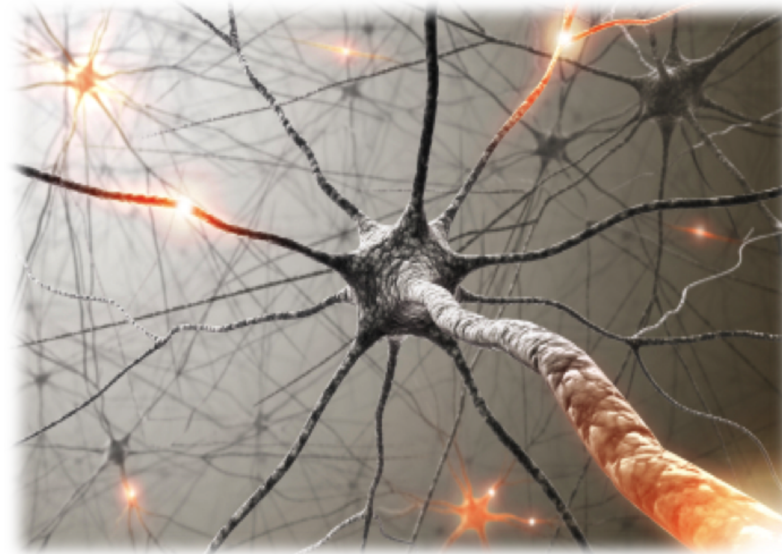
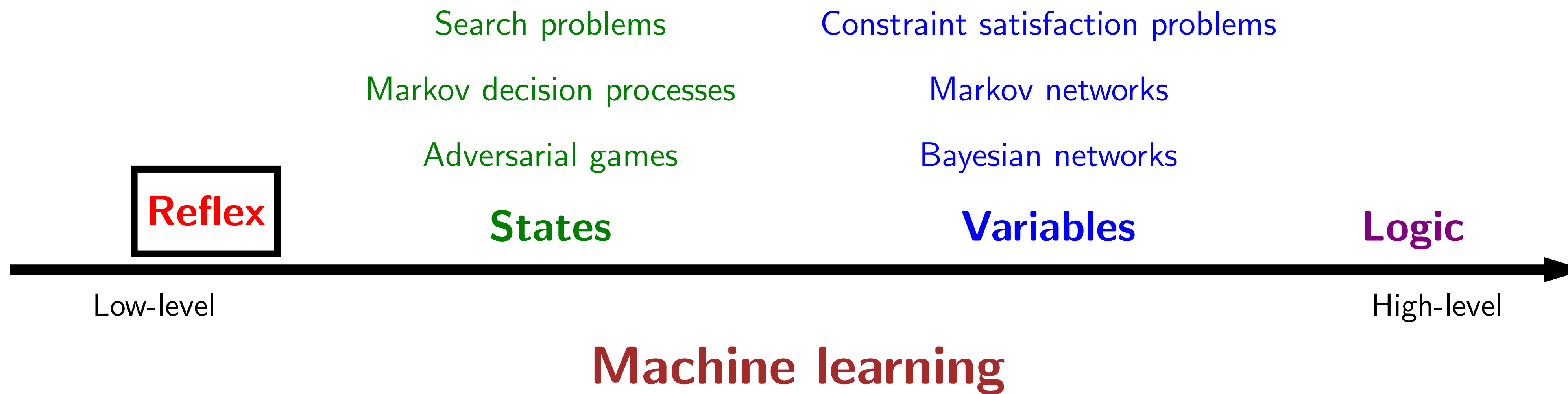
Lecture 2: Machine Learning 1



Course plan



Roadmap

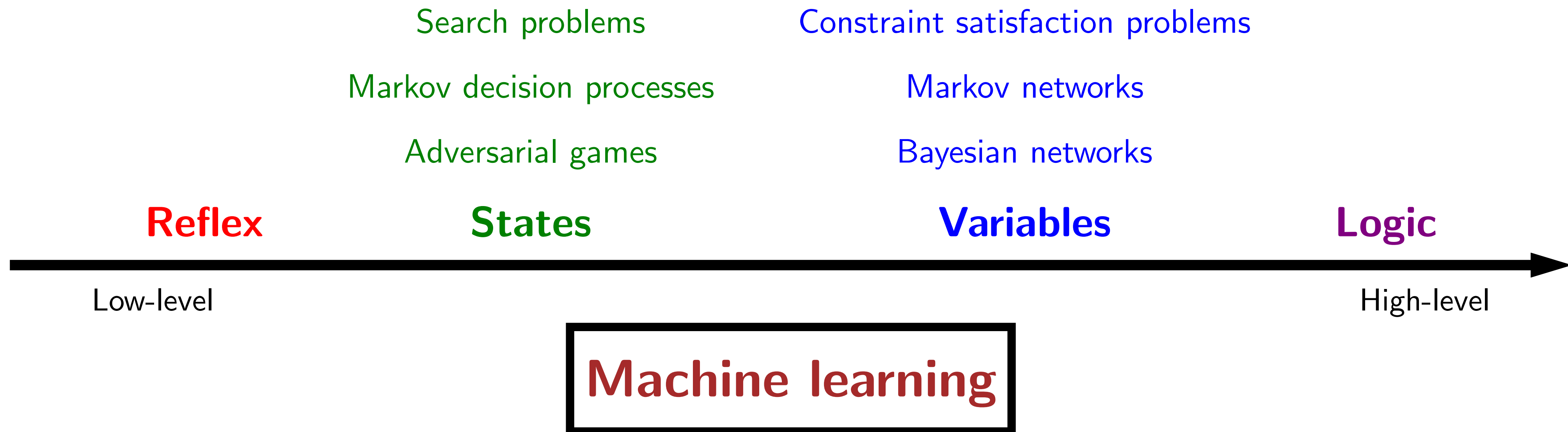
Machine learning overview

Linear regression

Linear classification

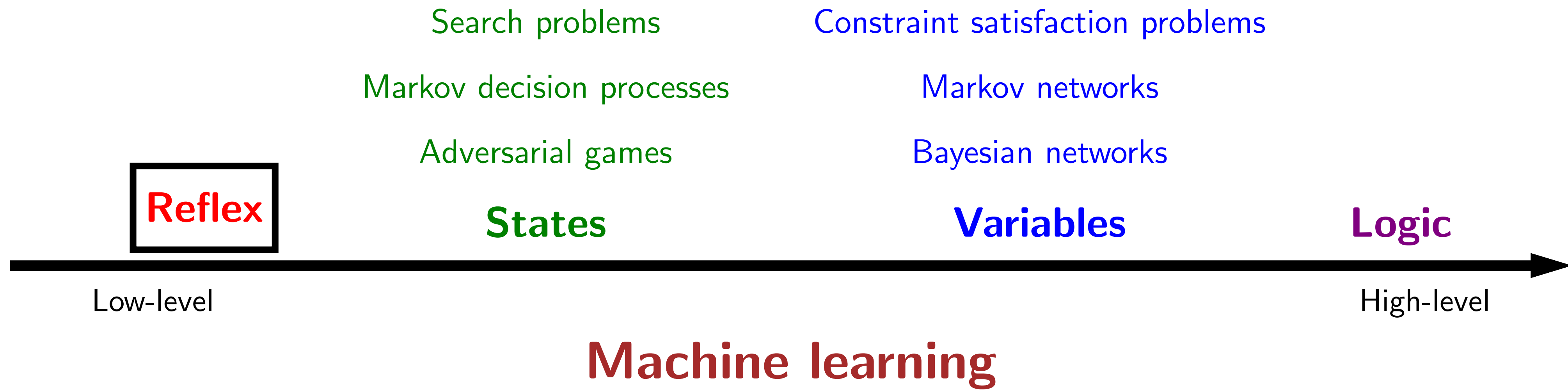
- First, we will outline the topics we plan to cover under machine learning.

Course plan



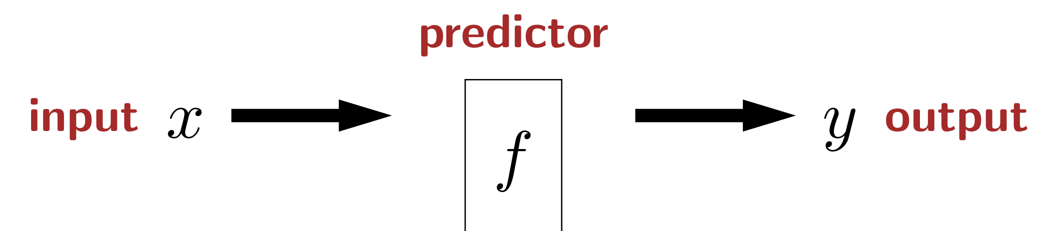
- Recall that machine learning is the process of turning data into a model. Then with that model, you can perform inference on it to make predictions.

Course plan



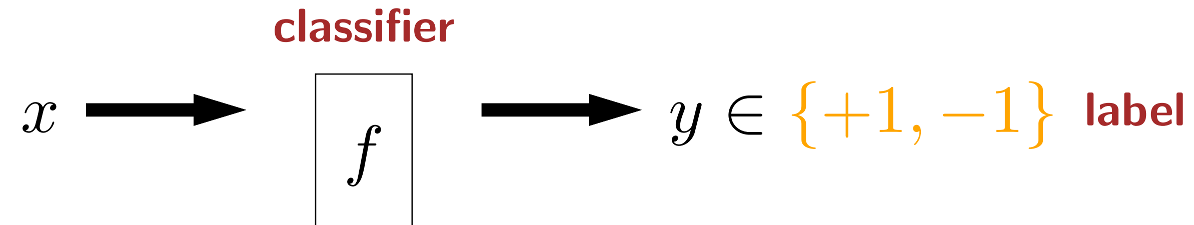
- While machine learning can be applied to any type of model, we will focus our attention on reflex-based models, which include models such as linear classifiers and neural networks.
- In reflex-based models, inference (prediction) involves a fixed set of fast, feedforward operations.

