Text Classification

Or, A Crash Course in Machine Learning

Aaron Mueller

CAS CS 505: Introduction to Natural Language Processing

Spring 2026

Boston University

Admin

- **Homework 0** (text classification) has been released!
  - Due **Feb. 3**
  - I highly recommend saving your late days for homeworks 1 and 2
  - Please come to office hours or use Piazza for any questions!
    - For coding questions, ask Ge first (office hours on Thursdays).
    - For conceptual questions, ask me first (office hours Tuesdays and Fridays).

- Everyone enrolled in the course should have been added to the Piazza. Please email me if you had any difficulties joining
- Check the syllabus and Piazza for resources for reviewing probability and linear algebra

This is the movie for those who believe cinema is the seventh art, not an entertainment business. Lars von Trier creates a noir atmosphere of post-war Germany utterly captivating. You get absorbed into the dream and you're let go only at the end credits.

Is this a good review or a bad review?

How do you know?

This film was probably inspired by Godard's Masculin, féminin and I urge you to see that film instead.

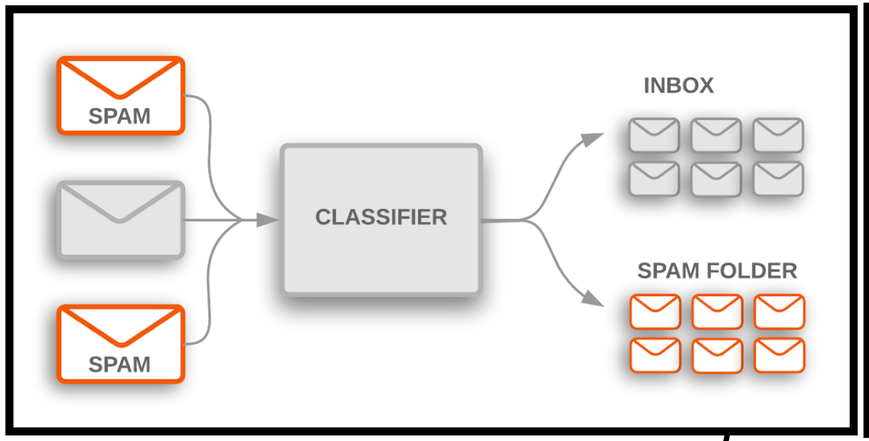
The film has two strong elements and those are, (1) the realistic acting (2) the impressive, undeservedly good, photo. Apart from that, what strikes me most is the endless stream of silliness.

Is this a good review or a bad review?

How do you know?

Text Classification

Text classification is one of the most common applications of NLP:



Spam detection

"I love this movie! I've seen it many times and it's still thrilling."

"The only thing more mediocre than this movie's writing was the acting."

Sentiment classification

رکورد امتیاز نادال در مقابل حریف کانادایی 7–2 است.

听证会也会根据嫌疑人的权利，定下快速审判的日期。

Xabsiga Abu Gharib ee Ciraaq ayaa dab la qabadsiiyay intii rabshaduhu socdeen.

Language identification

Binary classification: Choose from one of two possible classes

Multinomial classification: Choose from one (or more) of ≥ two classes

Overview of Concepts

(Binary/Multinomial) Logistic regression

Machine learning basics:

- Sigmoid
- Softmax
- Gradient descent
- Train/test splits

Evaluation:

- Precision
- Recall
- F1

“I love this movie! I’ve seen it many times and it’s still thrilling.” 👍

“The only thing more mediocre than this movie’s writing was the acting.” 👎

رکورد امتیاز نادال در مقابل حریف کانادایی 7–2 است. 🇮🇷

听证会也会根据嫌疑人的权利，定下快速审判的日期。 🇨🇳

Xabsiga Abu Gharib ee Ciraaq ayaa dab la qabadsiiyay intii rabshaduhu socdeen. 🇸🇴

Text Classification

- The goal of text classification is to take text input x and predict a label $\hat{y}$
  - The set of possible labels is $Y = \{y_1, y_2, \ldots\}$
  - The true label will be denoted y
  - The hat/circumflex notation $\hat{y}$ denotes a predicted or estimated value
  - *Example*: In sentiment classification, the review is x, and Y is {positive, negative} (or, equivalently, {0, 1})

- Today's focus: **supervised machine learning** methods

Supervised Machine Learning

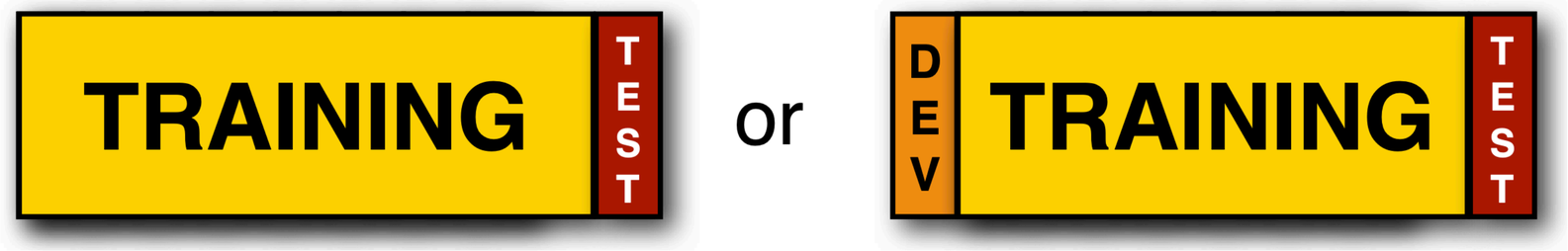
- Given a labeled dataset:

$$D = \{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(n)}, y^{(n)}\right)\}$$

- We want to **train** a system $\gamma : x \rightarrow \hat{y}$ on D
- After training, to produce the correct labels ($\hat{y}^{(i)} = y^{(i)}$) for as many **unseen** inputs $\{x^{(i)} \mid i \notin D\}$ as possible
- **Probabilistic classifiers** return an answer $\hat{y}$, but also probabilities $p(\hat{y} = \hat{y}_j \mid x)$ for all classes j

Machine Learning Datasets

- We always split our data into *disjoint* **train,** (**dev**elopment) and **test** sets:
  - The **train** set is used to optimize the loss function
  - The **test** set is used to evaluate the quality of the classifier



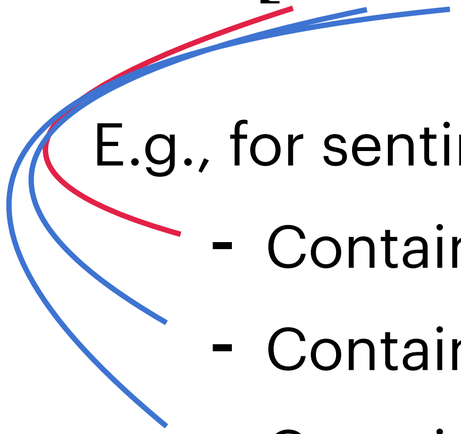
Why shouldn't we use the training set for evaluation?

