CSC321 Neural Networks and Machine Learning

Lecture 2

January 15, 2020

Agenda

First hour:

- Homework 1 (one clarification regarding notation)
- Gradient Descent
- Vectorization

Second hour:

- Linear Classification
- Logistic Regression

Homework 1 notation

There was an update to question 4 on Jan 13th.

Q: Can we write the sum $\sum_{i=1}^{N} |x_i|$ like this:

$$\sum_{i=1}^{N} |x_i| = \begin{cases} \sum_{i=1}^{N} x_i & \text{if} x_i >= 0 \\ \sum_{i=1}^{N} -x_i & \text{otherwise} \end{cases}$$

Homework 1 notation

There was an update to question 4 on Jan 13th.

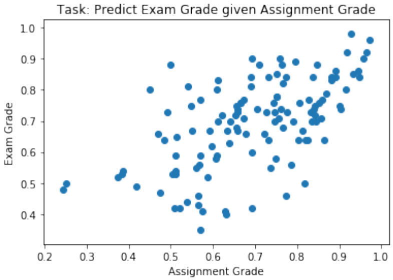
Q: Can we write the sum $\sum_{i=1}^{N} |x_i|$ like this:

$$\sum_{i=1}^{N} |x_i| = \begin{cases} \sum_{i=1}^{N} x_i & \text{if} x_i >= 0 \\ \sum_{i=1}^{N} -x_i & \text{otherwise} \end{cases}$$

No, we can't! Each of $x_1, x_2, ..., x_N$ could have *different* signs!

Regression Review

We would like to make predictions about some continuous value (e.g. exam grade) given some input (e.g. assignment grade)



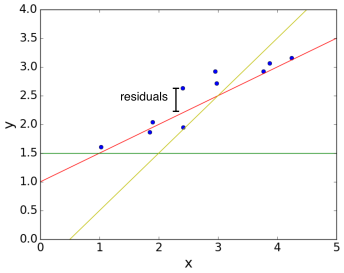
- Data: $(x^{(1)}, t^{(1)})$, $(x^{(2)}, t^{(2)})$, $\ldots$ $(x^{(N)}, t^{(N)})$
- The $x^{(i)}$ are called *inputs*
- The $t^{(i)}$ are called *targets*

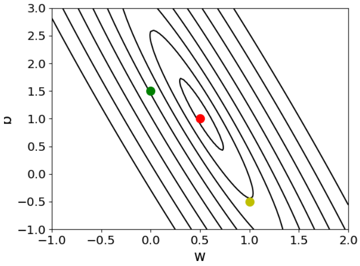
Linear Regression Review

Hypothesis	$y = wx + b$
Parameters	$w,\ b$
Cost Function	$\mathcal{E}(w, b) = \frac{1}{2N}\sum_i ((wx^{(i)} + b) - t^{(i)})^2$
Goal	Find w, b that minimize $\mathcal{E}(w, b)$

Optimization

How do we find w, b that minimize $\mathcal{E}(w, b)$?





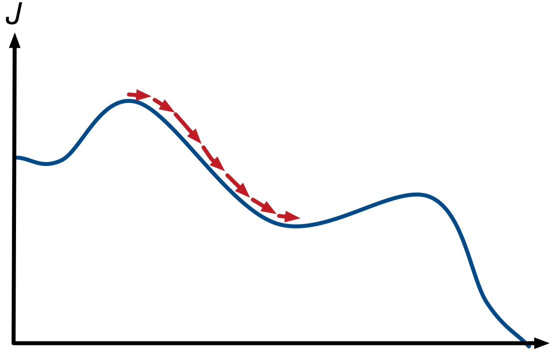
Last time:

- Grid search: slow, especially if $\mathbf{w}$ is high dimensional
- Direct solution: won't work for many models and loss functions

Today: Gradient Descent

Gradient Descent

Minimizing a scalar function f(x)



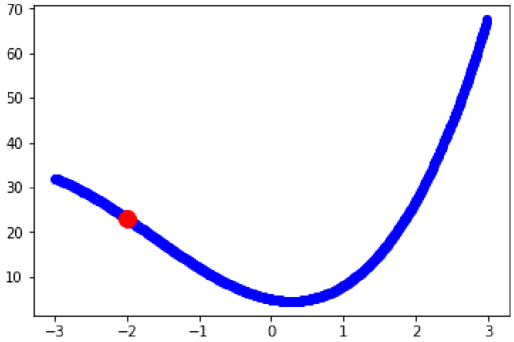
Gradient Descent is an iterative method used to find the minima of a function.