Reflex-based models



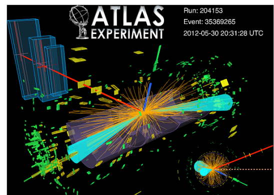
- Abstractly, a **reflex-based model** (which we will call a **predictor** f) takes some **input** x and produces some **output** y.
- (In statistics, y is known as the response, and when x is a real vector, it is known as covariates or sometimes predictors, which is an unfortunate naming clash.)
- The input can usually be arbitrary (an image or sentence), but the form of the output y is generally restricted, and what it is determines the type of **prediction task**.

Binary classification



Fraud detection: credit card transaction $\rightarrow$ fraud or no fraud

Toxic comments: online comment $\rightarrow$ toxic or not toxic

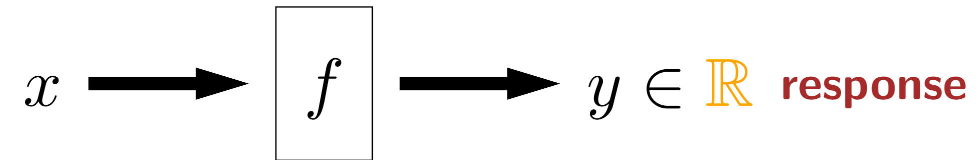


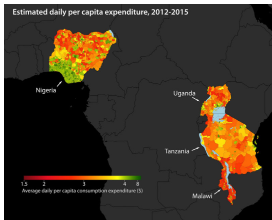
Higgs boson: measurements of event $\rightarrow$ decay event or background

Extension: multiclass classification: $y \in \{1, \ldots, K\}$

- One common prediction task is binary classification, where the output y, typically expressed as positive ($+1$) or negative (-1).
- In the context of classification tasks, f is called a **classifier** and y is called a **label** (sometimes class, category, or tag).
- Here are some practical applications.
- One application is fraud detection: given information about a credit card transaction, predict whether it is a fraudulent transaction or not, so that the transaction can be blocked.
- Another application is moderating online discussion forums: given an online comment, predict whether it is toxic (and therefore should get flagged or taken down) or not.
- A final application comes from physics: After the discovery of the Higgs boson, scientists were interested in how it decays. The Large Hadron Collider at CERN smashes protons against each other and then detects the ensuing events. The goal is to predict whether each event is a Higgs boson decaying (into two tau particles) or just background noise.
- Each of these applications has an associated Kaggle dataset. You can click on the pictures to find out more details.
- As an aside, **multiclass classification** is a generalization of binary classification where the output y could be one of K possible values. For example, in digit classification, $K = 10$.

Regression





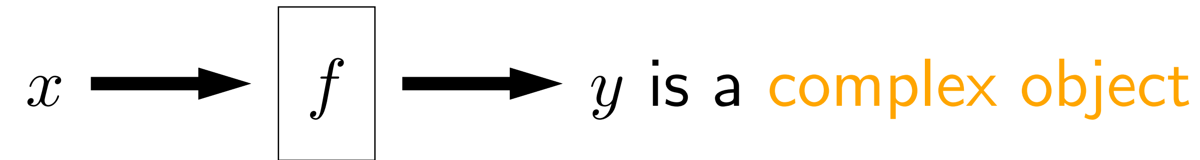
Poverty mapping: satellite image $\rightarrow$ asset wealth index

Housing: information about house $\rightarrow$ price

Arrival times: destination, weather, time $\rightarrow$ time of arrival

- The second major type of prediction task we'll cover is regression. Here, the output y is a real number (often called the **response** or target).
- One application is poverty mapping: given a satellite image, predict the average asset wealth index of the homes in that area. This is used to measure poverty across the world and determine which areas are in greatest need of aid.
- Another application: given information about a house (e.g., location, number of bedrooms), predict its price.
- A third application is to predict the arrival time of some service, which could be package deliveries, flights, or rideshares.
- The key distinction between classification and regression is that classification has **discrete** outputs (e.g., "yes" or "no" for binary classification), whereas regression has **continuous** outputs.

Structured prediction



Machine translation: English sentence $\rightarrow$ Japanese sentence

Dialogue: conversational history $\rightarrow$ next utterance

Image captioning: image $\rightarrow$ sentence describing image

Image segmentation: image $\rightarrow$ segmentation

- The final type of prediction task we will consider is structured prediction, which is a bit of a catch all.
- In **structured prediction**, the output y is a complex object, which could be a sentence or an image. So the space of possible outputs is huge.
- One application is machine translation: given an input sentence in one language, predict its translation into another language.
- Dialogue can be cast as structured prediction: given the past conversational history between a user and an agent (in the case of virtual assistants), predict the next utterance (what the agent should say).
- In image captioning, say for visual assistive technologies: given an image, predict a sentence describing what is in that image.
- In image segmentation, which is needed to localize objects for autonomous driving: given an image of a scene, predict the segmentation of that image into regions corresponding to objects in the world.
- Generating an image or a sentence can seem daunting, but there's a secret here. A structured prediction task can often be broken up into a sequence of multiclass classification tasks. For example, to predict an entire sentence, predict one word at a time, going left to right. This is a very powerful reduction!
- Aside: one challenge with this approach is that the errors might cascade: if you start making errors, then you might go off the rails and start making even more errors.

Roadmap

Tasks

Linear regression

Linear classification

K-means

Optimization Algorithms

Gradient descent

Stochastic gradient descent

Backpropagation

Models

Non-linear features

Feature templates

Neural networks

Considerations

Generalization

Best practices

- Here are the rest of the modules under the machine learning unit.
- We will start by talking about regression and binary classification, the two most fundamental tasks in machine learning. Specifically, we study the simplest setting: **linear regression** and **linear classification**, where we have linear models trained by gradient descent.
- Next, we will introduce **stochastic gradient descent**, and show that it can be much faster than vanilla gradient descent.
- Then we will push the limits of linear models by showing how you can define **non-linear features**, which effectively gives us non-linear predictors using the machinery of linear models! **Feature templates** provide us with a framework for organizing the set of features.
- Then we introduce **neural networks**, which also provide non-linear predictors, but allow these non-linearities to be learned automatically from data. We follow up immediately with **backpropagation**, an algorithm that allows us to automatically compute gradients needed for training without having to take gradients manually.
- So far we have focused on supervised learning. We take a brief detour and discuss **K-means**, which is a simple unsupervised learning algorithm for clustering data points.
- We end on a more reflective note: **Generalization** is about answering the question: when does a model trained on set of training examples actually generalize to new test inputs? This is where model complexity comes up. Finally, we discuss **best practices** for doing machine learning in practice.

Roadmap

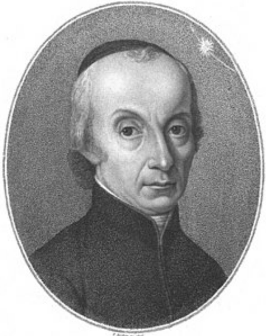
Machine learning overview

Linear regression

Linear classification

- Next, we will cover the basics of linear regression.

The discovery of Ceres



1801: astronomer Piazzi discovered Ceres, made 19 observations of location before it was obscured by the sun

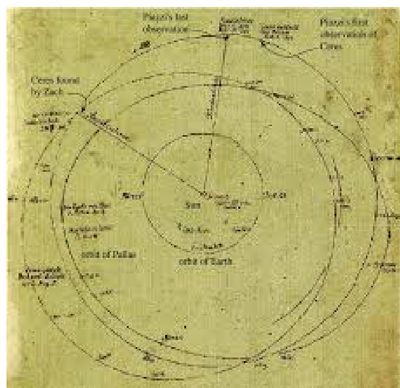
Time	Right ascension	Declination
Jan 01, 20:43:17.8	50.91	15.24
Jan 02, 20:39:04.6	50.84	15.30
...	...	...
Feb 11, 18:11:58.2	53.51	18.43

When and where will Ceres be observed again?

- Our story of linear regression starts on January 1, 1801, when an Italian astronomer Giuseppe Piazzi noticed something in the night sky while looking for stars, which he named Ceres. Was it a comet or a planet? He wasn't sure.
- He observed Ceres over 42 days and wrote down 19 data points, where each one consisted of a timestamp along with the right ascension and declination, which identifies the location in the sky.
- Then Ceres moved too close to the sun and was obscured by its glare. Now the big question was when and where will Ceres come out again?
- It was now a race for the top astronomers at the time to answer this question.