We wouldn't be able to tell the difference between memorization and generalization

The Machine Learning Pipeline

1. **Feature representations** of the input:

$$\mathbf{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ \ \ldots]$$



E.g., for sentiment classification, we could use features like:

- Contains the word “good”
- Contains the word “bad”
- Contains “not” next to a positive word

The Machine Learning Pipeline

1. **Feature representations** of the input:

$$\mathbf{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ \ \ldots]$$

2. A classification function that computes $p(y \mid \mathbf{x})$, like the **sigmoid** or **softmax**
3. An **objective** or **loss function** that we want to optimize, like **cross-entropy loss**
4. An algorithm for optimizing the loss function, like **stochastic gradient descent**

Logistic Regression

Today, we will focus on **logistic regression**. Why?

Commonly used as a baseline in NLP studies

Common tool in the natural and social sciences

Good introduction to the basics of neural networks

Logistic Regression

- Given an input feature vector:

$$\mathbf{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ \ \ldots]$$

- We want to learn a weighted combination of the features:

$$z = \left(\sum_{i=1}^{n} w_i x_i \right) + b$$

bias term (learned)

weight (learned)

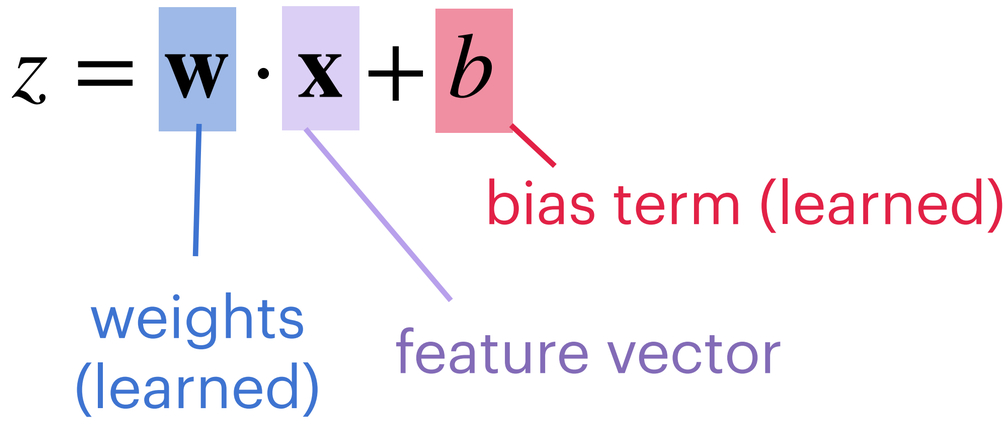
element of feature vector

Logistic Regression

- Given an input feature vector:

$$\mathbf{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ \ \ldots]$$

- We want to learn a weighted combination of the features:



Logistic Regression

- Given an input feature vector:

$$\mathbf{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ \ \ldots]$$

- We want to learn a weighted combination of the features:

- Notice: z is a real number, and doesn't have to be in [0, 1]!

  How do we normalize z to be a probability?

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

bias term (learned)

weights (learned)

feature vector

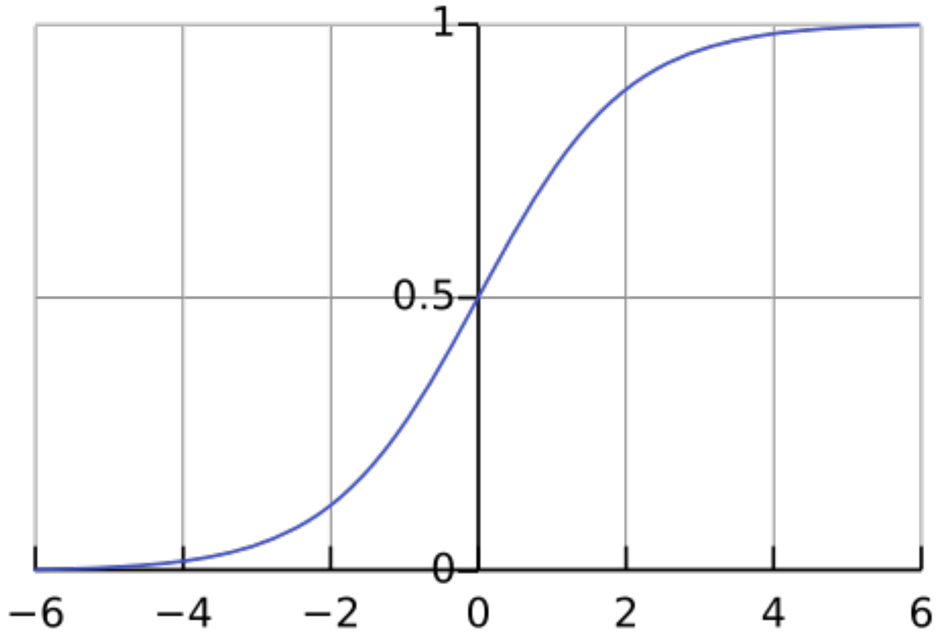
Sigmoids

- The **sigmoid function** (often written as σ) is defined as:

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

$$\equiv$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



- Key properties: monotonic, bounded in [0, 1]
  - $1 - \sigma(x) = \sigma(-x)$

Logistic Regression

- Given an input feature vector:

$$\mathbf{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ \ldots]$$

- We normalize a weighted combination of features to produce a class probability:

$$p(y^{(i)} = 1 \mid \mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

sigmoid

weights (learned)

feature vector

bias term (learned)

$$p(y^{(i)} = 0 \mid \mathbf{x}) = 1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

We generate label $\hat{y}^{(i)} = 1$ if $p(y^{(i)} = 1 \mid \mathbf{x}) > 0.5$

Logistic Regression

- Given an input feature vector:

$$\mathbf{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ \ \ldots]$$

- We normalize a weighted combination of features to produce a class probability:

$$p(y^{(i)} = 1 \mid \mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

logit

- We call it this because the **logit** function is the inverse of the sigmoid:

$$\sigma^{-1}(x) = \ln \frac{p}{1-p}$$

One-hot Encoding

Let's create a vocabulary list of length V of all words:

[a,	the,	John,	Ebenezer,	saw,	yesterday,	dog]
0	1	2	3	4	5	6

A word's feature vector representation will then be a **one-hot vector**, where the only non-zero element is the one corresponding to the word's index:

John: [0, 0, 1, 0, 0, 0, 0]

a: [1, 0, 0, 0, 0, 0, 0]

yesterday: [0, 0, 0, 0, 0, 1, 0]

(We'll see later that the idea of representing words as vectors can be very powerful!)

Bag-of-words Classification

- If we add all the one-hot vectors together, we get a vector of word counts:

"the man saw the dog"

	the	man	saw	dog	amazing
$\mathbf{x}^{(i)} =$ [	2	1	1	1	0]

- This vector contains no information about word order, context, etc.
- What are some ways you could "trick" a classifier based on this representation?
- Let's add a few more features to make this more robust...