We'll start by thinking about a *scalar function* (1D)

To minimize a function f(x), we start with a random point x_0 and iterate an update rule that we will derive.

Deriving Gradient Descent Update

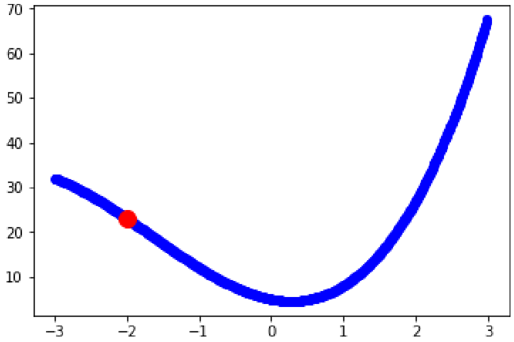
Consdier this function f(x)



Q: If we want to move the red point closer to the minima, do we move left or right?

Deriving Gradient Descent Update

Consdier this function f(x)

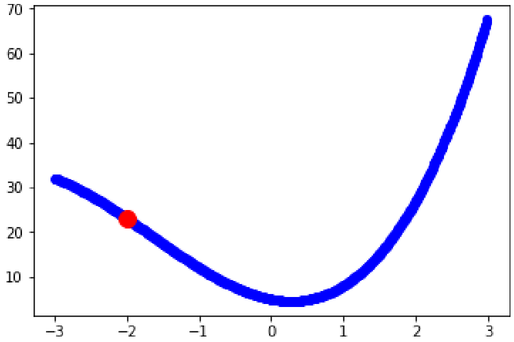


Q: If we want to move the red point closer to the minima, do we move left or right?

Q: At the red point x, is the derivative $f'(x)$ positive or negative?

Deriving Gradient Descent Update

Consdier this function f(x)

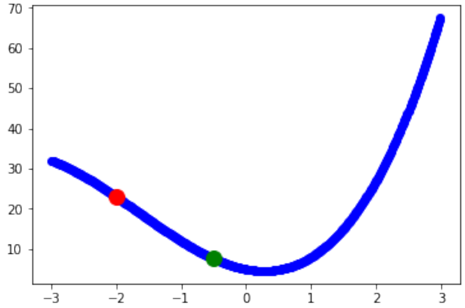


Q: If we want to move the red point closer to the minima, do we move left or right?

Q: At the red point x, is the derivative $f'(x)$ positive or negative?

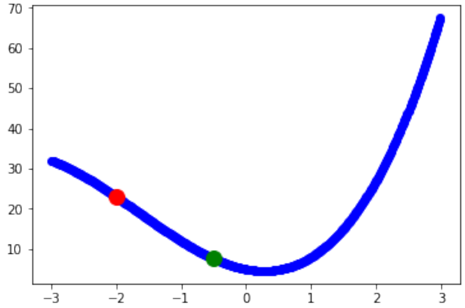
We want to move x towards the negative direction of the gradient!

How much do we move?



Q: Should we make a larger jump at the red point or green?

How much do we move?



Q: Should we make a larger jump at the red point or green?