Gauss's triumph

September 1801: Gauss took Piazzi's data and created a model of Ceres's orbit, makes prediction

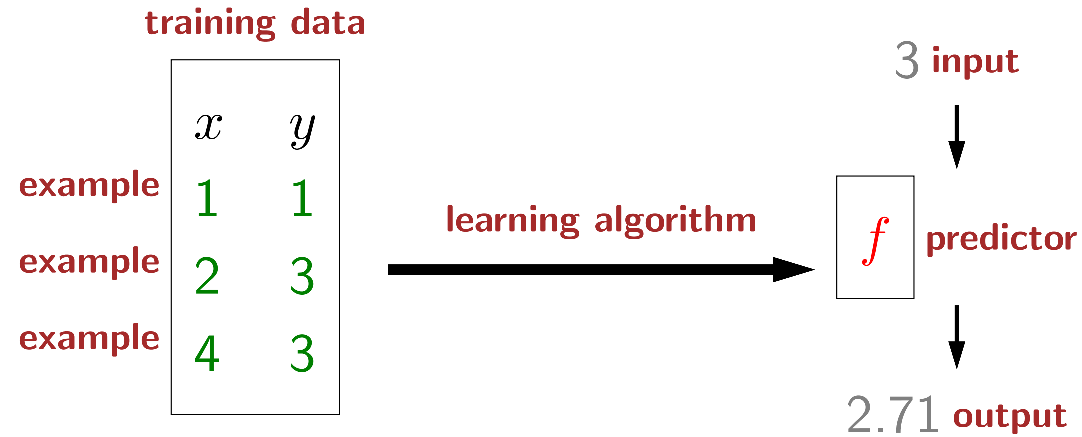


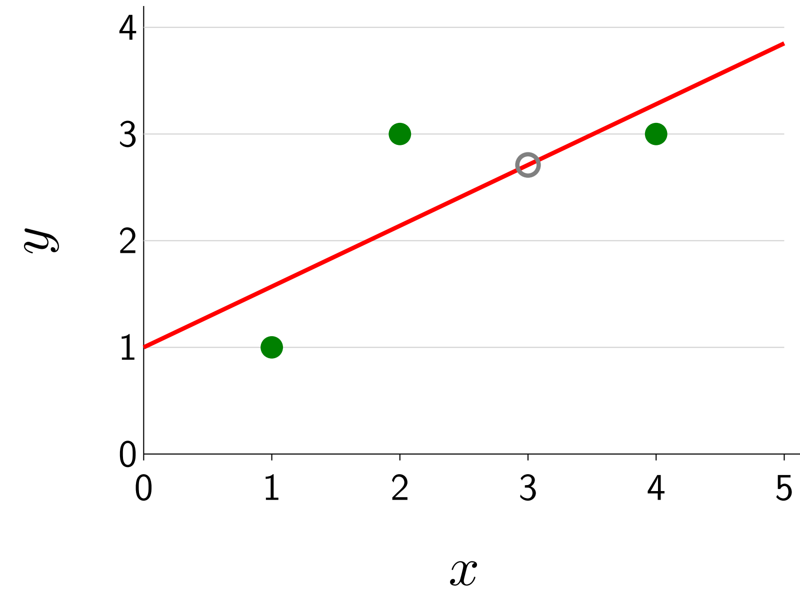
December 7, 1801: Ceres located within 1/2 degree of Gauss's prediction, much more accurate than other astronomers

Method: least squares linear regression

- Carl Friedrich Gauss, the famous German mathematician, took the data and developed a model of Ceres's orbit and used it to make a prediction.
- Clearly without a computer, Gauss did all his calculations by hand, taking over 100 hours.
- This prediction was actually quite different than the predictions made by other astronomers, but in December, Ceres was located again, and Gauss's prediction was by far the most accurate.
- Gauss was very secretive about his methods, and a French mathematician Legendre actually published the same method in 1805, though Gauss had developed the method as early as 1795.
- The method here is least squares linear regression, which is a simple but powerful method used widely today, and it captures many of the key aspects of more advanced machine learning techniques.

Linear regression framework





Design decisions:

Which predictors are possible? **hypothesis class**

How good is a predictor? **loss function**

How do we compute the best predictor? **optimization algorithm**

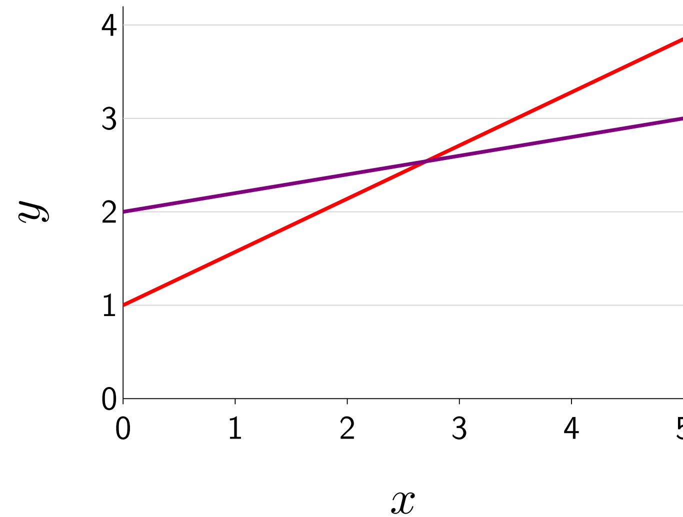
- Let us now present the linear regression framework.
- Suppose we are given **training data**, which consists of a set of examples. Each **example** (also known as data point, instance, case) consists of an input x and an output y. We can visualize the training set by plotting y against x.
- A learning algorithm takes the training data and produces a model f, which we will call a **predictor** in the context of regression. In this example, f is the red line.
- This predictor allows us to make predictions on new inputs. For example, if you feed 3 in, you get $f(3)$, corresponding to the gray circle.
- There are three design decisions to make to fully specify the learning algorithm:
- First, which predictors f is the learning algorithm allowed to produce? Only lines or curves as well? In other words, what is the **hypothesis class**?
- Second, how does the learning algorithm judge which predictor is good? In other words, what is the **loss function**?
- Finally, how does the learning algorithm actually find the best predictor? In other words, what is the **optimization algorithm**?

Hypothesis class: which predictors?

$f(x) = 1 + 0.57x$

$f(x) = 2 + 0.2x$

$f(x) = w_1 + w_2 x$



Vector notation:

weight vector $\mathbf{w} = [w_1, w_2]$

feature extractor $\phi(x) = [1, x]$ **feature vector**

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$ **score**

$f_{\mathbf{w}}(3) = [1, 0.57] \cdot [1, 3] = 2.71$

Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^2\}$$

- Let's consider the first design decision: what is the hypothesis class? One possible predictor is the red line, where the intercept is 1 and the slope is 0.57, Another predictor is the purple line, where the intercept is 2 and the slope is 0.2.
- In general, let's consider all predictors of the form $f(x) = w_1 + w_2 x$, where the intercept w_1 and and the slope w_2 can be arbitrary real numbers.
- Now let us generalize this further using vector notation. Let's pack the intercept and slope into a single vector, which we will call the **weight vector** (more generally called the parameters of the model).
- Similarly, we will define a **feature extractor** (also called a feature map) ϕ, which takes x and converts it into the **feature vector** $[1, x]$.
- Now we can succinctly write the predictor $f_{\mathbf{w}}$ to be the dot product between the weight vector and the feature vector, which we call the **score**.
- To see this predictor in action, let us feed $x = 3$ as the input and take the dot product.
- Finally, define the hypothesis class $\mathcal{F}$ to be simply the set of all possible predictors $f_{\mathbf{w}}$ as we range over all possible weight vectors $\mathbf{w}$. This is the possible functions that we want our learning algorithm to consider.

Loss function: how good is a predictor?

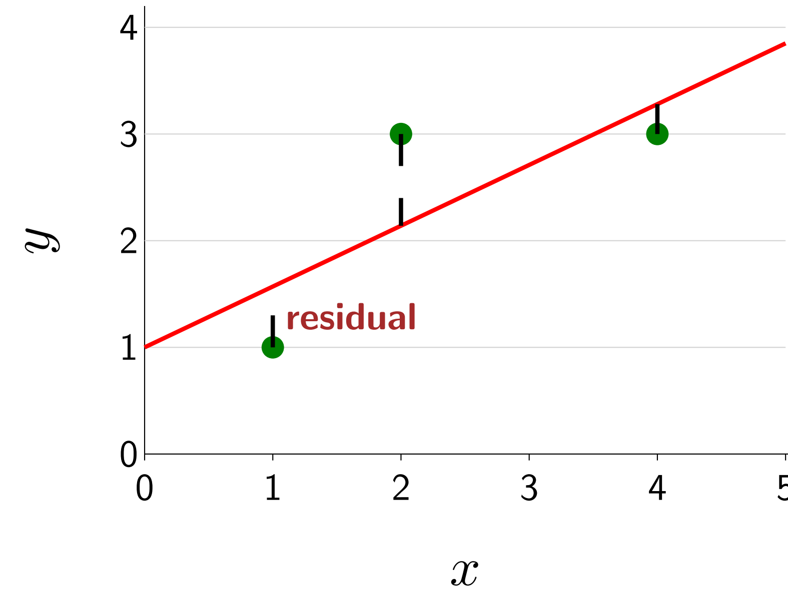
$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$