Example

Var	Definition
x_1	count(positive lexicon words $\in$ doc)
x_2	count(negative lexicon words $\in$ doc)
x_3	$\begin{cases} 1 & \text{if ``no'' } \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_4	count(1st and 2nd pronouns $\in$ doc)
x_5	$\begin{cases} 1 & \text{if ``!'' } \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_6	ln(word+punctuation count of doc)

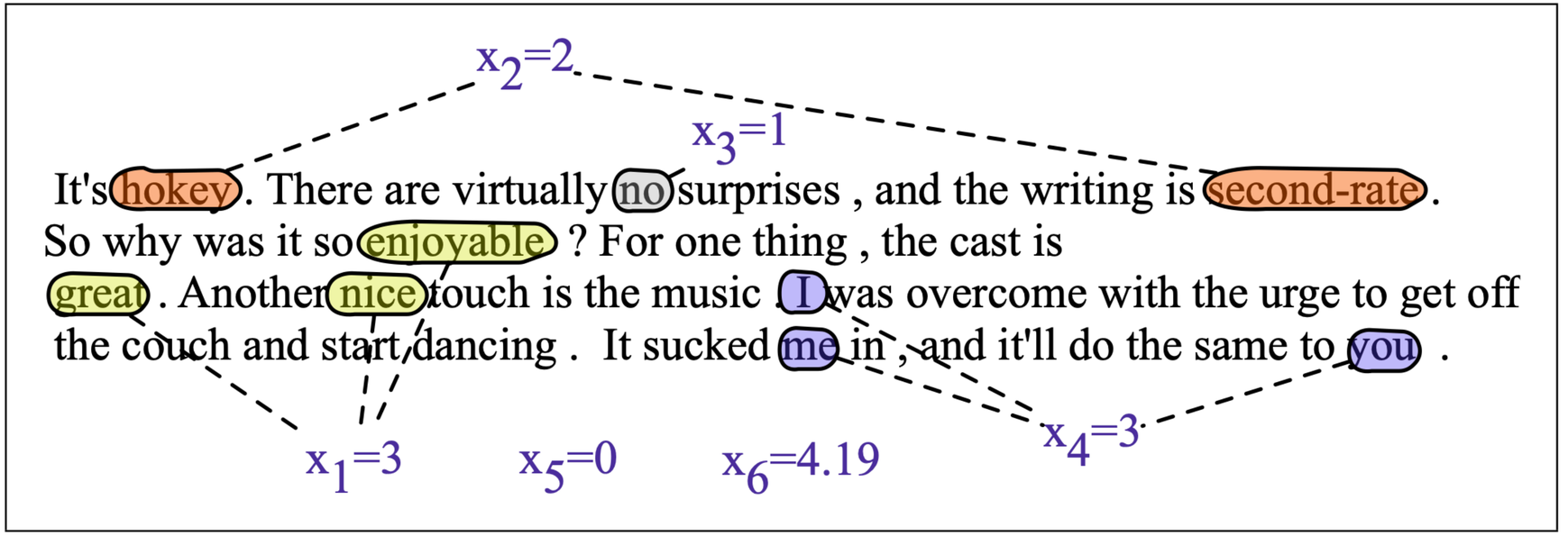
$$\mathbf{x}^{(i)} = [3\ 2\ 1\ 3\ 0\ 4.19]$$

Assume we've learned weights

$$\mathbf{w} = [2.5 \quad -5 \quad -1.2 \quad 0.5 \quad 2.0 \quad 0.7]$$

and bias $b = 0.1$. Then we have:

$$p(y^{(i)} = 1 \mid \mathbf{x}^{(i)}) = \sigma(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$$



$$= \sigma(2.5 \cdot 3 + -5 \cdot 2 + \ldots)$$
$$= \sigma(.833)$$
$$= 0.7$$

0.7 > 0.5, so we generate the label *positive* (1)

Summary: Binary Logistic Regression

- Given:
  - A set of classes, like {negative sentiment, positive sentiment}
  - A vector $\mathbf{x}$ of features $[x_1, x_2, \ldots, x_n]$
    - x_1: could be a word count
    - x_2: could be log(length of document)
  - A vector $\mathbf{w}$ of weights $[w_1, w_2, \ldots, w_n]$ and a bias b

$$p(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$
$$= \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}}$$

Multinomial Logistic Regression

$$p(y^{(i)} = 1 \mid \mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

- The above binary formulation only works when we have exactly two classes to choose from.
- How can we generalize this to more than 2 classes? We'll make two changes:

1. Learn separate weights $\mathbf{w}_k$ and biases b_k for each class $k \in [1, K]$:

$$z_k = \mathbf{w}_k \cdot \mathbf{x}^{(i)} + b_k$$

2. Instead of the sigmoid, we'll use the **softmax**:

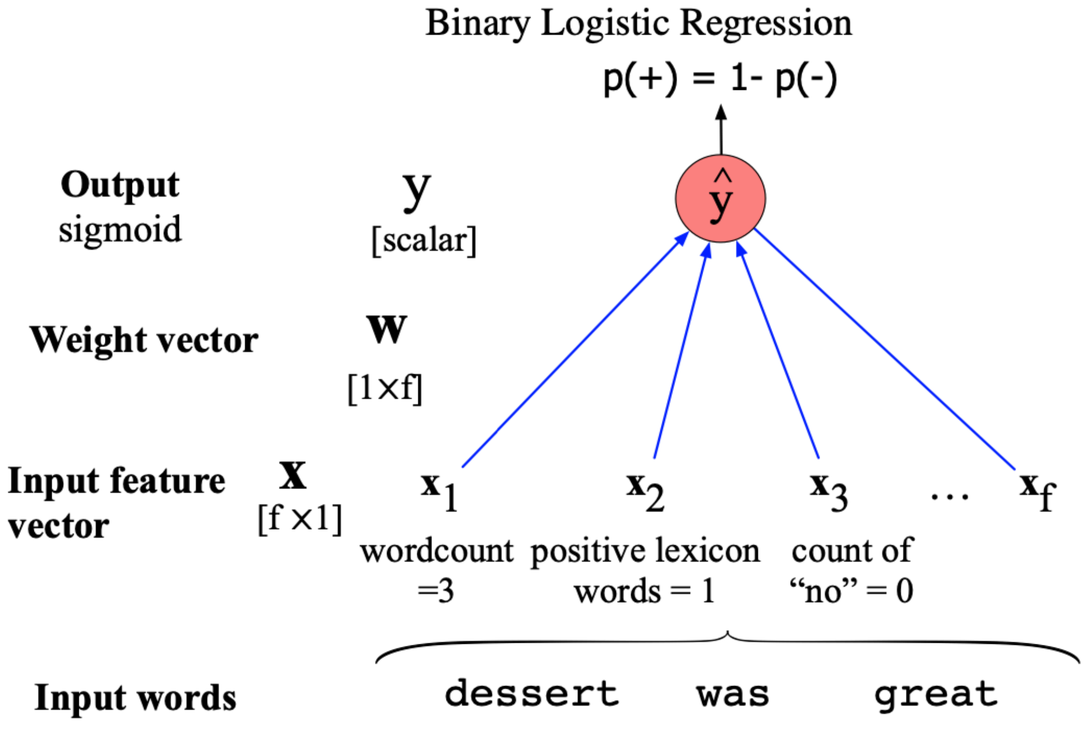
$$\text{softmax}(z_j) = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}$$

