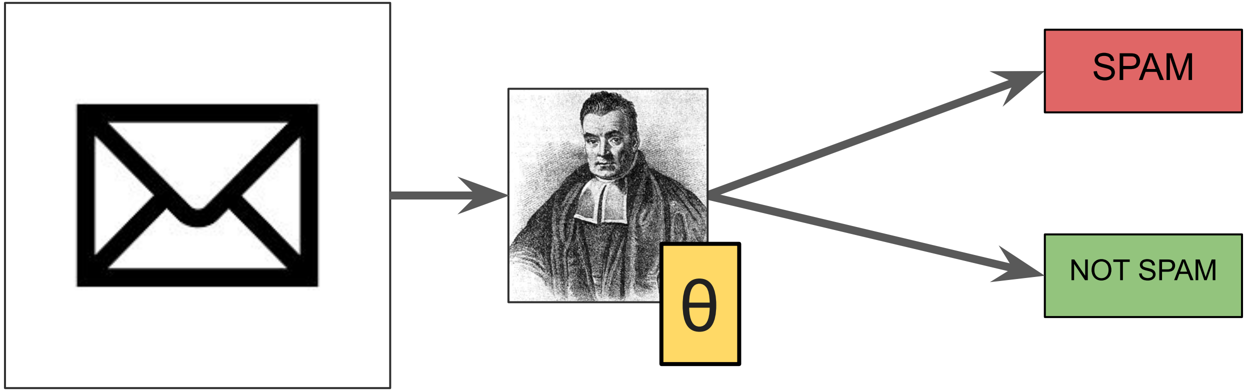
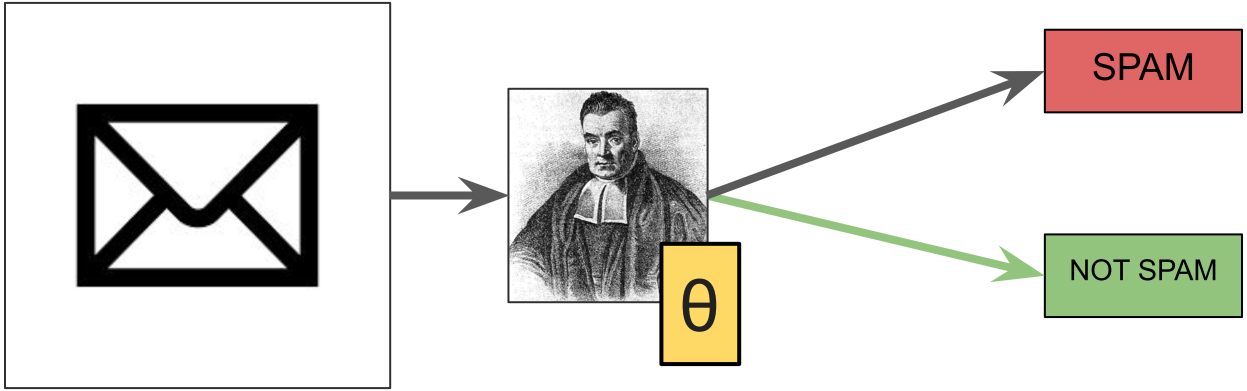
Naive Bayes Classifier