The larger $|f'(x)|$, the more we should move. We *slow down* close to a minima.

$$x \leftarrow x - \alpha f'(x)$$

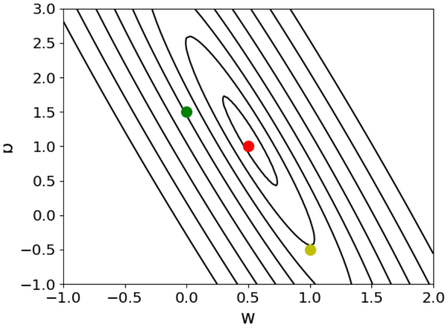
The term α is the **learning rate**

Gradient Descent for Linear Regression (2D)

The same idea holds in higher dimensions:

$$w \leftarrow w - \alpha \frac{\partial \mathcal{E}}{\partial w}$$

$$b \leftarrow b - \alpha \frac{\partial \mathcal{E}}{\partial b}$$



Gradient Descent for Linear Regression (high dimensional)

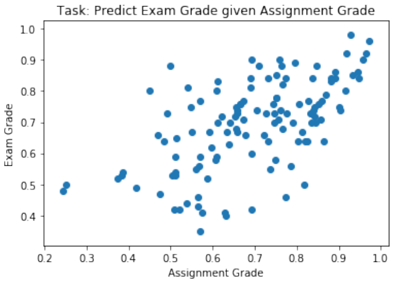
Or, in general:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{E}}{\partial \mathbf{w}}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial \mathcal{E}}{\partial w_1} \\ \dots \\ \frac{\partial \mathcal{E}}{\partial w_D} \end{bmatrix}$$

It turns out that the gradient is the direction of the **steepest descent**.

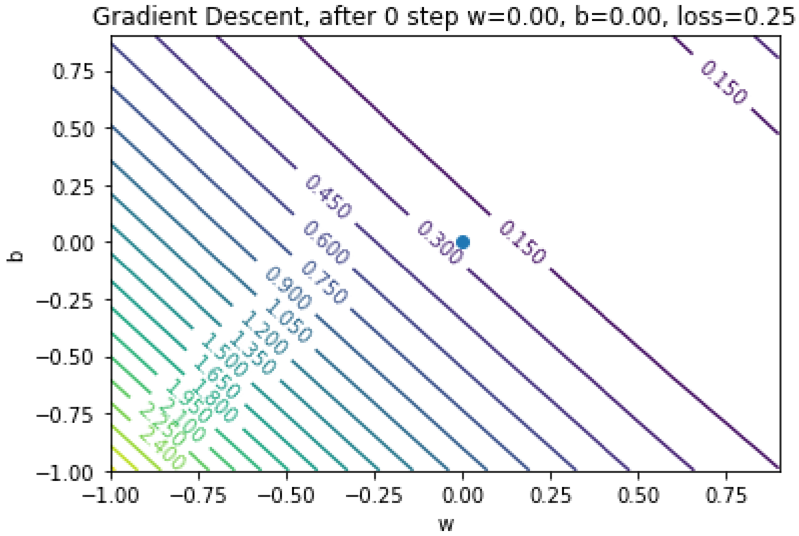
Gradient Descent for Grade Prediction



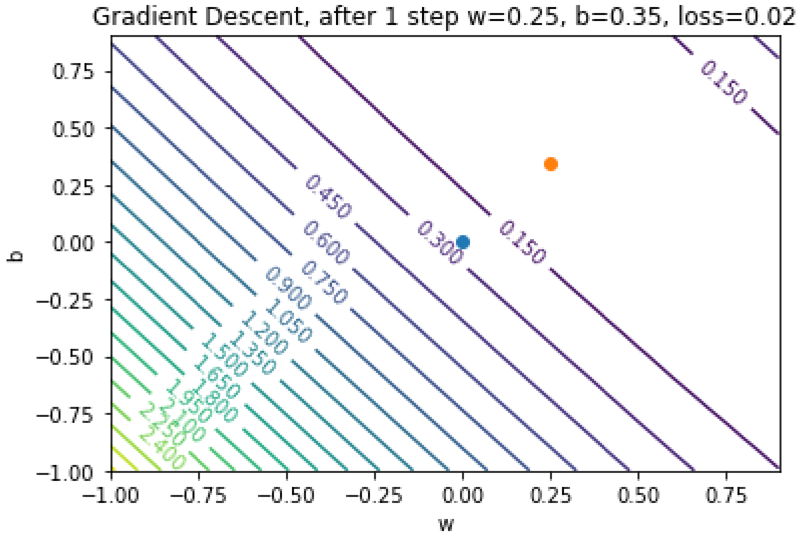
We'll initialize $w = 0$ and $b = 0$ (arbitrary choice)