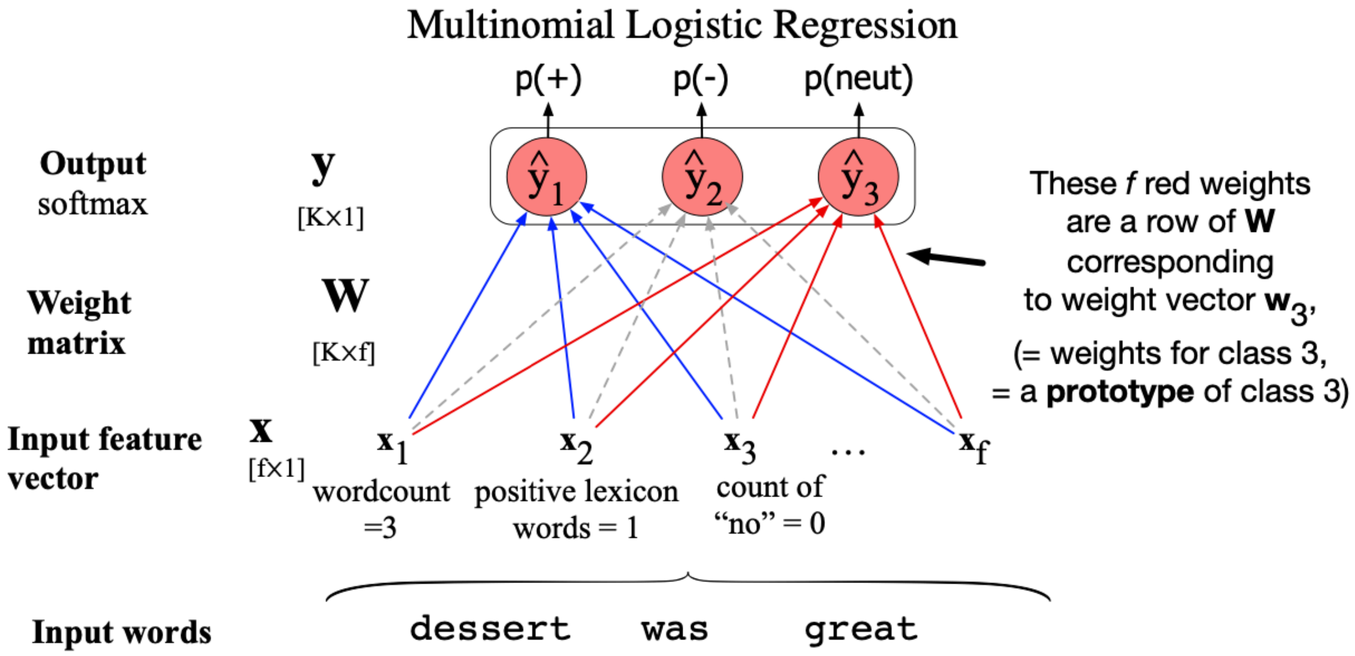
Key property: $\sum_{k=1}^{K} \text{softmax}(z_k) = 1$, plus or minus some rounding error

Features in Multinomial Classifiers

Now, a feature can have separate contributions to each class logit:

Feature	Definition	$w_{5,+}$	$w_{5,-}$	$w_{5,0}$
$f_5(x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	3.5	3.1	−5.3





Learning by Being Less Wrong

We want a metric that tells us how far the classifier's prediction

$$\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

Is from the true output:

$$y \in \{0,1,\ldots\}$$

We'll call this difference $L(\hat{y}, y)$, the **loss**.

Specifically, we'll use the **cross-entropy loss.**

Cross-Entropy

Intuition

Our **objective** is to maximize the likelihood of the training data.

We choose parameters $\mathbf{w}, b$ that maximize

- The log-probability
- of the *true* labels y in the training data
- given observations $\mathbf{x}$.

The **cross-entropy loss** is this log-likelihood, but negated.

(**negative log-likelihood**)

Cross-Entropy

Definition

Goal: maximize probability of the correct label

$$p(y \mid \mathbf{x}) = \hat{y}^{y}(1 - \hat{y})^{1-y}$$

If $y = 1$, this is just $\hat{y}$. If $y = 0$, this is just $1 - \hat{y}$.

To keep the numbers in a reasonable range for a computer, we'll take the log:

$$p(y \mid \mathbf{x}) = y \log \hat{y} + (1 - y)\log(1 - \hat{y})$$

Note: $\log p(y \mid \mathbf{x})$ is proportional to $p(y \mid \mathbf{x})$, so maximizing one is the same as maximizing the other!

Learning by Being Less Wrong

How are the weights and biases of our logistic regression models *learned*?

1. Initialize the weights $\mathbf{w}_k$ and biases b_k to random near-zero values.
2. Apply the regressor to an input $\mathbf{x}^{(i)}$ and measure how wrong its prediction $\hat{y}$ is using a **loss function**. We'll use the **cross-entropy loss:**

$$L_{\text{CE}}(y, \hat{y}) = -\left[y \log \hat{y} + (1 - y) \log(1 - \hat{y})\right]$$

As the model's confidence in the correct answers increases, the loss approaches **0**.

As the model's confidence in the *incorrect* answers increases, the loss approaches ∞.

Learning by Being Less Wrong

How are the weights and biases of our logistic regression models learned?

1. Initialize the weights $\mathbf{w}_k$ and biases b_k to random near-zero values.
2. Apply the regressor to an input $\mathbf{x}^{(i)}$ and measure how wrong its prediction $\hat{y}$ is using a **loss function**. We'll use the **cross-entropy loss:**

$$L_{\text{CE}}(y, \hat{y}) = -\left[y \log \hat{y} + (1 - y) \log(1 - \hat{y})\right]$$