Naive Bayes Classifier

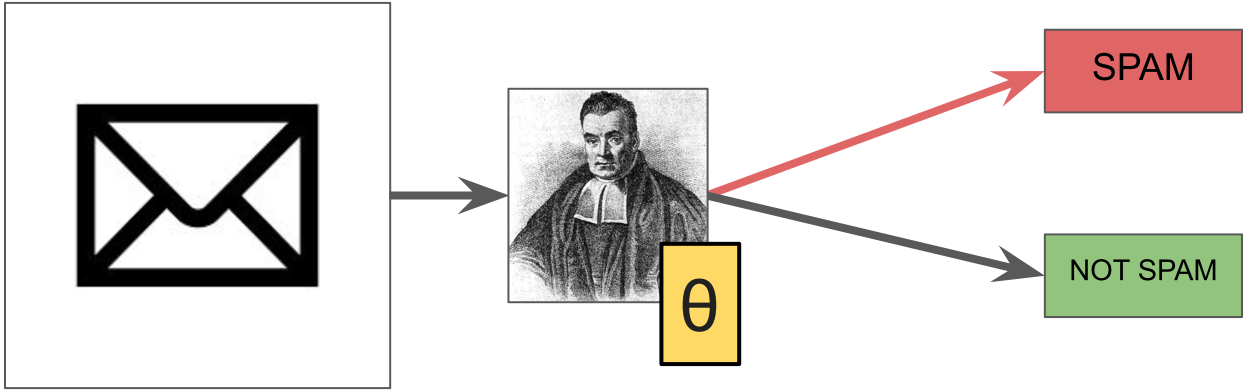


Naive Bayes Classifier



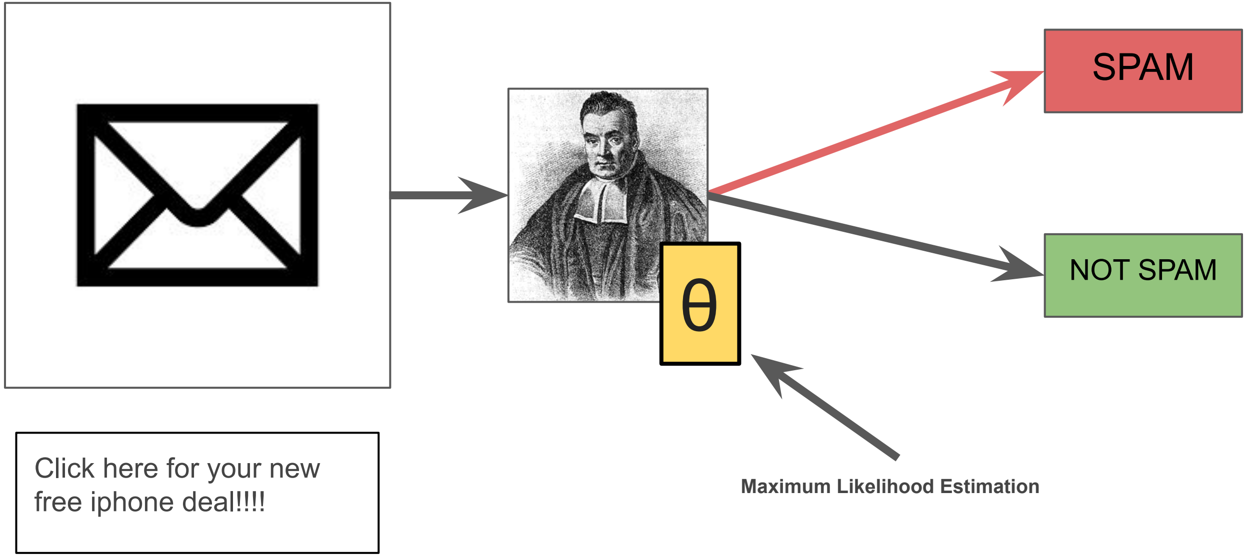
Since this is still the beginning of the semester, here is a quick reminder that the due dates for Lab 1 and Quiz 1 are quickly approaching

Naive Bayes Classifier



Click here for your new free iphone deal!!!!

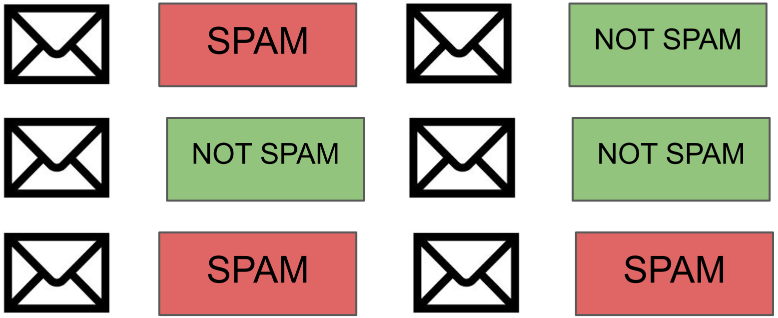
Naive Bayes Classifier



Naive Bayes Classifier

Given a labeled dataset, we want a model that takes as input a text and returns a class label

$$\hat{y} = \arg\max_{y} P(y|\mathbf{x};\theta)$$



Naive Bayes Classifier

We represent text as a bag of words, and the model should return the maximum posterior probability given the text

Applying Bayes' Rule and simplifying:

$$\hat{y} = \arg\max_y P(y|\mathbf{x};\theta) = \arg\max_y \frac{P(y|\mathbf{x})P(y)}{P(\mathbf{x})} = \arg\max_y P(y|\mathbf{x})P(y)$$

Naive Bayes Classifier

We represent text as a bag of words, and the model should return the maximum posterior probability given the text

Applying Bayes' Rule and simplifying:

$$\hat{y} = \arg\max_y P(y|\mathbf{x};\theta) = \arg\max_y \frac{P(y|\mathbf{x})P(y)}{P(\mathbf{x})} = \arg\max_y P(\mathbf{x}|y)P(y)$$

Goal: learn **class prior** and **likelihood** distributions

Naive Bayes Classifier

We represent text as a bag of words, and the model should return the maximum posterior probability given the text

Applying Bayes' Rule and simplifying:

$$\hat{y} = \arg\max_{y} P(y|\mathbf{x};\theta) = \arg\max_{y} \frac{P(y|\mathbf{x})P(y)}{P(\mathbf{x})} = \arg\max_{y} P(\mathbf{x}|y)P(y)$$

Goal: learn **class prior** and **likelihood** distributions

This is easy
to compute

Naive Bayes Classifier

We represent text as a bag of words, and the model should return the maximum posterior probability given the text

Applying Bayes' Rule and simplifying:

This requires more work

$$\hat{y} = \arg\max_y P(y|\mathbf{x};\theta) = \arg\max_y \frac{P(y|\mathbf{x})P(y)}{P(\mathbf{x})} = \arg\max_y P(\mathbf{x}|y)P(y)$$

This is easy to compute

Goal: learn **class prior** and **likelihood** distributions

Naive Bayes Assumptions

Bag of Words

P(y | **x**) = P(SPAM| click, here, for, your, new, …)

= P(y | **x**) = P(SPAM| here, for, click, new, your, …)

= …

Jurafsky and Martin, Ch. 4

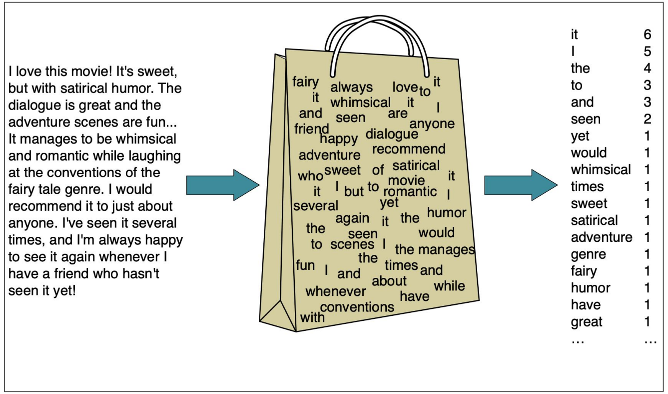


Figure 4.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

Naive Bayes Assumptions

Bag of Words

P(y | **x**) = P(SPAM| click, here, for, your, new, …)

= P(y | **x**) = P(SPAM| here, for, click, new, your, …)

= …

Example of this assumption going wrong?