$\mathbf{w} = [1, 0.57]$

$\phi(x) = [1, x]$

training data $\mathcal{D}_{\text{train}}$

x	y
1	1
2	3
4	3



$$\text{Loss}(x, y, \mathbf{w}) = (f_{\mathbf{w}}(x) - y)^2$$ **squared loss**

$$\text{Loss}(1, 1, [1, 0.57]) = ([1, 0.57] \cdot [1, 1] - 1)^2 = 0.32$$

$$\text{Loss}(2, 3, [1, 0.57]) = ([1, 0.57] \cdot [1, 2] - 3)^2 = 0.74$$

$$\text{Loss}(4, 3, [1, 0.57]) = ([1, 0.57] \cdot [1, 4] - 3)^2 = 0.08$$

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

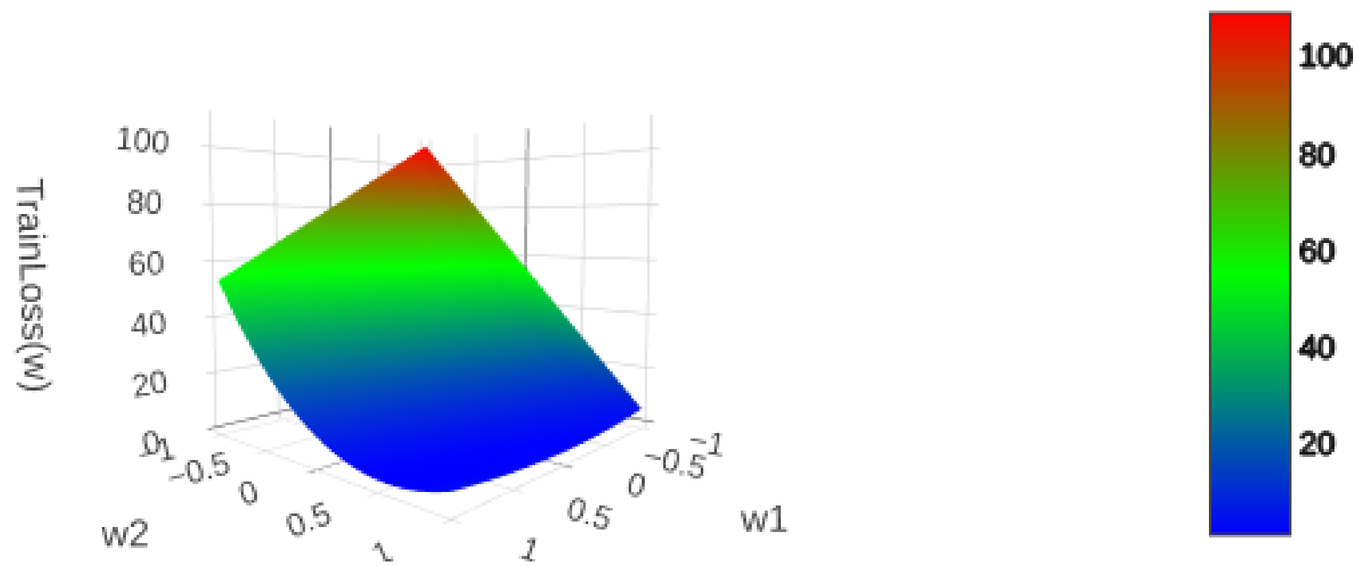
$$\text{TrainLoss}([1, 0.57]) = 0.38$$

- The next design decision is how to judge each of the many possible predictors.
- Going back to our running example, let's consider the red predictor defined by the weight vector [1, 0.57], and the three training examples.
- Intuitively, a predictor is good if it can fit the training data. For each training example, let us look at the difference between the predicted output $f_{\mathbf{w}}(x)$ and the actual output y, known as the **residual**.
- Now define the **loss function** on given example with respect to $\mathbf{w}$ to be the residual squared (giving rise to the term least squares). This measures how badly the function f screwed up on that example.
- Aside: You might wonder why we are taking the square of the residual as opposed to taking an absolute value of the residual (known as the absolute deviation): the answer for now is both mathematical and computational convenience, though if your data potentially has outliers, it is beneficial to use the absolute deviation.
- For each example, we have a per-example loss computed by plugging in the example and the weight vector. Now, define the **training loss** (also known as the training error or empirical risk) to be simply the average of the per-example losses of the training examples.
- The training loss of this red weight vector is 0.38.
- If you were to plug in the purple weight vector or any other weight vector, you would get some other training loss.

Loss function: visualization

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} (f_{\mathbf{w}}(x) - y)^2$$

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$



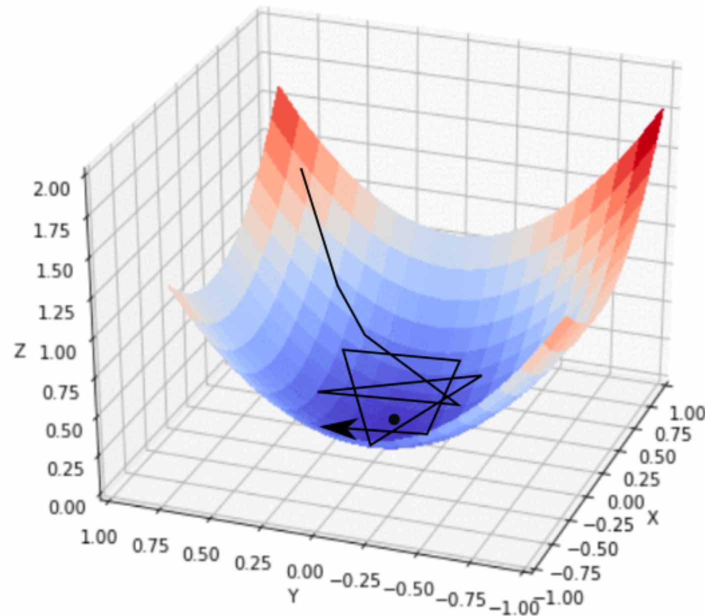
- We can visualize the training loss in this case because the weight vector $\mathbf{w}$ is only two-dimensional. In this plot, for each w_1 and w_2, we have the training loss. Red is higher, blue is lower.
- It is now clear that the best predictor is simply the one with the lowest training loss, which is somewhere down in the blue region. Formally, we wish to solve the optimization problem.

Optimization algorithm: how to compute best?

Goal: $\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

Definition: gradient

The gradient $\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$ is the direction that increases the training loss the most.



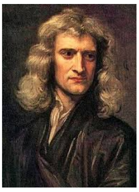
Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$: **epochs**

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

- Now the third design decision: how do we compute the best predictor, i.e., the solution to the optimization problem?
- To answer this question, we can actually forget that we're doing linear regression or machine learning. We simply have an objective function $\text{TrainLoss}(\mathbf{w})$ that we wish to minimize.
- We will adopt the "follow your nose" strategy, i.e., **iterative optimization**. We start with some $\mathbf{w}$ and keep on tweaking it to make the objective function go down.
- To do this, we will rely on the gradient of the function, which tells us the direction to move in that will decrease the objective function the most.
- Formally, this iterative optimization procedure is called **gradient descent**. We first initialize $\mathbf{w}$ to some value (say, all zeros).
- Then perform the following update T times, where T is the number of **epochs**: Take the current weight vector $\mathbf{w}$ and subtract a positive constant η times the gradient. The **step size** η specifies how aggressively we want to pursue a direction.
- The step size η and the number of epochs T are two **hyperparameters** of the optimization algorithm, which we will discuss later.



Computing the gradient

Objective function:

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} (\mathbf{w}\cdot\phi(x) - y)^2$$

Gradient (use chain rule):

$$\nabla_{\mathbf{w}}\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} 2(\underbrace{\mathbf{w}\cdot\phi(x) - y}_{\text{prediction}-\text{target}})\phi(x)$$

- To apply gradient descent, we need to compute the gradient of our objective function $\text{TrainLoss}(\mathbf{w})$.
- You could throw it into TensorFlow or PyTorch, but it is pedagogically useful to do the calculus, which can be done by hand here.
- The main thing here is to remember that we're taking the gradient with respect to $\mathbf{w}$, so everything else is a constant.
- The gradient of a sum is the sum of the gradient, the gradient of an expression squared is twice that expression times the gradient of that expression, and the gradient of the dot product $\mathbf{w} \cdot \phi(x)$ is simply $\phi(x)$.
- Note that the gradient has a nice interpretation here. For the squared loss, it is the residual (prediction - target) times the feature vector $\phi(x)$.
- Aside: no matter what the loss function is, the gradient is always something times $\phi(x)$ because $\mathbf{w}$ only affects the loss through $\mathbf{w} \cdot \phi(x)$.

Gradient descent example

training data $\mathcal{D}_{\text{train}}$

x	y
1	1
2	3
4	3

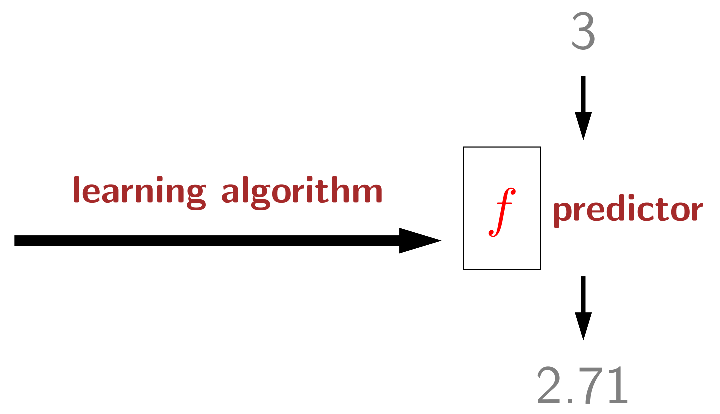
$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} 2(\mathbf{w}\cdot\phi(x) - y)\phi(x)$$

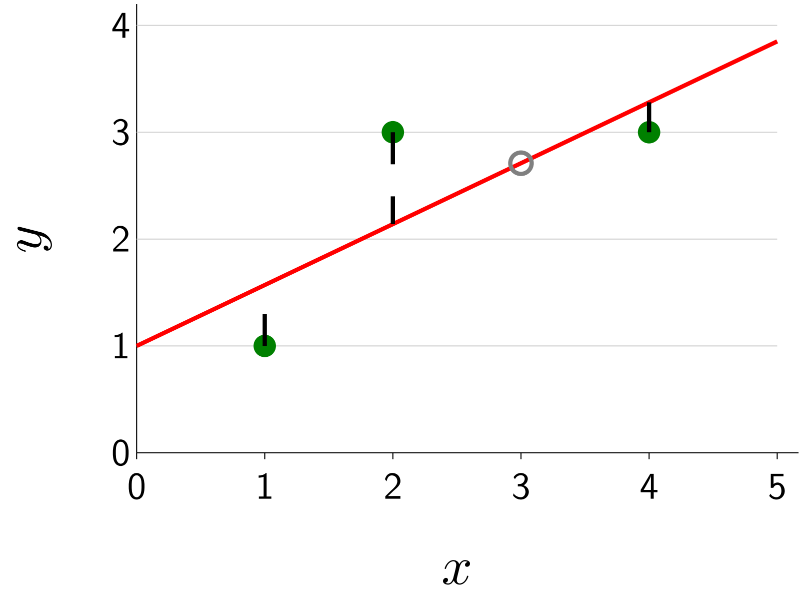
Gradient update: $\mathbf{w} \leftarrow \mathbf{w} - 0.1\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

t	$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$	$\mathbf{w}$
		$[0, 0]$
1	$\underbrace{\frac{1}{3}(2([0,0]\cdot[1,1]-1)[1,1] + 2([0,0]\cdot[1,2]-3)[1,2] + 2([0,0]\cdot[1,4]-3)[1,4])}_{=[-4.67,-12.67]}$	$[0.47, 1.27]$
2	$\underbrace{\frac{1}{3}(2([0.47,1.27]\cdot[1,1]-1)[1,1] + 2([0.47,1.27]\cdot[1,2]-3)[1,2] + 2([0.47,1.27]\cdot[1,4]-3)[1,4])}_{=[2.18,7.24]}$	$[0.25, 0.54]$
...	...	...
200	$\underbrace{\frac{1}{3}(2([1,0.57]\cdot[1,1]-1)[1,1] + 2([1,0.57]\cdot[1,2]-3)[1,2] + 2([1,0.57]\cdot[1,4]-3)[1,4])}_{=[0,0]}$	$[1, 0.57]$

- Let's step through an example of running gradient descent.
- Suppose we have the same dataset as before, the expression for the gradient that we just computed, and the gradient update rule, where we take the step size $\eta = 0.1$.
- We start with the weight vector $\mathbf{w} = [0, 0]$. Let's then plug this into the expression for the gradient, which is an average over the three training examples, and each term is the residual (prediction - target) times the feature vector.
- That vector is multiplied by the step size (0.1 here) and subtracted out of the weight vector.
- We then take the new weight vector and plug it in again to the expression for the gradient. This produces another gradient, which is used to update the weight vector.
- If you run this procedure for long enough, you eventually get the final weight vector. Note that the gradient at the end is zero, which indicates that the algorithm has converged and running it longer will not change anything.