3. Use **gradient descent** to change the weights and biases such that the prediction is less wrong.

Gradient Descent

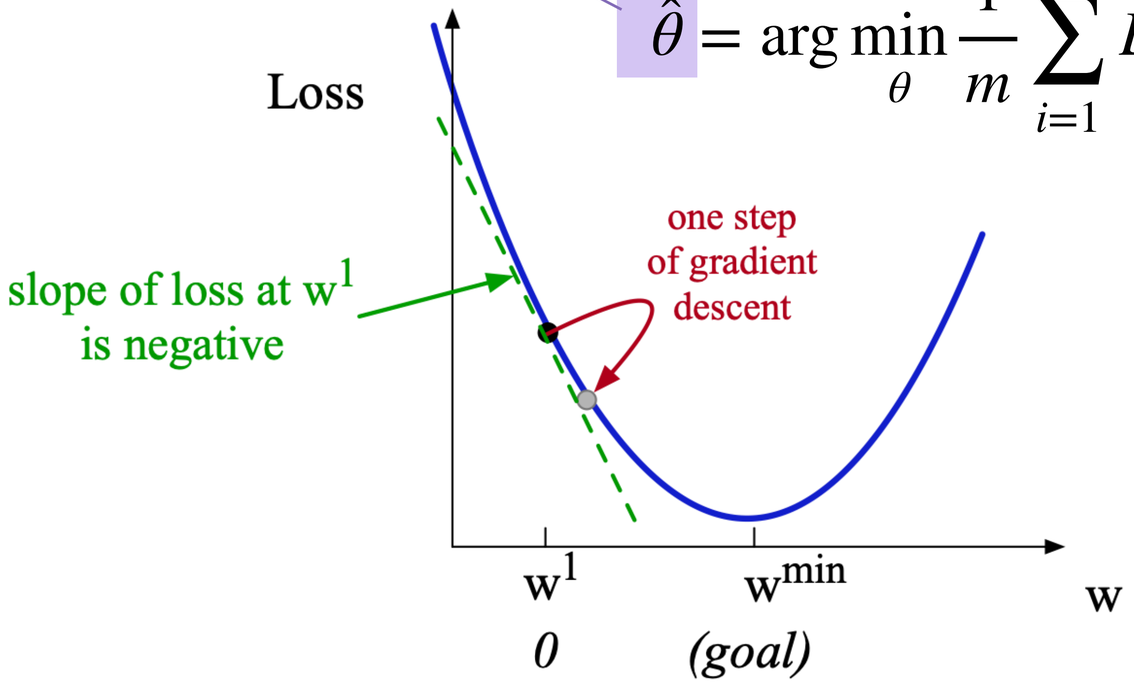
- All learnable components, like weights and biases, are **parameters** θ.
- The goal is to find parameters $\hat{\theta}$ that minimize the loss function:

Optimal parameters

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}\left(f(x^{(i)}; \theta), y^{(i)}\right)$$

Classifier prediction

Correct label



We do this by computing the gradient, and using it to adjust the parameters to reduce the loss.

A **gradient** is like a multi-variable slope.

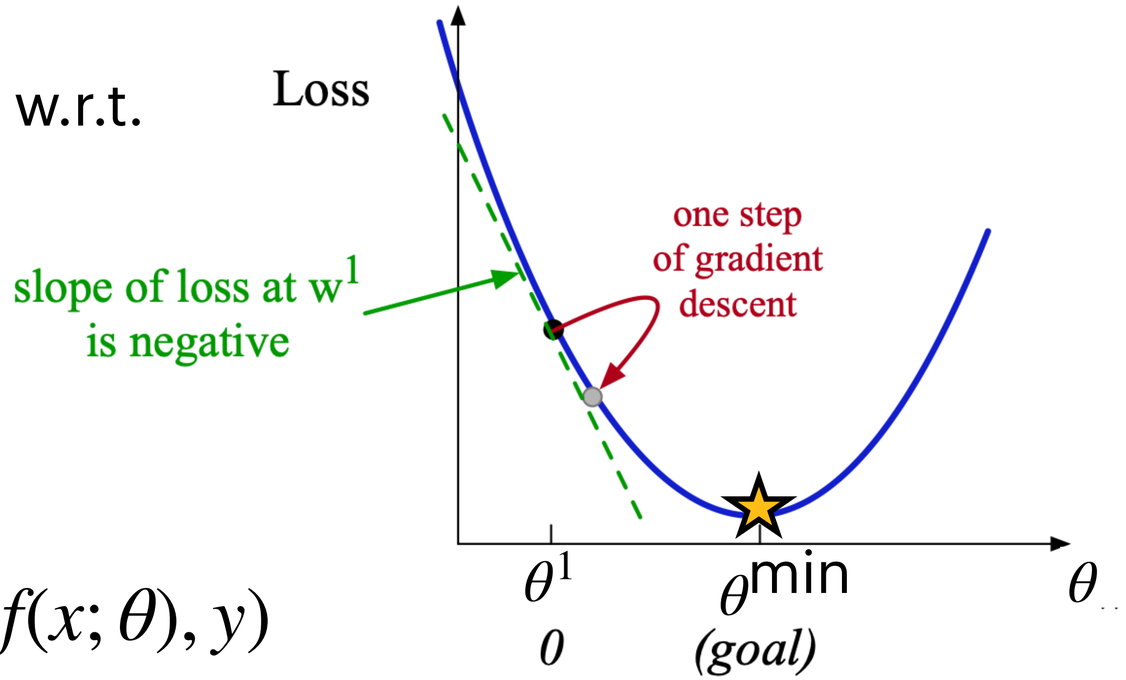
Gradient Descent

1. Compute output of model $\hat{y}$ with current parameters θ.
2. Compute loss $L_{\text{CE}}(f(x;\theta), y)$.
3. Compute the gradient of the loss function w.r.t. the weights: $\frac{d}{d\theta} L(f(x;\theta), y)$
4. Update the weights by subtracting the gradient of the loss w.r.t. the weights:

$$\theta^{t+1} = \theta^t - \alpha \frac{d}{d\theta} L(f(x;\theta), y)$$

learning rate: a coefficient that determines how fast the weights move

Look how nice and **convex**! This function has one obvious minimum solution. (This will not be the case for neural nets.)



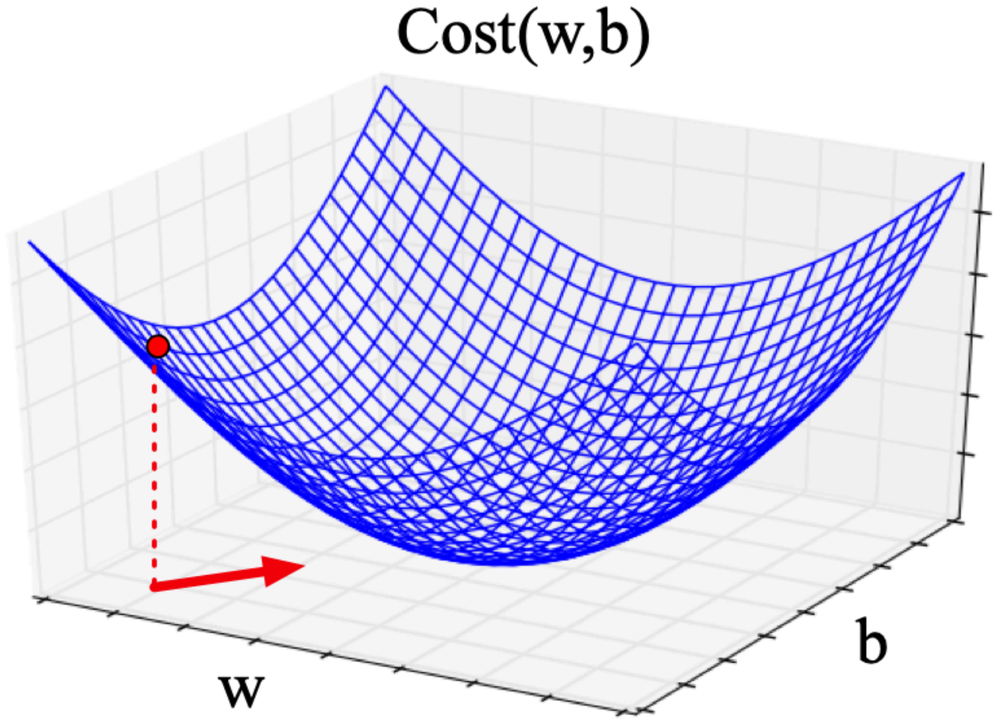
What is a gradient?