We'll also choose $\alpha = 0.5$

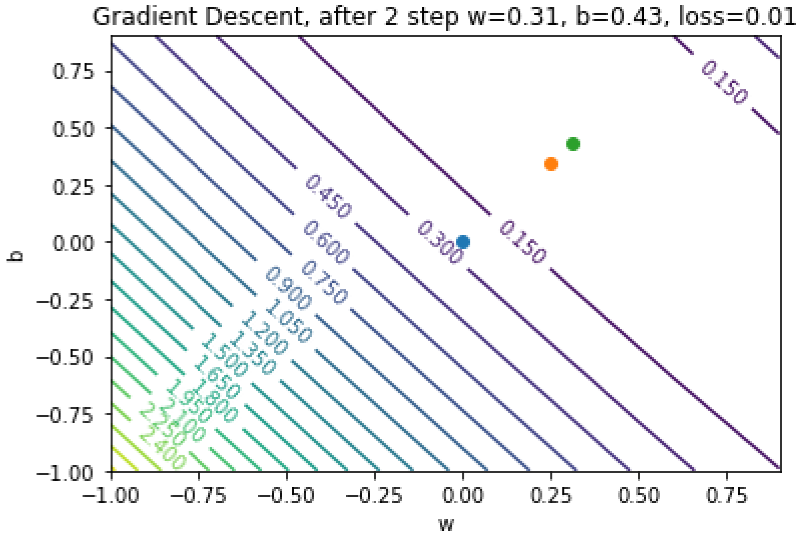
Gradient Descent: Step 0



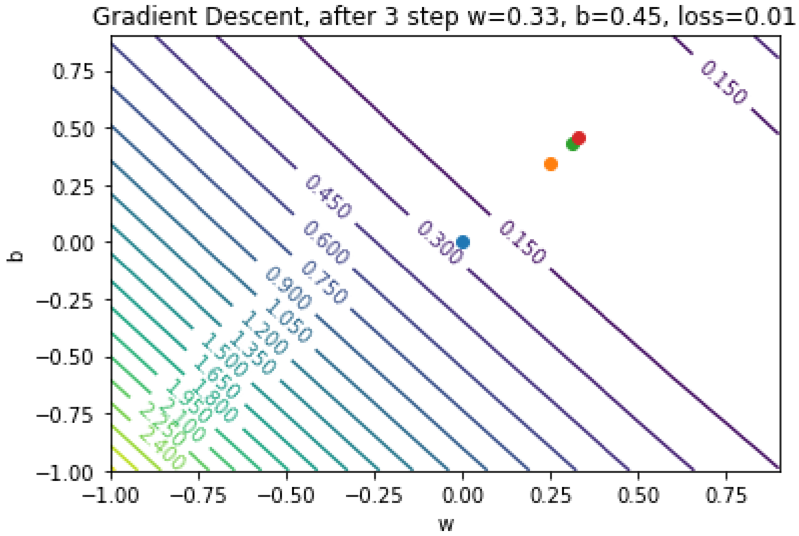
Gradient Descent: Step 1



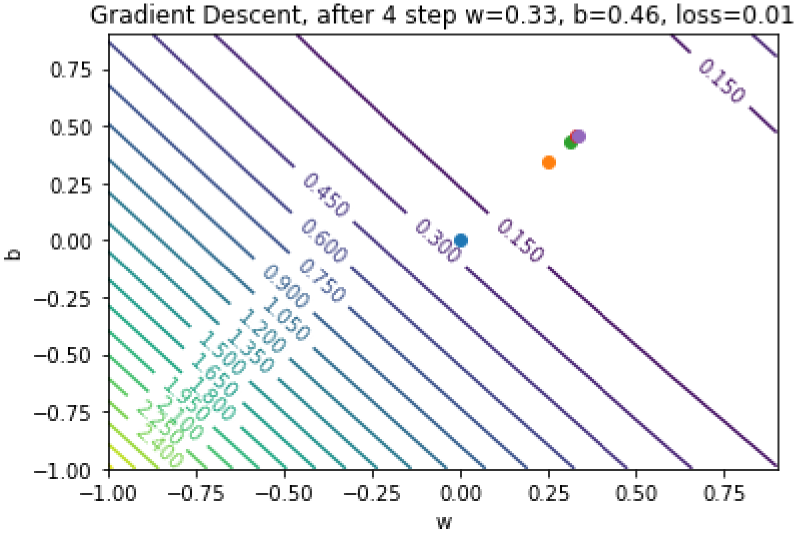
Gradient Descent: Step 2



Gradient Descent: Step 3



Gradient Descent: Step 4



Gradient Descent: when to stop?