Jurafsky and Martin, Ch. 4

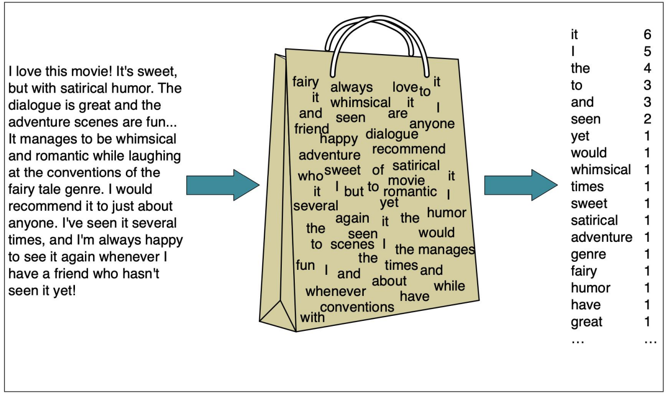


Figure 4.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

Naive Bayes Assumptions

Bag of Words

$P(y \mid \mathbf{x}) = P(\text{SPAM} \mid \text{click, here, for, your, new}, \ldots)$

$= P(y \mid \mathbf{x}) = P(\text{SPAM} \mid \text{here, for, click, new, your}, \ldots)$

$= \ldots$

Example of this assumption going wrong?

What issue do we have here with MLE?

Jurafsky and Martin, Ch. 4

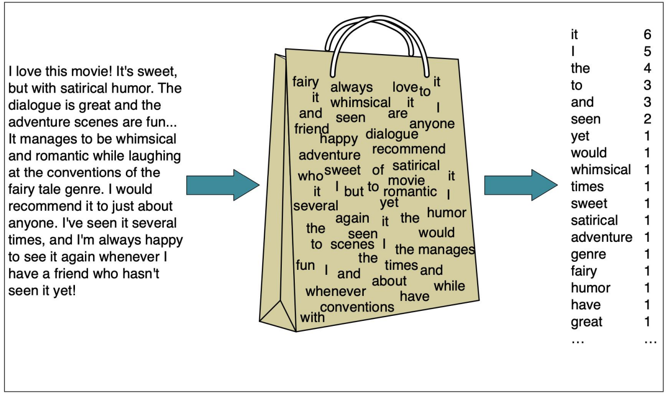


Figure 4.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

Naive Bayes Assumptions

Bag of Words

Multinomial Naive Bayes

Treat each token independent, conditioned on class label

Naive Bayes Assumptions

Bag of Words

Multinomial Naive Bayes
Treat each token independent, conditioned on class label:

P(**x**|y) factors given the label. For example:

P(click, here, for | SPAM) = P(click | SPAM) x P(here | SPAM) x P(for | SPAM)

Naive Bayes Assumptions

Bag of Words

Multinomial Naive Bayes

Treat each token independent, conditioned on class label:

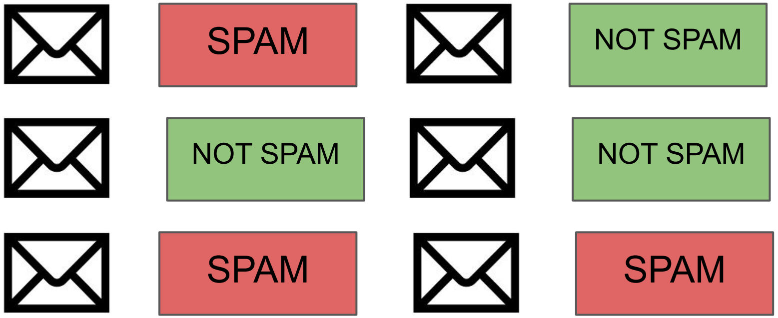
P(**x**|y) factors given the label. For example:

P(click, here, for | SPAM) = P(click | SPAM) x P(here | SPAM) x P(for | SPAM)