We usually update many parameters ($\mathbf{w}$ and b) at once:

$$\theta = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ b \end{bmatrix} \qquad \nabla_\theta L(f(x;\theta), y) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta), y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta), y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta), y) \\ \frac{\partial}{\partial b} L(f(x;\theta), y) \end{bmatrix}$$

denotes the gradient w.r.t. the parameters

Learning in this setting means moving around on this manifold (hopefully to lower points).



Deriving the Gradient for Logistic Regression

- To update our parameters, our model *and* our loss function must be differentiable.

$$L_{\text{CE}} = -\left[y \log \hat{y} + (1-y)\log(1-\hat{y})\right]$$

$$L_{\text{CE}} = -\left[y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b)) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))\right]$$

- In our case, this is true, and thankfully easy to derive for a given weight w_j:

$$\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} = [\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]x_j = (\hat{y}-y)x_j$$

- This is *the difference between the true and predicted label*, weighted by the input feature x_j

Deriving the Gradient for *Multinomial* Logistic Regression

c : correct class

k : any class

K : the set of all classes

Assume this is 1

$$L_{\text{CE}} = -\sum_{k=1}^{K} p(y_k)\log p(\hat{y}_k)$$

$$L_{\text{CE}} = -\log p(\hat{y}_c)$$

softmax

$$L_{\text{CE}} = -\log\left(\frac{\exp(\mathbf{w}_c \cdot \mathbf{x} + b_c)}{\sum_{k=1}^{K}\exp(\mathbf{w}_k \cdot \mathbf{x} + b_k)}\right)$$

$$\frac{\partial L_{\text{CE}}}{\partial w_{k,i}} = -(y_k - \hat{y}_k)x_i = -\left(y_k - \frac{\exp(\mathbf{w}_k \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w}_j \cdot \mathbf{x} + b_j)}\right)x_i$$

Note: the gradient is higher the more incorrect the model is

Stochastic Gradient Descent

- Which input $x^{(i)}$ should we use? Let's just randomly, or **stochastically**, pick one for each update step.
- The **learning rate** α is a **hyperparameter** that can be manually tuned.

A loop over the entire training dataset is called an **epoch**

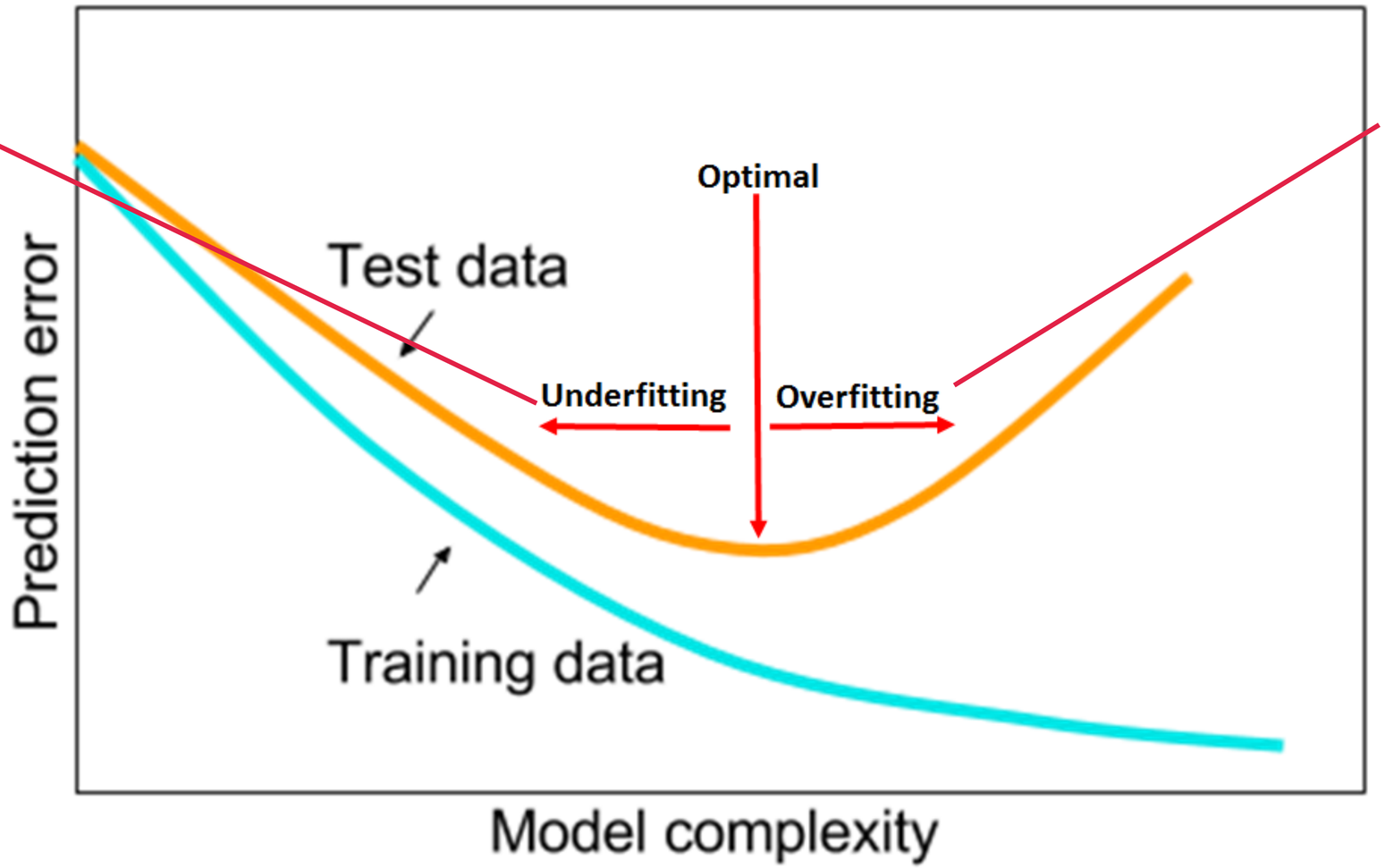
function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

```
    # where: L is the loss function
    #        f is a function parameterized by θ
    #        x is the set of training inputs x(1), x(2),..., x(m)
    #        y is the set of training outputs (labels) y(1), y(2),..., y(m)

θ ← 0          # (or small random values)
```

Underfitting

Because the model doesn't quite learn enough from the training set



Overfitting

Usually because the model memorizes the quirks of the training set, which makes the model fragile

Efficient and Smooth Training with Mini-batches

- Stochastic gradient descent can be slow and choppy because single examples yield highly varying updates
- We could use **batch** training by computing a gradient over entire dataset… but this is memory-intensive and unrealistic
- Let's instead use **mini-batches** by sampling a group of m examples, computing their losses in parallel, and taking the average:

$$L_{\text{CE}}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)})$$

- This implies that the gradient is the average of individual examples' gradients.