Summary





Which predictors are possible?
Hypothesis class

Linear functions
$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)\}, \phi(x) = [1, x]$

How good is a predictor?
Loss function

Squared loss
$\text{Loss}(x, y, \mathbf{w}) = (f_{\mathbf{w}}(x) - y)^2$

How to compute best predictor?
Optimization algorithm

Gradient descent
$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \textbf{TrainLoss}(\mathbf{w})$

- In this module, we have gone through the basics of linear regression. A learning algorithm takes training data and produces a predictor f, which can then be used to make predictions on new inputs.
- Then we addressed the three design decisions:
- First, what is the hypothesis class (the space of allowed predictors)? We focused on linear functions, but we will later see how this can be generalized to other feature extractors to yield non-linear functions, and beyond that, neural networks.
- Second, how do we assess how good a given predictor is with respect to the training data? For this we used the squared loss, which gives us least squares regression. We will see later how other losses allow us to handle problems such as classification.
- Third, how do we compute the best predictor? We described the simplest procedure, gradient descent. Later, we will see how stochastic gradient descent can be much more computational efficient.
- And that concludes this module.

Roadmap

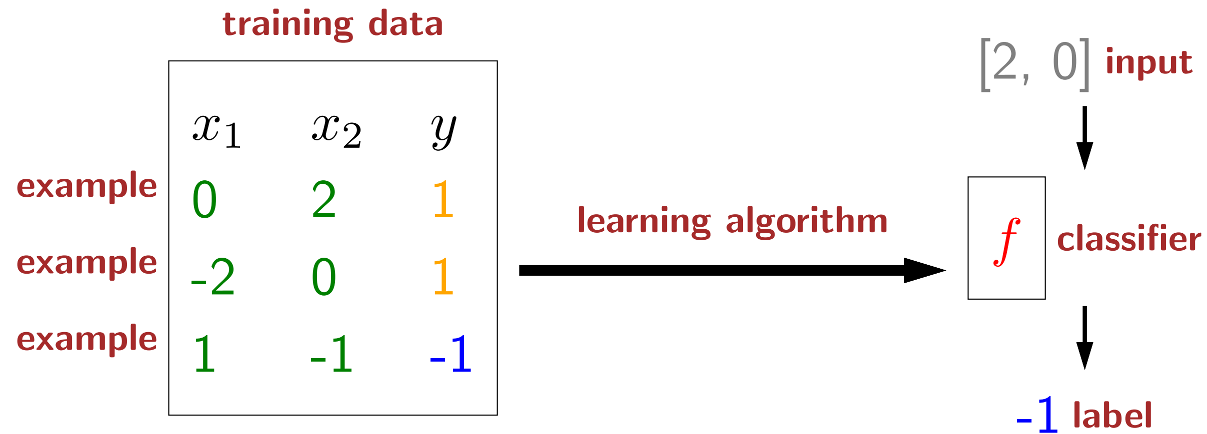
Machine learning overview

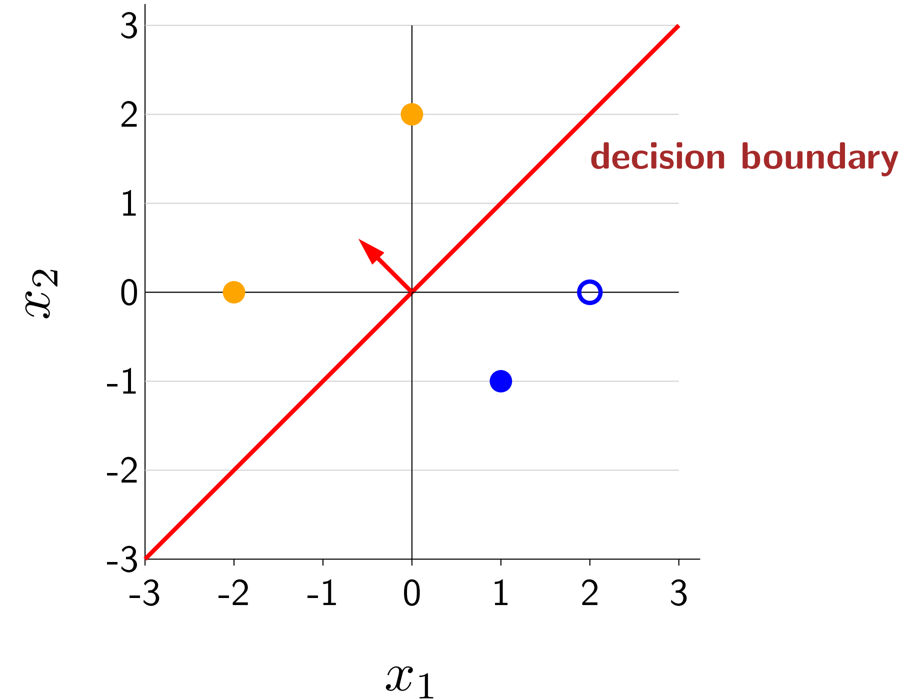
Linear regression

Linear classification

- We now present linear (binary) classification, working through a simple example just like we did for linear regression.

Linear classification framework





Design decisions:

Which classifiers are possible? **hypothesis class**

How good is a classifier? **loss function**

How do we compute the best classifier? **optimization algorithm**

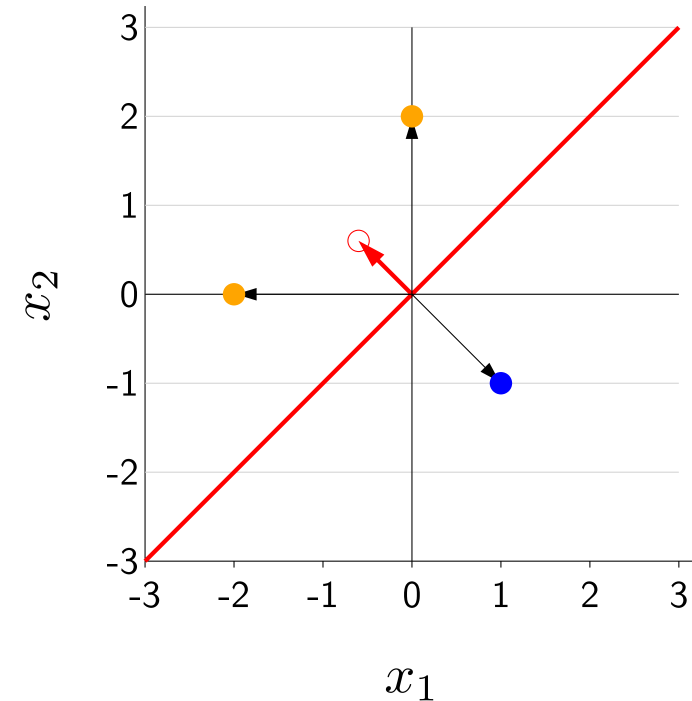
- Here is the linear classification framework.
- As usual, we are given **training data**, which consists of a set of examples. Each **example** consists of an input $x = (x_1, x_2)$ and an output y. We are considering two-dimensional inputs now to make the example a bit more interesting. The examples can be plotted, with the color denoting the label (orange for $+1$ and blue for -1).
- We still want a learning algorithm that takes the training data and produces a model f, which we will call a **classifier** in the context of classification.
- The classifier takes a new input and produces an output. We can visualize a classifier on the 2D plot by its **decision boundary**, which divides the input space into two regions: the region of input points that the classifier would output $+1$ and the region that the classifier would output -1. By convention, the arrow points to the positive region.
- Again, there are the same three design decisions to fully specify the learning algorithm:
- First, which classifiers f is the learning algorithm allowed to produce? Must the decision boundary be straight or can it curve? In other words, what is the **hypothesis class**?
- Second, how does the learning algorithm judge which classifier is good? In other words, what is the **loss function**?
- Finally, how does the learning algorithm actually find the best classifier? In other words, what is the **optimization algorithm**?