Naive Bayes Classifier

To learn the **class priors** P(y), the relative frequency estimates are

P(SPAM) = (# SPAM / K) = 3 / 6

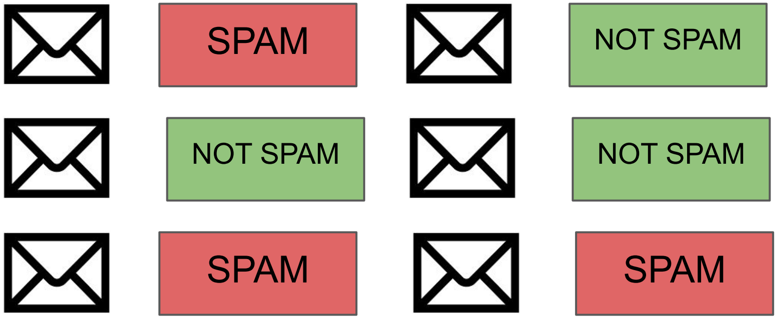


Naive Bayes Classifier

To learn the **class priors** P(y), the relative frequency estimates are

P(SPAM) = (# SPAM / K) = 3 / 6

P(NOT SPAM) = (# NOT SPAM / K) = 3 / 6



Naive Bayes Classifier

To learn the **likelihoods** $P(\mathbf{x}|y)$, the Naive Bayes assumption gives

$P(\mathbf{x}|y) = \prod_j P(x_j|y)$

and the relative frequency for some $P(x_j|y)$ is

$P(x_j|y) = \text{count}(x_j, y) / \text{count}(y)$

Naive Bayes Classifier

To learn the **likelihoods** $P(\mathbf{x}|y)$, the Naive Bayes assumption gives

$P(\mathbf{x}|y) = \prod_j P(x_j|y)$

and the relative frequency for some $P(x_j|y)$ is

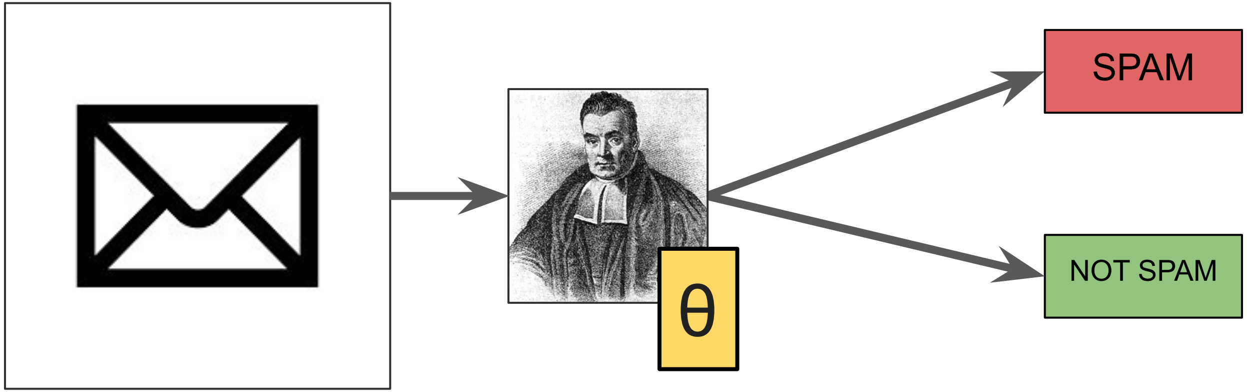
$P(x_j|y) = \text{count}(x_j, y) / \text{count}(y)$

Notice that the product $\prod_j P(x_j|y)$ is zero if any $P(x_j|y)$ is estimated to be zero!

Caveats: Smoothing

We've learned a model but during testing, what if we encounter a word not seen in training?

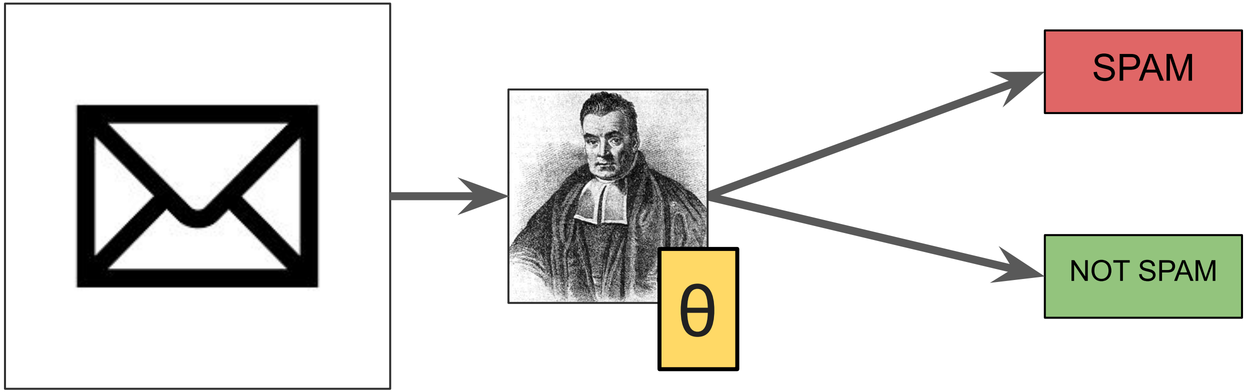
Laplace smoothing: $P(x_j|y) = count(x_j, y) + 1 / count(y) + V + 1$



Caveats: Smoothing

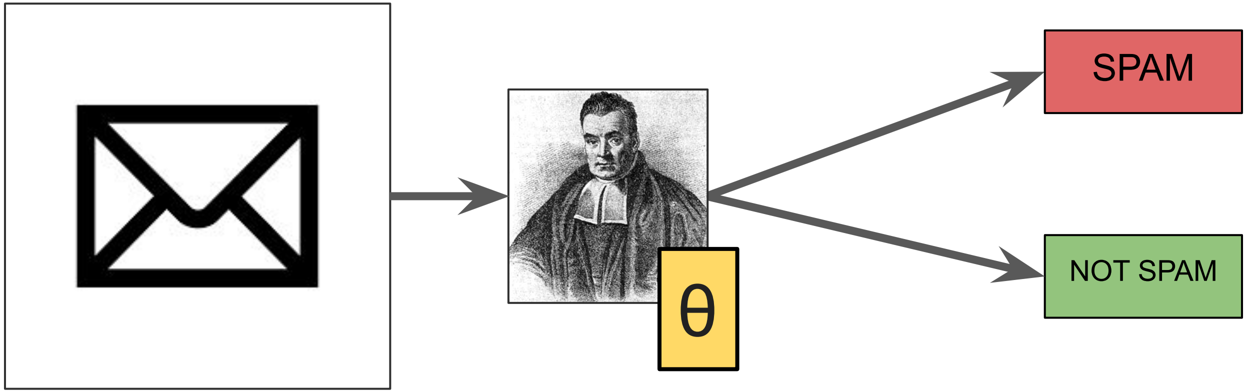
We've learned a model but during testing, what if we encounter a word not seen in training?