Evaluation

A **confusion matrix** shows us how many items that are labeled as a class (rows) actually belonged to which class (columns). High numbers along the diagonal are best.

		gold labels		
		urgent	normal	spam
system output	urgent	8	10	1
	normal	5	60	50
	spam	3	30	200

Evaluation

Confusion matrices can help us diagnose specific failures in a model.

		gold labels		
		urgent	normal	spam
system output	urgent	8	10	1
	normal	5	60	50
	spam	3	30	200

50 spam emails were incorrectly labeled as normal

200 spam emails were correctly labeled

Evaluation

Confusion matrices can help us diagnose specific failures in a model.

		gold labels		
		urgent	normal	spam
system output	urgent	8	10	1
	normal	5	60	50
	spam	3	30	200

50 spam emails were incorrectly labeled as normal

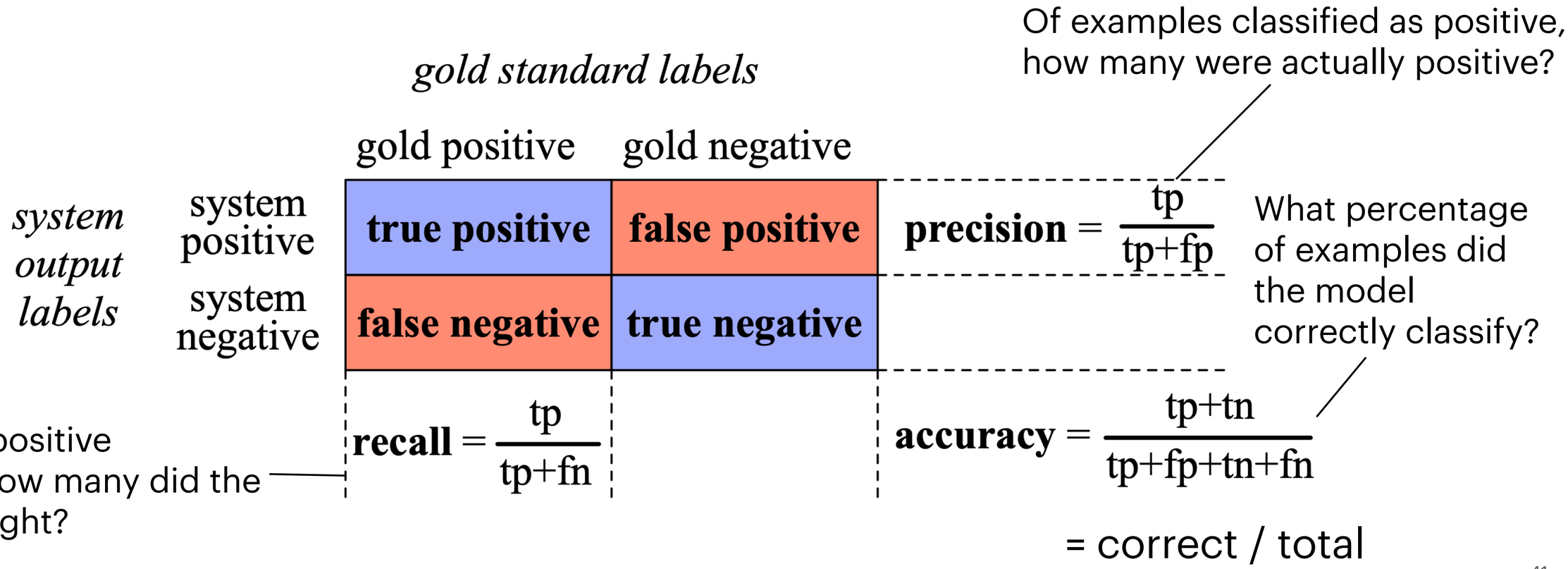
200 spam emails were correctly labeled

3 emails predicted to be spam were actually urgent.

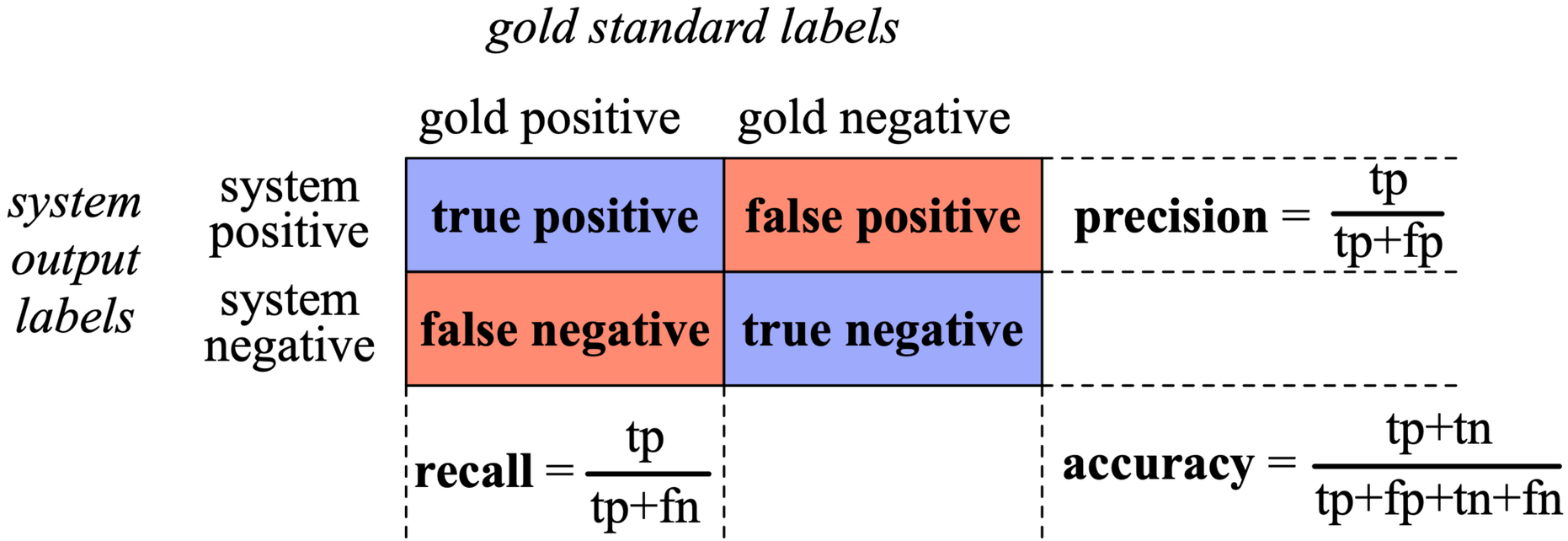
Evaluation

F1

Now we know how to train a good logistic regression model. How do we evaluate its quality?