In theory:

- Stop when w and b stop changing (convergence)

In practice:

- Stop when $\mathcal{E}$ almost stops changing (another notion of convergence)
- Stop until we're tired of waiting

Gradient Descent: how to choose the learning rate?

- If α is too small, then training will be *slow*
  - Take a long time to converge
- If α is too large, then we can have divergence!
  - Take a long time to converge

Computing the gradient

To compute the gradient $\frac{\partial \mathcal{E}}{\partial w}$

$$\frac{\partial \mathcal{E}}{\partial w} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}(y^{(i)}, t^{(i)})}{\partial w}$$

But this computation can be expensive if N is large!

Computing the gradient

To compute the gradient $\frac{\partial \mathcal{E}}{\partial w}$

$$\frac{\partial \mathcal{E}}{\partial w} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}(y^{(i)}, t^{(i)})}{\partial w}$$

But this computation can be expensive if N is large!

Solution: estimate $\frac{\partial \mathcal{E}}{\partial w}$ using a *subset* of the data

Stochastic Gradient Descent

Full batch gradient descent:

$$\frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}(y^{(i)}, t^{(i)})}{\partial w}$$

Stochastic Gradient Descent:

Estimate the above quantity by computing the average of $\frac{\partial \mathcal{L}(y^{(i)}, t^{(i)})}{\partial w}$ across a small number of i's

The set of examples that we use to estimate the gradient is called a **mini-batch**.

The number of examples in each mini-batch is called the **mini-batch size** or just the **batch size**

Stochastic Gradient Descent Algorithm

In theory, any way of sampling a mini-batch is okay.

In practice, SGD is almost always implemented like this:

```
# repeat until convergence:
    # group the data set into mini-batches of size $k$
    # for each mini-batch:
        # estimate the gradient using the mini-batch
        # update the parameters based on the estimate
```

Stochastic Gradient Descent Algorithm

In theory, any way of sampling a mini-batch is okay.

In practice, SGD is almost always implemented like this:

```
# repeat until convergence:
    # group the data set into mini-batches of size $k$
    # for each mini-batch:
        # estimate the gradient using the mini-batch
        # update the parameters based on the estimate
```

- Each pass of the inner loop is called an **iteration**.
    - One iteration = one update for each weight
- Each pass of the outer loop is called an **epoch**.
    - One epoch = one pass over the data set

Iterations, Epochs, and Batch Size

Suppose we have 1000 examples in our training set.

Q: How many iterations are in one epoch if our batch size is 10?

Iterations, Epochs, and Batch Size

Suppose we have 1000 examples in our training set.

Q: How many iterations are in one epoch if our batch size is 10?

Q: How many iterations are in one epoch if our batch size is 50?

Batch size choice

Q: What happens if the batch size is **too large**?

Q: What happens if the batch size is **too small**?

Vectorization

Linear Regression Vectorization

Use vectors rather than writing

$\mathcal{E}(\mathbf{w}, b) = \frac{1}{2N} \sum_i ((\mathbf{w}\mathbf{x}^{(i)} + b) - t^{(i)})^2$

So we have:

$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$, where

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_D^{(2)} \\ \dots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_D^{(N)} \end{bmatrix}, \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_D \end{bmatrix}, \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(N)} \end{bmatrix}, \mathbf{t} = \begin{bmatrix} t^{(1)} \\ t^{(2)} \\ \dots \\ t^{(N)} \end{bmatrix}$$

(You can also fold the bias b into the weight $\mathbf{w}$, but we won't.)