Lidstone smoothing: $P(x_j|y) = count(x_j, y) + \alpha$ / $count(y) + \alpha(V + 1)$



Caveats: Underflow

We store log probabilities to avoid numerical underflow



Gradient Descent Optimization

The chicken and the egg

Training neural networks involves **two basic ingredients**

1. Optimizing the parameters of a network with an iterative approach like **gradient descent**
2. Computing the gradient of the loss function efficiently using **backpropagation**

Gradient Descent

We’ve previously talked about the central ingredients to a machine learning problem: data, a model with an associated choice of loss function, and an optimization algorithm

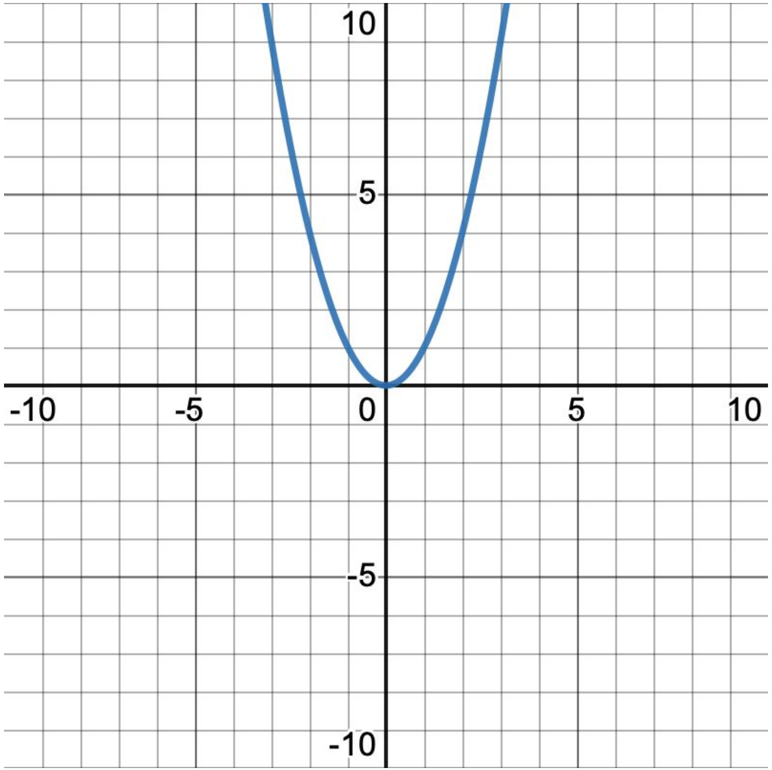
Gradient Descent

We’ve previously talked about the central ingredients to a machine learning problem: data, a model with an associated choice of loss function, and an optimization algorithm

- In supervised learning, we saw that the learning process unfolds using labeled **data**
- We select a **model** that specifies inductive biases that are hopefully relevant to the task (c.f., BOW, Naive Bayes assumptions)
- The **loss function** measures how well the model is predicting the labels
- The **optimization algorithm** describes how the model parameters are adjusted

Gradient Descent

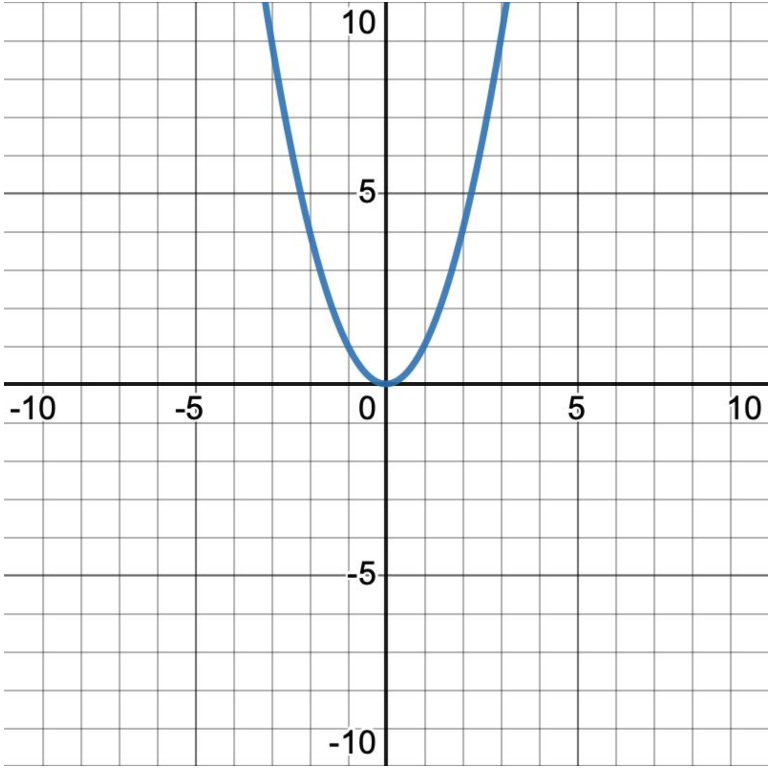
What do we do when we want to determine the optima of a function?



Gradient Descent

What do we do when we want to determine the optima of a function?

How do we find the minimum of $f(x) = x^2$? Exercise: find the minimum.

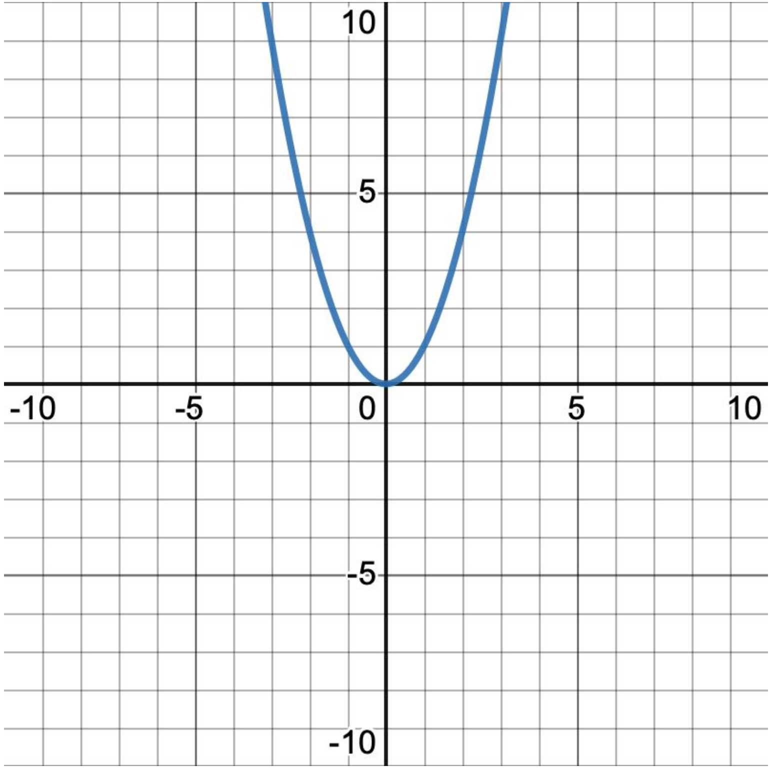


Gradient Descent

What do we do when we want to determine the optima of a function?