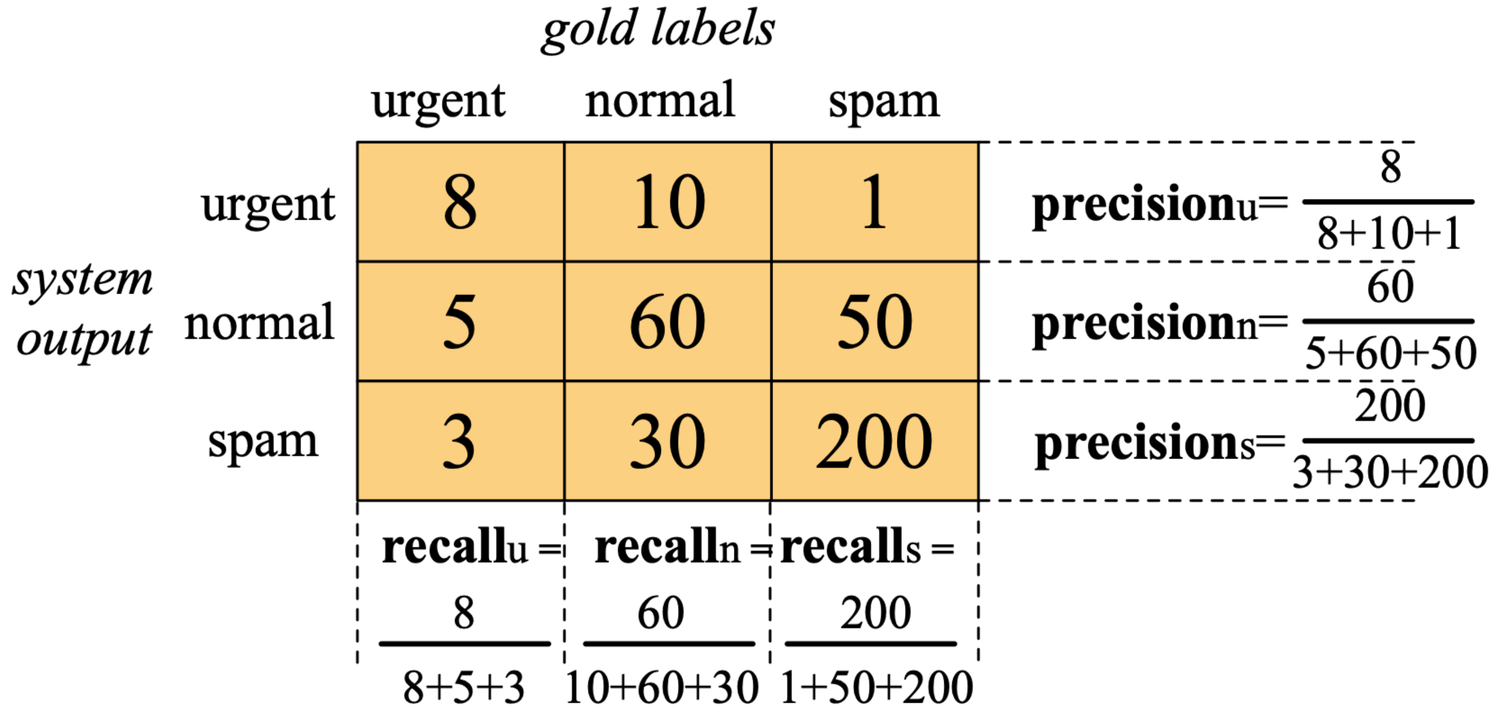
Evaluation



$$F_1 = \frac{2(P \cdot R)}{P + R}$$

We should strongly prefer the F1 score when labels are imbalanced (i.e., there are way more of some labels than others).

Evaluation with >2 Classes



- We can compute separate P and R for each class, and thus separate F1 scores per class
- We could **macroaverage** or **microaverage** these class-specific F1 scores:
  - **Macroaverage**: an average across all *classes*
  - **Microaverage**: an average across *items*

Evaluation with >2 Classes

Class 1: Urgent

	true urgent	true not
system urgent	8	11
system not	8	340

$$\text{precision} = \frac{8}{8+11} = .42$$

Class 2: Normal

	true normal	true not
system normal	60	55
system not	40	212

$$\text{precision} = \frac{60}{60+55} = .52$$

Class 3: Spam

	true spam	true not
system spam	200	33
system not	51	83

$$\text{precision} = \frac{200}{200+33} = .86$$

$$\text{macroaverage precision} = \frac{.42+.52+.86}{3} = \mathbf{.60}$$

Macroaverage: an average across all *classes*

Evaluation with >2 Classes

Class 1: Urgent

	true urgent	true not
system urgent	8	11
system not	8	340

Class 2: Normal

	true normal	true not
system normal	60	55
system not	40	212

Class 3: Spam

	true spam	true not
system spam	200	33
system not	51	83

Pooled

	true yes	true no
system yes	268	99
system no	99	635

Microaverage: an average across *items*

$$\text{microaverage precision} = \frac{268}{268+99} = \mathbf{.73}$$

Social Considerations

- Machine learning is very prone to learning **shortcuts**, rather than robust solutions.
- **Kiritchenko & Mohammad [2018]**: sentiment classifiers often assign lower sentiment to identical documents where traditionally European names (like *Tyler*) are switched out with traditionally African American names (like *Tyrone)*
- Toxicity classifiers often classify as toxic any mention of women **[Park et al., 2018]**, LGBT+ people **[Dixon et al., 2018]**, or even dialectal features of African American English **[Sap et al., 2019; Davidson et al., 2019]**
- *We should not immediately trust the output of an ML system—especially those that are not straightforwardly* ***interpretable.***

Features

- In the previous examples, our feature were handcrafted
- We also did not consider any **feature interactions**
- These days, we usually just throw a neural network at the data and have it learn the features itself via **representation learning**
  - More on this when we get to embeddings

Homework Tips

Avoid for loops over the elements of vectors/matrices! These will make your code too slow to pass the autograder. Use `torch` commands instead.

Here are all the `torch` commands you will need to complete homework 0:

- **`torch.matmul(A, B)`**: multiply matrices **`A`** and **`B`**
- **`torch.outer(A, B)`**: take the outer product of **`A`** and **`B`** (needed in gradient descen
- **`torch.zeros(l)`**: create a vector of length **`l`**, where each element is a 0
- **`torch.max(a)`**: get the maximum value from vector/matrix **`a`**
- **`torch.argmax(a)`**: get the *index of* the maximum value from vector/matrix **`a`**
- **`torch.log(a)`**: take the logarithm of all elements in vector/matrix **`a`**
- **`torch.exp(a):`** exponentiate every element of vector/matrix **`a`** by e. Useful for implementing softmax!

There are also PyTorch tutorials linked in the syllabus. Please check these out and use Piazza/come to office hours with questions!