Vectorized Loss Function

After vectorization, the loss function becomes:

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2N}(\mathbf{y} - \mathbf{t})^T(\mathbf{y} - \mathbf{t})$$

or

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2N}(\mathbf{X}\mathbf{w} + b\mathbf{1} - \mathbf{t})^T(\mathbf{X}\mathbf{w} + b\mathbf{1} - \mathbf{t})$$

Vectorized Gradient Descent

$$
\begin{aligned}
b &\leftarrow b - \alpha \frac{\partial \mathcal{E}}{\partial b} \\
\mathbf{w} &\leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{E}}{\partial \mathbf{w}}
\end{aligned}
$$

Where $\frac{\partial \mathcal{E}}{\partial \mathbf{w}}$ is the vector of partial derivatives:

$$
\frac{\partial \mathcal{E}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial \mathcal{E}}{\partial w_1} \\ \dots \\ \frac{\partial \mathcal{E}}{\partial w_D} \end{bmatrix}
$$

Why vectorize?

Vectorization is *not* just for mathematical elegance. (Tutorial 2)

When using Python with numpy/PyTorch, code that performs vector computations is faster than code that loops.

Same holds for many other high level languages and software.

Classification

Classification Setup

- Data: $(x^{(1)}, t^{(1)})$, $(x^{(2)}, t^{(2)})$, ... $(x^{(N)}, t^{(N)})$
- The $x^{(i)}$ are called *inputs*
- The $t^{(i)}$ are called *targets*

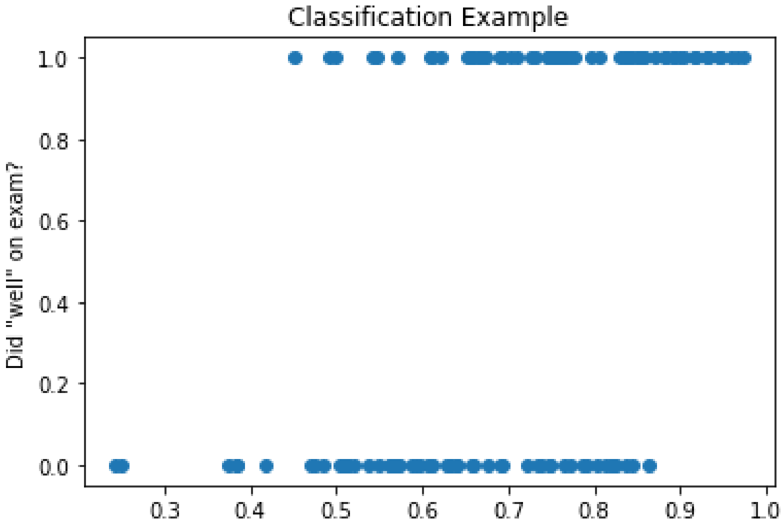
In classification, the $t^{(i)}$ are discrete.

In binary classification, we'll use the labels $t \in 0, 1$. Training examples with

- $t = 1$ is called a **positive example**
- $t = 0$ is called a **negative example** (sorry)

Classification Running Example

- $x^{(i)}$ represents a person's assignment grade
- $t^{(i)}$ represents whether that person had a "high" exam grade (arbitrary cutoff)



Q: Why not use regression?

Why can't we set up this problem as a regression problem?

Use the model:

$$y = wx + b$$

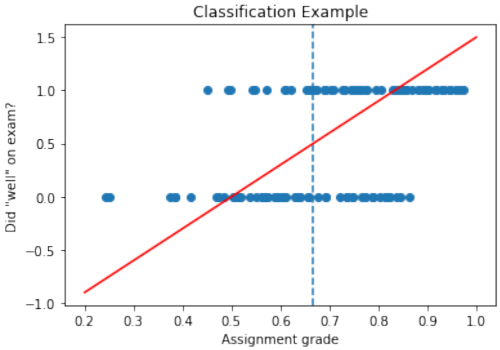
Our prediction for t would be 1 if $y >= 0.5$, and 0 otherwise.

With the loss function

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

And minimize the cost function via gradient descent?