How do we find the minimum of $f(x) = x^2$? Exercise: find the minimum.

But in reality, we're using neural networks because they're capable of representing complex relationships—we can't rely on analytical forms for the functions or their derivatives

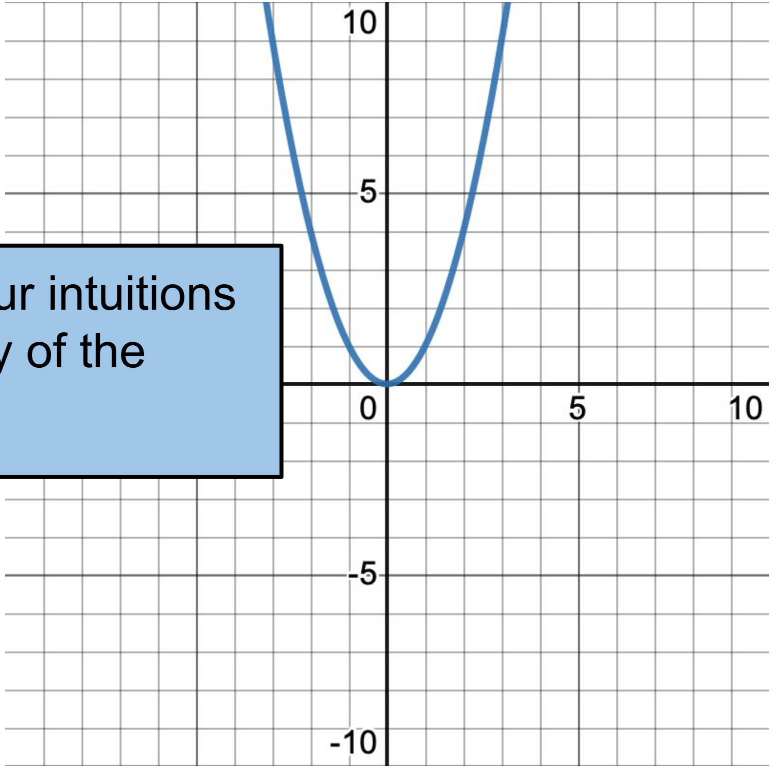


Gradient Descent

What do we do when we want to determine the optima of a function?

How do we find the minimum of $f(x) = x^2$? Exercise: find the

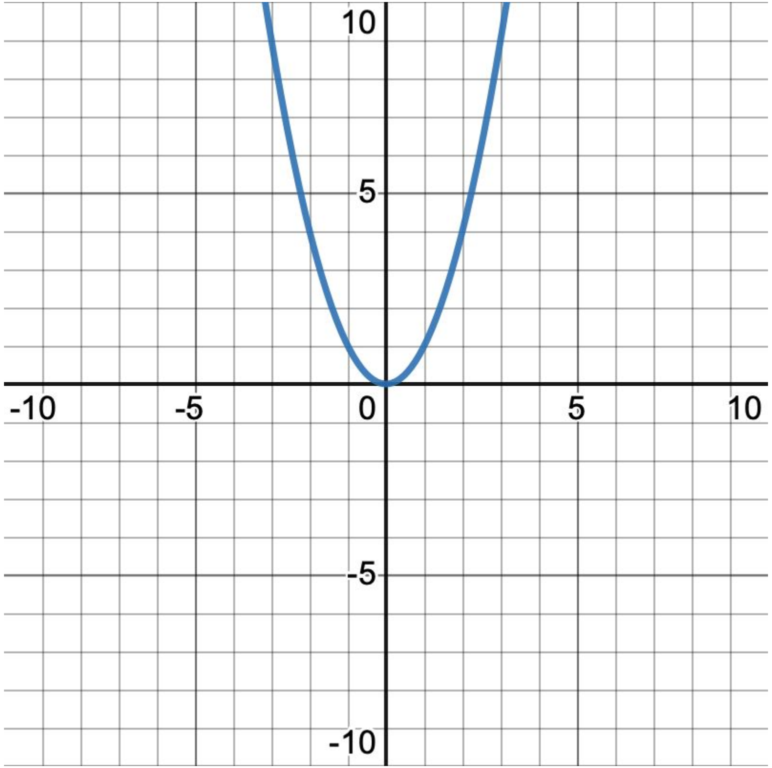
But in reality, we're u
networks because th
representing complex relationships—we can't rely on analytical forms for the functions or their derivatives



Can we hold onto our intuitions when the complexity of the problem grows?