An example linear classifier

$$f(x) = \text{sign}(\underbrace{[-0.6, 0.6]}_{\mathbf{w}} \cdot \underbrace{[x_1, x_2]}_{\phi(x)})$$

$$\text{sign}(z) = \begin{cases} +1 & \text{if } z > 0 \\ -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \end{cases}$$

x_1	x_2	$f(x)$
0	2	1
-2	0	1
1	-1	-1



$$f([0, 2]) = \text{sign}([-0.6, 0.6] \cdot [0, 2]) = \text{sign}(1.2) = 1$$

$$f([-2, 0]) = \text{sign}([-0.6, 0.6] \cdot [-2, 0]) = \text{sign}(1.2) = 1$$

$$f([1, -1]) = \text{sign}([-0.6, 0.6] \cdot [1, -1]) = \text{sign}(-1.2) = -1$$

Decision boundary: x such that $\mathbf{w} \cdot \phi(x) = 0$

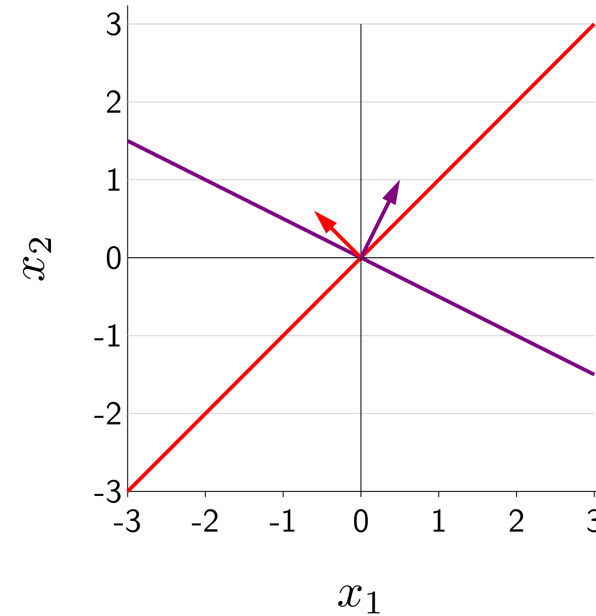
- Before we talk about the hypothesis class over all classifiers, we will start by exploring the properties of a specific linear classifier.
- First take the dot product between a fixed weight vector $\mathbf{w}$ and the identity feature vector $\phi(x)$. Then take the sign of the dot product.
- The **sign** of a number z is $+1$ if $z > 0$ and -1 if $z < 0$ and 0 if $z = 0$.
- Let's now visualize what f does. First, we can plot $\mathbf{w}$ either as a point or as a vector from the origin to that point; the latter will be most useful for our purposes.
- Let's feed some inputs into f.
- Take the first point, which can be visualized on the plot as a vector. Recall from linear algebra that the dot product between two vectors is the cosine of the angle between them. In particular, the dot product is positive iff the angle is acute and negative iff the angle is obtuse. The first point forms an acute angle and therefore is classified as $+1$, which can also be verified mathematically.
- The second point also forms an acute angle and therefore is classified as $+1$.
- The third point forms an obtuse angle and is classified as -1.
- Now you can hopefully see the pattern now. All points in this region (consisting of points forming an acute angle) will be classified $+1$, and all points in this region (consisting of points forming an obtuse angle) will be classified -1.
- Points x which form a right angle ($\mathbf{w} \cdot \phi(x) = 0$) form the **decision boundary**.
- Indeed, you can see pictorially that the decision boundary is perpendicular to the weight vector.

Hypothesis class: which classifiers?

$\phi(x) = [x_1, x_2]$

$f(x) = \text{sign}([-0.6, 0.6] \cdot \phi(x))$

$f(x) = \text{sign}([0.5, 1] \cdot \phi(x))$



General binary classifier:

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$$

Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^2\}$$

- We've looked at one particular red classifier.
- We can also consider an alternative purple classifier, which has a different decision boundary.
- In general for binary classification, given a particular weight vector $\mathbf{w}$ we define $f_{\mathbf{w}}$ to be the sign of the dot product.

Loss function: how good is a classifier?

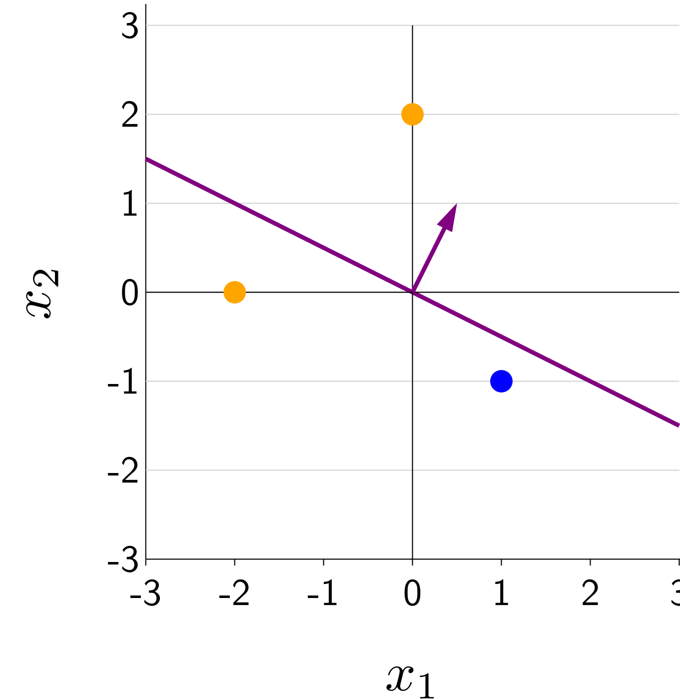
$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$

$\mathbf{w} = [0.5, 1]$

$\phi(x) = [x_1, x_2]$

training data $\mathcal{D}_{\text{train}}$

x_1	x_2	y
0	2	1
-2	0	1
1	-1	-1



$\text{Loss}_{\text{0-1}}(x, y, \mathbf{w}) = \mathbf{1}[f_{\mathbf{w}}(x) \neq y]$ **zero-one loss**

$\text{Loss}([0, 2], 1, [0.5, 1]) = \mathbf{1}[\text{sign}([0.5, 1] \cdot [0, 2]) \neq 1] = 0$

$\text{Loss}([-2, 0], 1, [0.5, 1]) = \mathbf{1}[\text{sign}([0.5, 1] \cdot [-2, 0]) \neq 1] = 1$

$\text{Loss}([1, -1], -1, [0.5, 1]) = \mathbf{1}[\text{sign}([0.5, 1] \cdot [1, -1]) \neq -1] = 0$

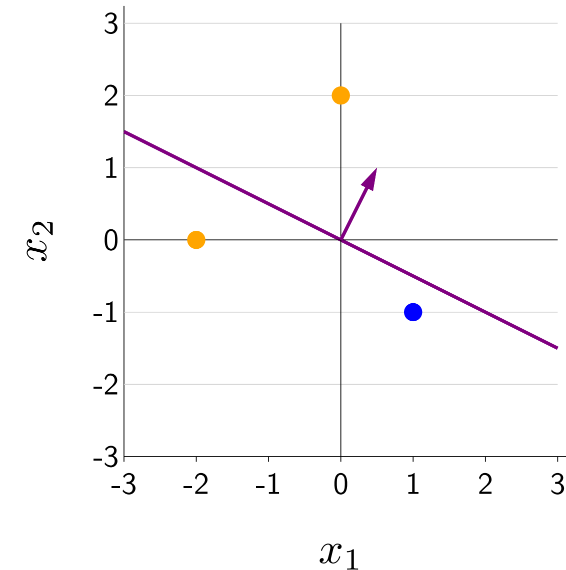
$\text{TrainLoss}([0.5, 1]) = 0.33$

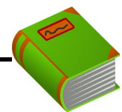
- Now we proceed to the second design decision: the loss function, which measures how good a classifier is.
- Let us take the purple classifier, which can be visualized on the graph, as well as the training examples.
- Now we want to define a loss function that captures how the model predictions deviate from the data. We will define the **zero-one loss** to check if the model prediction $f_{\mathbf{w}}(x)$ disagrees with the target label y. If so, then the indicator function $\mathbf{1}[f_{\mathbf{w}}(x) \neq y]$ will return 1; otherwise, it will return 0.
- Let's see this classifier in action.
- For the first training example, the prediction is 1, the target label is 1, so the loss is 0.
- For the second training example, the prediction is -1, the target label is 1, so the loss is 1.
- For the third training example, the prediction is -1, the target label is -1, so the loss is 0.
- The total loss is simply the average over all the training examples, which yields $1/3$.

Score and margin

Predicted label: $f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$

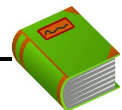
Target label: y





Definition: score

The score on an example (x, y) is $\mathbf{w} \cdot \phi(x)$, how **confident** we are in predicting $+1$.

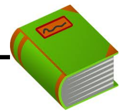


Definition: margin

The margin on an example (x, y) is $(\mathbf{w} \cdot \phi(x))y$, how **correct** we are.

- Before we move to the third design decision (optimization algorithm), let us spend some time understanding two concepts so that we can rewrite the zero-one loss.
- Recall the definition of the predicted label and the target label.
- The first concept, which we already have encountered is the **score**. In regression, this is the predicted output, but in classification, this is the number before taking the sign.
- Intuitively, the score measures how confident the classifier is in predicting $+1$.
- Points farther away from the decision boundary have larger scores.
- The second concept is **margin**, which measures how correct the prediction is. The larger the margin the more correct, and non-positive margins correspond to classification errors. If $y = 1$, then the score needs to be very positive for a large margin. If $y = -1$, then the score needs to be very negative for a large margin.
- Note that if we look at the actual prediction $f_{\mathbf{w}}(x)$, we can only ascertain whether the prediction was right or not.
- By looking at the score and the margin, we can get a more nuanced view into the behavior of the classifier.