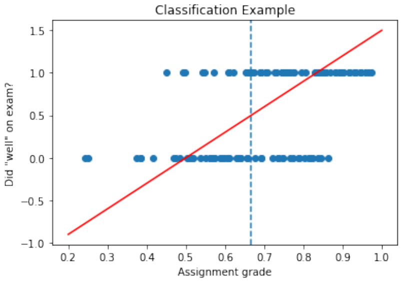
Classification as Regression: Problem



If we have $\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$, then **points that are correctly classified will still have high loss**!

(blue dotted line above = decision boundary)

The Problem (continued)



Example: a point on the top right

- Model makes the correct prediction for point on top right
- However, $(y - t)^2$ is large
- So we are penalizing our model, even though it is making the right prediction!

Q: Why not use classification error?

Why not still use the model:

$$y = \begin{cases} 1 \text{ if } \mathbf{w}^T \mathbf{x} > 0 \\ 0 \text{ otherwise} \end{cases}$$

But use this loss function instead:

$$\mathcal{L}(y, t) = \begin{cases} 0 \text{ if } y = t \\ 1 \text{ otherwise} \end{cases}$$

Gradient Descent Requires a *differentiable* Loss function

This loss function is **not differentiable**!

$$\mathcal{L}(y, t) = \begin{cases} 1 \text{ if } y = t \\ 0 \text{ otherwise} \end{cases}$$

So we cannot use gradient descent!

(The notes talk about perceptron learning rule, but we'll skip that.)

Ideal loss function

For a positive example:

- If $y = wx + b$ is large and positive, the loss should be small
- If $y = wx + b$ is close to zero, the loss should be moderate
- If $y = wx + b$ is large and *negative*, the loss should be large

Towards the Ideal Loss Function

To have the desired loss function behaviour, we need to do two things:

1. Change the model by adding a **nonlinearity** or **activation function**
2. Use the cross-entropy loss with our new model

Logistic Regression Model

Apply a **nonlinearity** or **activation function**:

$$\begin{aligned} z &= wx + b \\ y &= \sigma(z) \end{aligned}$$

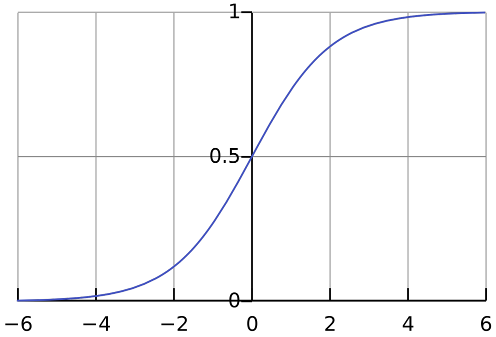
where

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

This model for solving a classification problem is called **logistic regression**

The sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

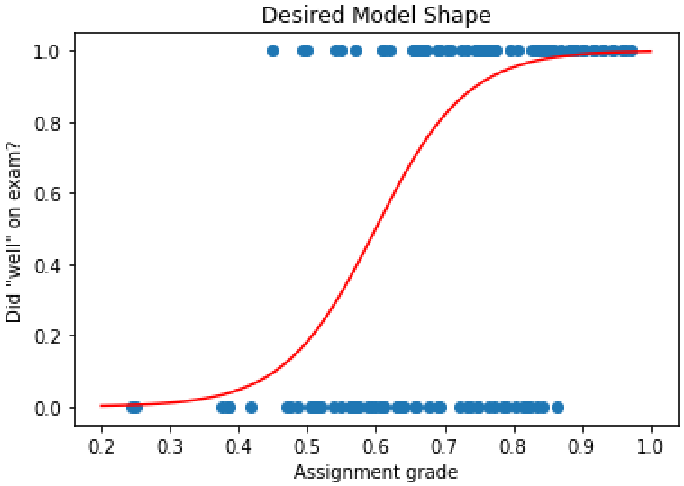


Properties:

- $\sigma(z)$ is between 0 and 1
- $\sigma(0)$ is 0.5

Logistic Regression Example

A logistic regression model will have this shape:



But how do we train this model?

Logistic Regression: Square Loss?