Gradient Descent

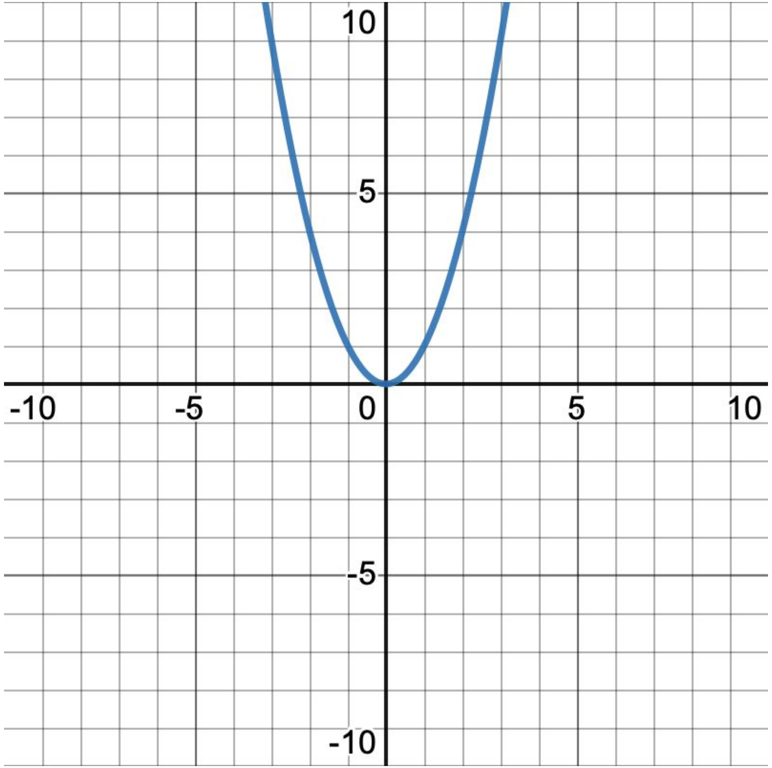
If we can still compute outputs and derivatives of the function for particular inputs, then we can still rely on the same intuition as before, that derivatives provide us with useful optimization information



Gradient Descent

If we can still compute outputs and derivatives of the function for particular inputs, then we can still rely on the same intuition as before, that derivatives provide us with useful optimization information

The non-linearity of neural networks means that the loss functions can be complex (non-convex)

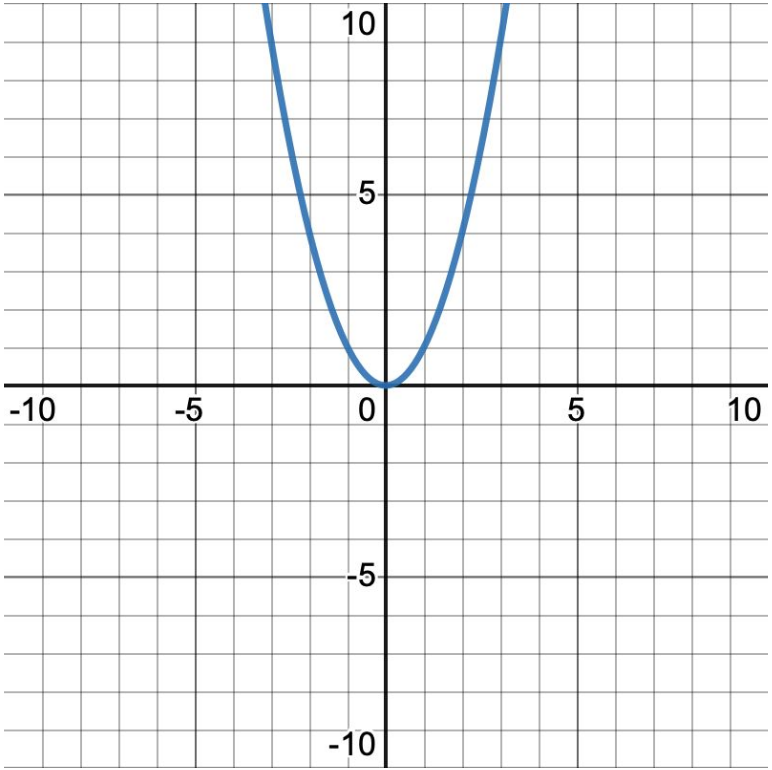


Gradient Descent

If we can still compute outputs and derivatives of the function for particular inputs, then we can still rely on the same intuition as before, that derivatives provide us with useful optimization information

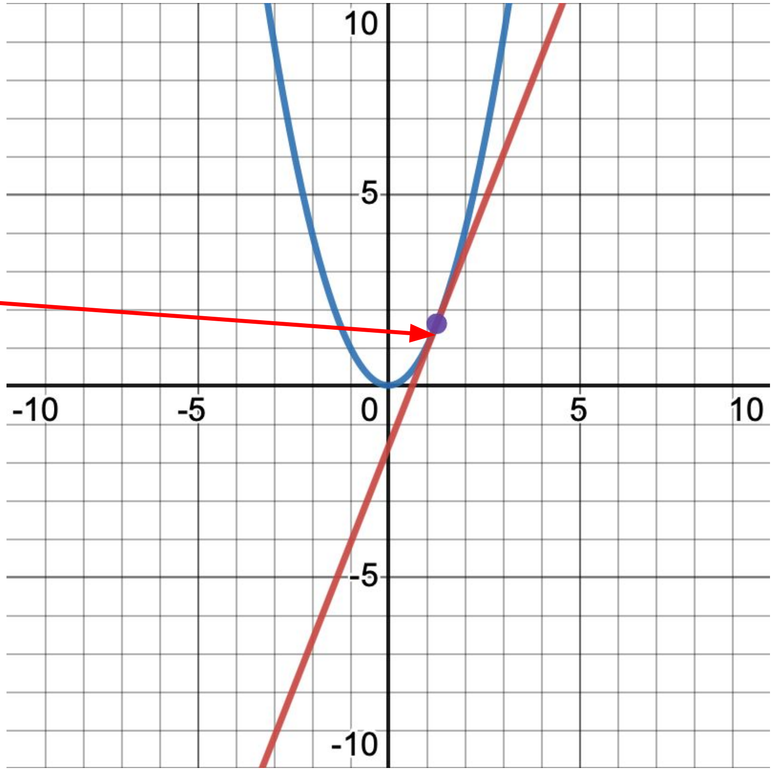
The non-linearity of neural networks means that the loss functions can be complex (non-convex)