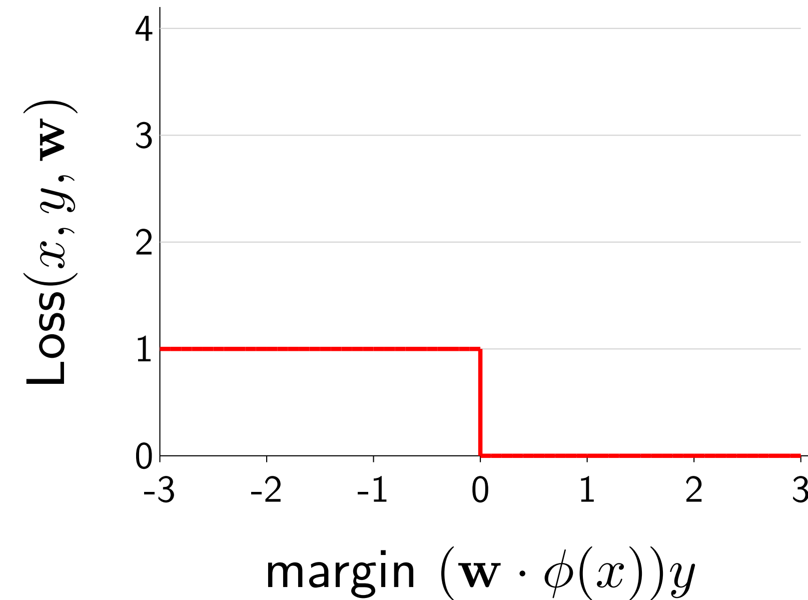
Zero-one loss rewritten



Definition: zero-one loss

$$\text{Loss}_{\text{0-1}}(x, y, \mathbf{w}) = \mathbf{1}[f_{\mathbf{w}}(x) \neq y]$$

$$= \mathbf{1}[\underbrace{(\mathbf{w} \cdot \phi(x))y}_{\text{margin}} \leq 0]$$



- Now let us rewrite the zero-one loss in terms of the margin.
- We can also plot the loss against the margin.
- Again, a positive margin yields zero loss while a non-positive margin yields loss 1.

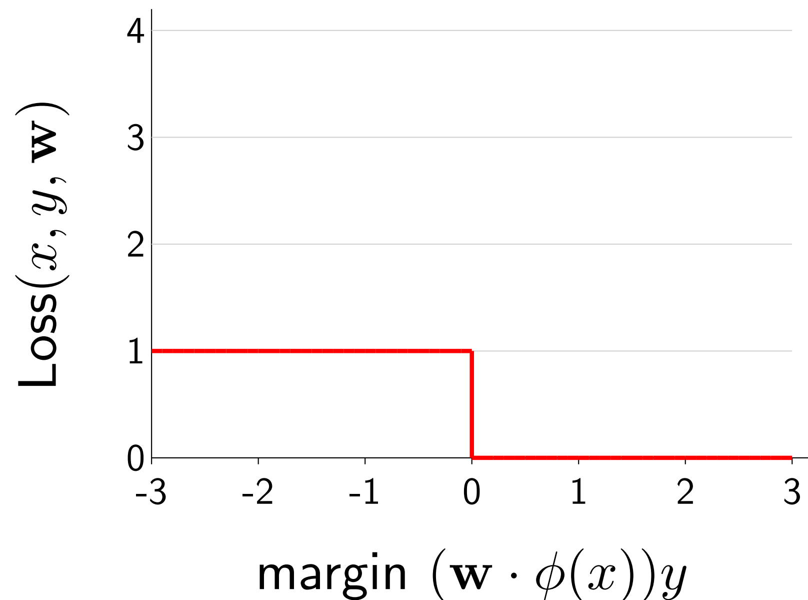
Optimization algorithm: how to compute best?

Goal: $\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

To run gradient descent, compute the gradient:

$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \nabla \text{Loss}_{\text{0-1}}(x, y, \mathbf{w})$$

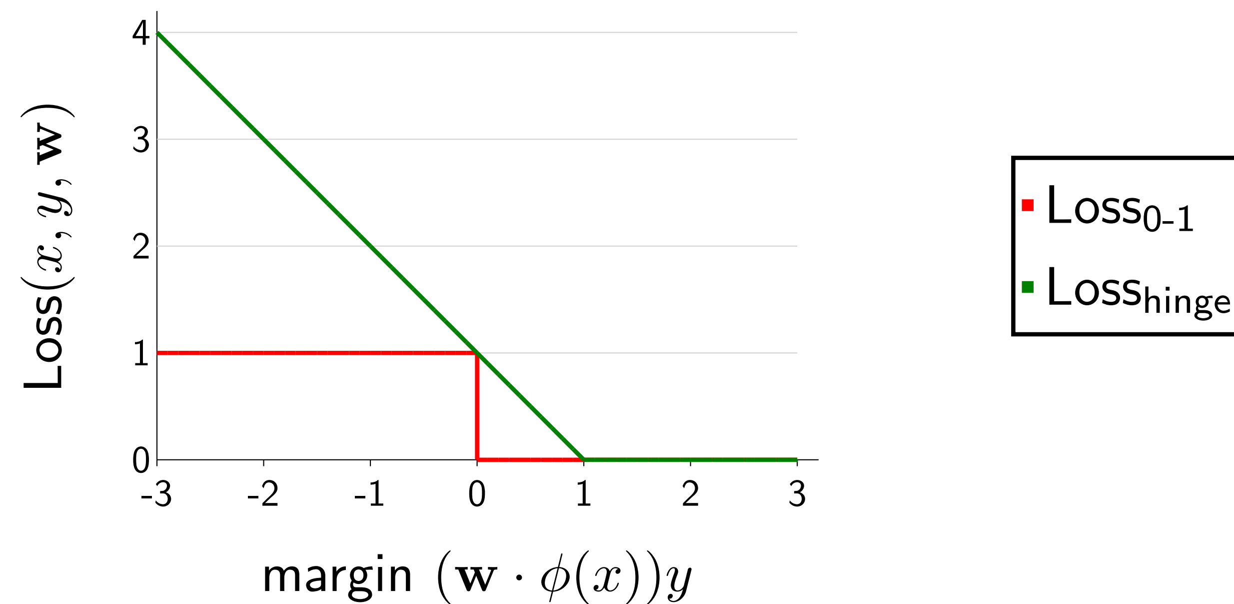
$$\nabla_{\mathbf{w}} \text{Loss}_{\text{0-1}}(x, y, \mathbf{w}) = \nabla \mathbf{1}[(\mathbf{w} \cdot \phi(x))y \leq 0]$$



Gradient is zero almost everywhere!

- Now we consider the third design decision, the optimization algorithm for minimizing the training loss.
- Let's just go with gradient descent. Recall that to run gradient descent, we need to first compute the gradient.
- The gradient of the training loss is just the average over the per-example losses. And then we need to take the gradient of indicator function...
- But this is where we run into problems: recall that the zero-one loss is flat almost everywhere (except at margin $= 0$), so the gradient is zero almost everywhere.
- If you try running gradient descent on a function with zero gradient, you will be stuck.
- One's first reaction to why the zero-one loss is hard to optimize is that it is not differentiable (everywhere). However, that is not really the real reason. The real reason is because it has zero gradients.

Hinge loss

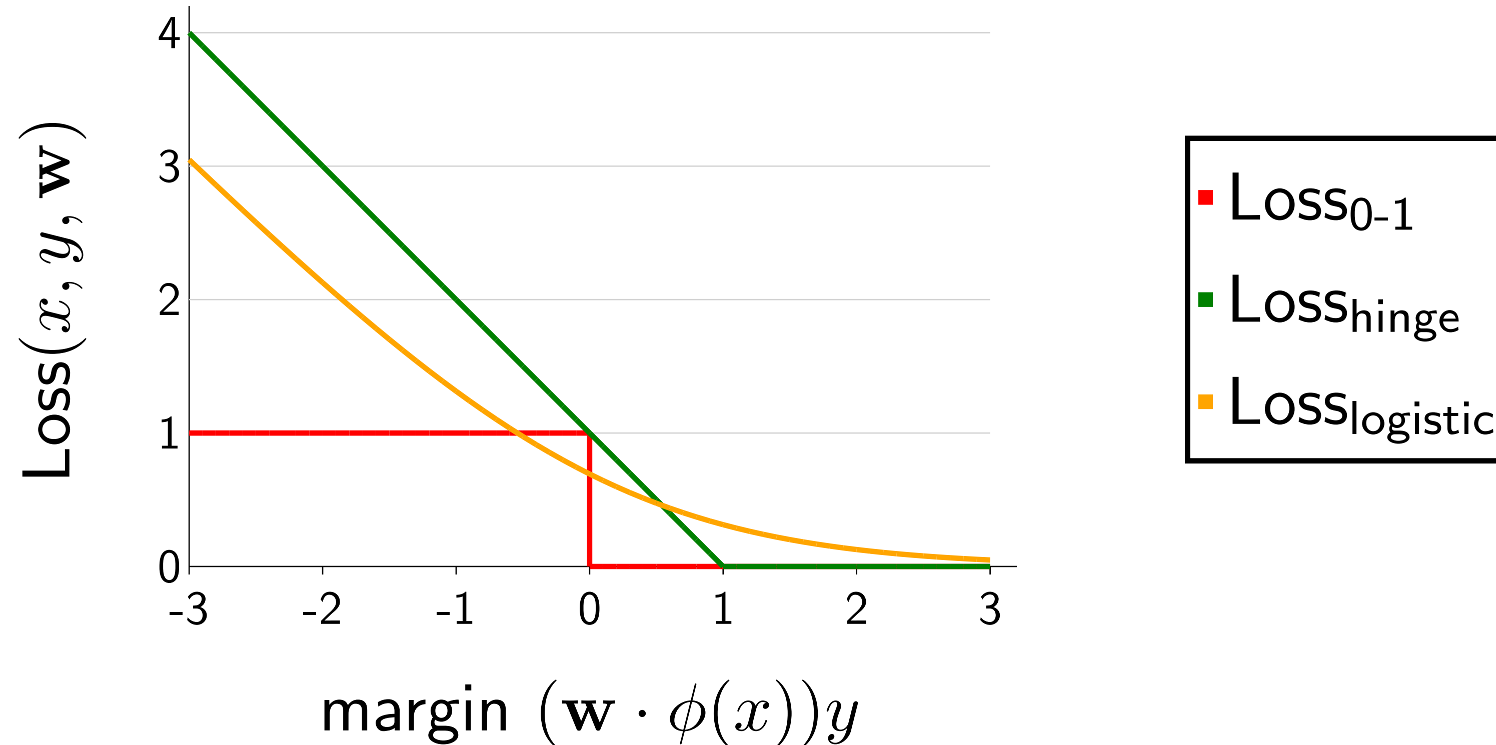


$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

- To fix this problem, we have to choose another loss function.
- A popular loss function is the **hinge loss**, which is the maximum over a descending line and the zero function. It is best explained visually.
- If the margin is at least 1, then the hinge loss is zero.
- If the margin is less than 1, then the hinge loss rises linearly.
- The 1 is there to provide some buffer: we ask the classifier to predict not only correctly, but by a (positive) margin of safety.
- Aside: Technically, the 1 can be any positive number. If we have regularization, it is equivalent to setting the regularization strength.
- Also note that the hinge loss is an upper bound on the zero-one loss, so driving down the hinge loss will generally drive down the zero-one loss. In particular, if the hinge loss is zero, the zero-one loss must also be zero.