Suppose we define the model like this:

$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L}_{SE}(y, t) &= \frac{1}{2}(y - t)^2
\end{aligned}
$$

The gradient of $\mathcal{L}$ with respect to w is (homework):

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial \mathcal{L}}{\partial y} \frac{dy}{dz} \frac{\partial z}{\partial w} \\
&= (y - t)y(1 - y)x
\end{aligned}
$$

The problem with square loss

Suppose we have a positive example ($t = 1$) that our model classifies extremely wrongly ($z = -5$):

Then we have $y = \sigma(z) \approx 0.0067$

The problem with square loss

Suppose we have a positive example ($t = 1$) that our model classifies extremely wrongly ($z = -5$):

Then we have $y = \sigma(z) \approx 0.0067$

Ideally, the *gradient* should give us strong signals regarding how to update w to do better.

But... $\frac{\partial \mathcal{L}}{\partial w} = (y - t)y(1 - y)x$ is small!

The problem with square loss

Suppose we have a positive example ($t = 1$) that our model classifies extremely wrongly ($z = -5$):

Then we have $y = \sigma(z) \approx 0.0067$

Ideally, the *gradient* should give us strong signals regarding how to update w to do better.

But... $\frac{\partial \mathcal{L}}{\partial w} = (y - t)y(1 - y)x$ is small!

Which means that the update $w \leftarrow w - \alpha \frac{\partial \mathcal{L}}{\partial w}$ won't change w much!

Gradient Signal

The problem with using sigmoid activation with square loss is that we get **poor gradient signal**.

- The loss for a *very wrong prediction* ($y = 0.0001$) vs a wrong prediction ($y=0.01) are similar
- This is a problem, because the gradients in the region would be close to 0

We need a loss function that distinguishes between a wrong prediction and a *very* wrong prediction.

The Cross Entropy Loss

The **cross entropy loss** provides the desired behaviour:

$$\mathcal{L}(y, t) = \begin{cases} -\log(y) \text{ if } t = 1 \\ -\log(1-y) \text{ if } t = 0 \end{cases}$$

We can write the loss as:

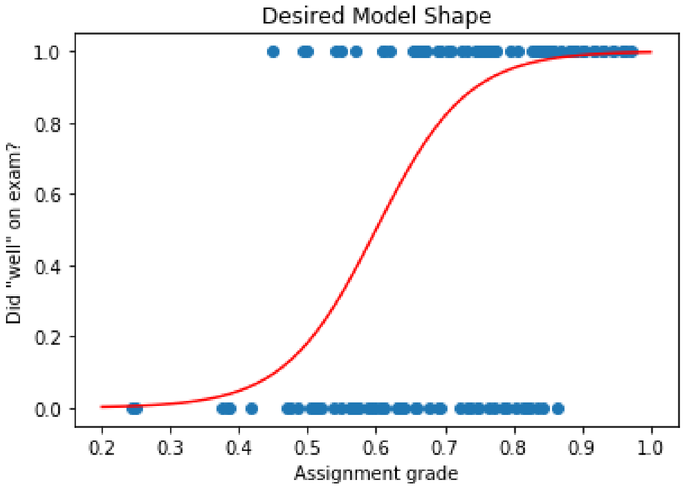
$$\mathcal{L}(y, t) = -t \log(y) - (1-t) \log(1-y)$$

Summary

Hypothesis	$y = \sigma(\mathbf{w}^T\mathbf{x} + b)$
Loss Function	$\mathcal{L}(y, t) = -t\log(y) - (1-t)\log(1-y)$
Optimization Problem	$\min_{\mathbf{w}, b} \mathcal{E}(\mathbf{w}, b)$
Gradient Descent	$\mathbf{w} \leftarrow \mathbf{w} - \alpha\frac{\partial \mathcal{E}}{\partial \mathbf{w}}, b \leftarrow b - \alpha\frac{\partial \mathcal{E}}{\partial b}$

Grade Classification Example

After running gradient descent, we'll get a model that looks something like:



More examples in Tutorial 3!

Project 1

- Handout is posted on the course website
- Should be done on Google Colab
- Start early!