So, we will use **iterative methods** that slowly decrement the loss



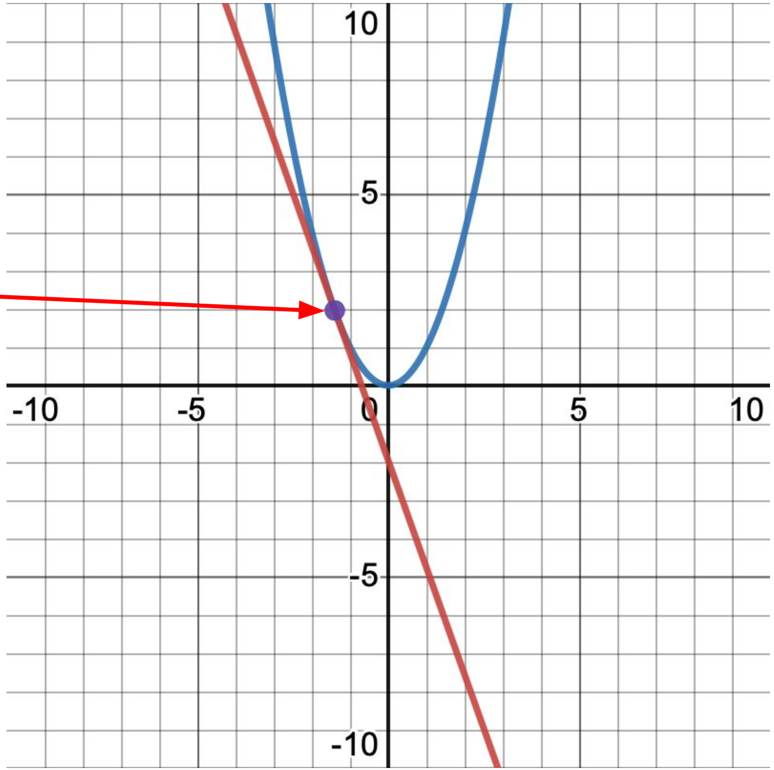
Gradient Descent

If we compute the derivative here, how we should we adjust our estimate of the minimum x?



Gradient Descent

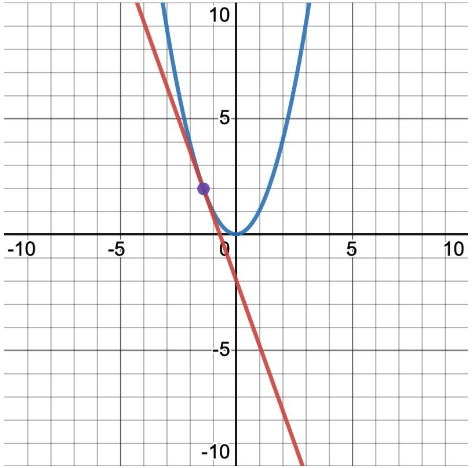
If we compute the derivative here, how we should we adjust our estimate of the minimum x?



Gradient Descent

Procedure:

- Randomly select x
- Evaluate gradient
- Move in the direction of the negative gradient (i.e., opposite the slope in each dimension)



$$\hat{g} \leftarrow \frac{1}{N} \nabla_\theta \sum_i L\left(f(x^{(i)};\theta), y^{(i)}\right)$$

$$\theta \leftarrow \theta - \alpha \hat{g}$$

Stochastic Gradient Descent

The key issue with vanilla/**batch gradient descent** is that the computation of the gradient is based on the entire dataset

$$\hat{g} \leftarrow \frac{1}{N} \nabla_\theta \sum_i L\left(f(x^{(i)}; \theta), y^{(i)}\right)$$

Stochastic Gradient Descent

The key issue with vanilla/**batch gradient descent** is that the computation of the gradient is based on the entire dataset

Stochastic gradient descent computes gradients based on one randomly-chosen example at a time

Stochastic Gradient Descent

The key issue with vanilla/**batch gradient descent** is that the computation of the gradient is based on the entire dataset

Stochastic gradient descent computes gradients based on one randomly-chosen example at a time

Mini-batch gradient descent computes gradients on small random sets of instances called *mini-batches*

$$\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i^m L\left(f(x^{(i)}; \theta), y^{(i)}\right)$$

Stochastic Gradient Descent

The key issue with vanilla/**batch gradient descent** is that the computation of the gradient is based on the entire dataset

Stochastic gradient descent computes gradients based on one randomly-chosen example at a time

Mini-batch gradient descent computes gradients on small random sets of instances called *mini-batches*

Often, authors refer to mini-batch gradient descent and stochastic gradient descent interchangeably

Stochastic Gradient Descent

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \dots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while

Goodfellow, Ch. 8, https://www.deeplearningbook.org/contents/optimization.html

Softmax Regression

Softmax Regression

Recall that the softmax function produces a

probability distribution from a numeric vector

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

When working with neural networks, we call these raw outputs **logits**. They are the inputs to the softmax.

The network outputs a categorical distribution, and through learning, the parameters are adjusted so the model distribution matches the target distribution codified by the labeled data

Softmax Regression

We use the **cross-entropy loss**, which is the negative log likelihood:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} - \mathbb{E}_{\mathbf{x},\mathbf{y} \sim P_D} [\log P(\mathbf{y}|\mathbf{x};\theta)]$$

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Softmax Regression

We use the **cross-entropy loss**, which is the negative log-likelihood:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} - \mathbb{E}_{\mathbf{x},\mathbf{y} \sim P_D} [\log P(\mathbf{y}|\mathbf{x}; \theta)]$$

Note that Naive Bayes is **generative** (since it classifies according to the joint probability of **x** and y), while logistic regression (K=2) and softmax regression (K>3) are **discriminative** (since they are trained to directly classify **x**)

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Softmax Regression

Softmax Regression Demo