Digression: logistic regression

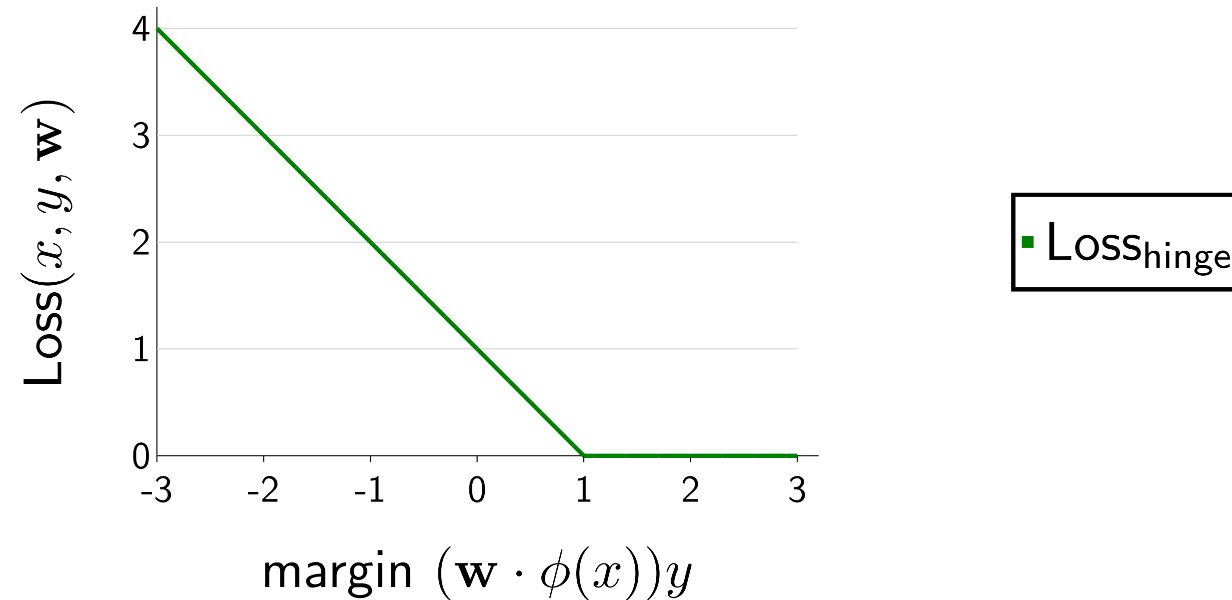
$$\text{Loss}_{\text{logistic}}(x, y, \mathbf{w}) = \log(1 + e^{-(\mathbf{w}\cdot\phi(x))y})$$



Intuition: Try to increase margin even when it already exceeds 1

- Another popular loss function used in machine learning is the **logistic loss**.
- The main property of the logistic loss is no matter how correct your prediction is, you will have non-zero loss, and so there is still an incentive (although a diminishing one) to push the loss down by increasing the margin.

Gradient of the hinge loss



$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

$$\nabla \text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \begin{cases} -\phi(x)y & \text{if } \{1 - (\mathbf{w} \cdot \phi(x))y\} > \{0\} \\ 0 & \text{otherwise} \end{cases}$$

- You should try to "see" the solution before you write things down formally. Pictorially, it should be evident: when the margin is less than 1, then the gradient is the gradient of $1 - (\mathbf{w} \cdot \phi(x))y$, which is equal to $-\phi(x)y$. If the margin is larger than 1, then the gradient is the gradient of 0, which is 0. Combining the two cases: $\nabla_{\mathbf{w}} \text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \begin{cases} -\phi(x)y & \text{if } (\mathbf{w} \cdot \phi(x))y < 1 \\ 0 & \text{if } (\mathbf{w} \cdot \phi(x))y > 1. \end{cases}$
- What about when the margin is exactly 1? Technically, the gradient doesn't exist because the hinge loss is not differentiable there. But in practice, you can take either $-\phi(x)y$ or 0.
- Technical note (can be skipped): given $f(\mathbf{w})$, the gradient $\nabla f(\mathbf{w})$ is only defined at points $\mathbf{w}$ where f is differentiable. However, subdifferentials $\partial f(\mathbf{w})$ are defined at every point (for convex functions). The subdifferential is a set of vectors called subgradients $z \in f(\mathbf{w})$ which define linear underapproximations to f, namely $f(\mathbf{w}) + z \cdot (\mathbf{w}' - \mathbf{w}) \leq f(\mathbf{w}')$ for all $\mathbf{w}'$.

Hinge loss on training data

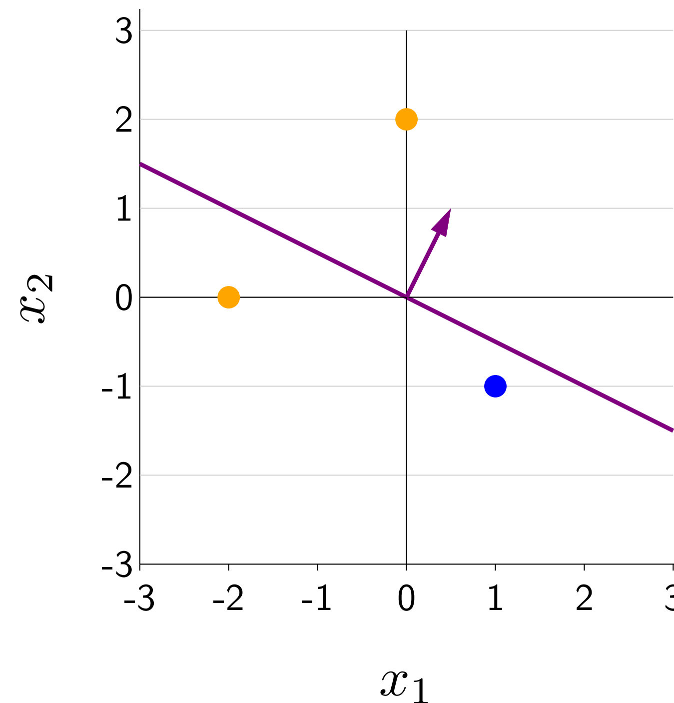
$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

$\mathbf{w} = [0.5, 1]$

$\phi(x) = [x_1, x_2]$

training data $\mathcal{D}_{\text{train}}$

x_1	x_2	y
0	2	1
-2	0	1
1	-1	-1



$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

$\text{Loss}([0, 2], 1, [0.5, 1]) = \max\{1 - [0.5, 1] \cdot [0, 2](1), 0\} = 0$

$\text{Loss}([-2, 0], 1, [0.5, 1]) = \max\{1 - [0.5, 1] \cdot [-2, 0](1), 0\} = 2$

$\text{Loss}([1, -1], -1, [0.5, 1]) = \max\{1 - [0.5, 1] \cdot [1, -1](-1), 0\} = 0.5$

$\text{TrainLoss}([0.5, 1]) = 0.83$

$\nabla\text{Loss}([0, 2], 1, [0.5, 1]) = [0, 0]$

$\nabla\text{Loss}([-2, 0], 1, [0.5, 1]) = [2, 0]$

$\nabla\text{Loss}([1, -1], -1, [0.5, 1]) = [1, -1]$

$\nabla\text{TrainLoss}([0.5, 1]) = [1, -0.33]$

- Now let us revisit our earlier setting with the hinge loss.
- For each example (x, y), we can compute its loss, and the final loss is the average.
- For the first example, the loss is zero, and therefore the gradient is zero.
- For the second example, the loss is non-zero which is expected since the classifier is incorrect. The gradient is non-zero.
- For the third example, note that the loss is non-zero even though the classifier is correct. This is because we have to have a margin of 1, but the margin in this case is only 0.5.

Summary so far

$$\underbrace{\mathbf{w} \cdot \phi(x)}_{\text{score}}$$

	Regression	Classification
Prediction $f_{\mathbf{w}}(x)$	score	sign(score)
Relate to target y	residual (score $- y$)	margin (score y)
Loss functions	squared absolute deviation	zero-one hinge logistic
Algorithm	gradient descent	gradient descent

- Let us end by comparing and contrasting linear classification and linear regression.
- The score is a common quantity that drives the prediction in both cases.
- In regression, the output is the raw score. In classification, the output is the sign of the score.
- To assess whether the prediction is correct, we must relate the score to the target y. In regression, we use the residual, which is the difference (lower is better). In classification, we use the margin, which is the product (higher is better).
- Given these two quantities, we can form a number of different loss functions. In regression, we studied the squared loss, but we could also consider the absolute deviation loss (taking absolute values instead of squared). In classification, we care about the zero-one loss (which corresponds to the missclassification rate), but we optimize the hinge or the logistic loss.
- Finally, gradient descent can be used in both settings.