CS447: Natural Language Processing

http://courses.engr.illinois.edu/cs447

Lecture 5: Logistic Regression

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Today's class

Wrapping up last class: running and evaluating classification experiments

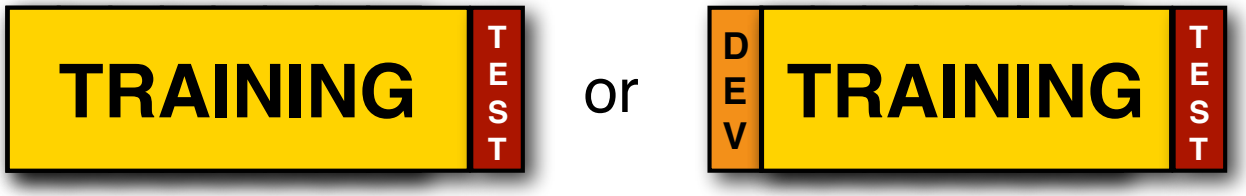
Logistic Regression:

Running and Evaluating Classification Experiments

Evaluating Classifiers

Evaluation setup:

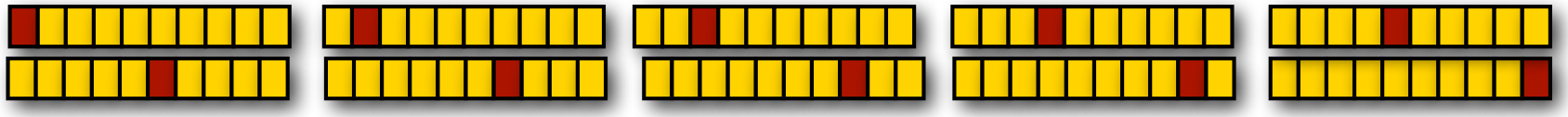
Split data into separate **training,** (**dev**elopment) and **test** sets.



Better setup: **n-fold cross validation**:

Split data into *n* sets of equal size

Run *n* experiments, using set *i* to test and remainder to train



This gives average, maximal and minimal accuracies

When **comparing two classifiers**:

Use the **same** test and training data with the same classes

Evaluation Metrics

Accuracy: How many documents in the test data did you classify correctly?

It's easy to get high accuracy if one class is very common (just label everything as that class)

But that would be a pretty useless classifier

Precision and recall

Precision and recall were originally developed as evaluation metrics for information retrieval:

- **Precision:** What percentage of retrieved documents are relevant to the query?
- **Recall**: What percentage of relevant documents were retrieved?

In NLP, they are often used in addition to accuracy:

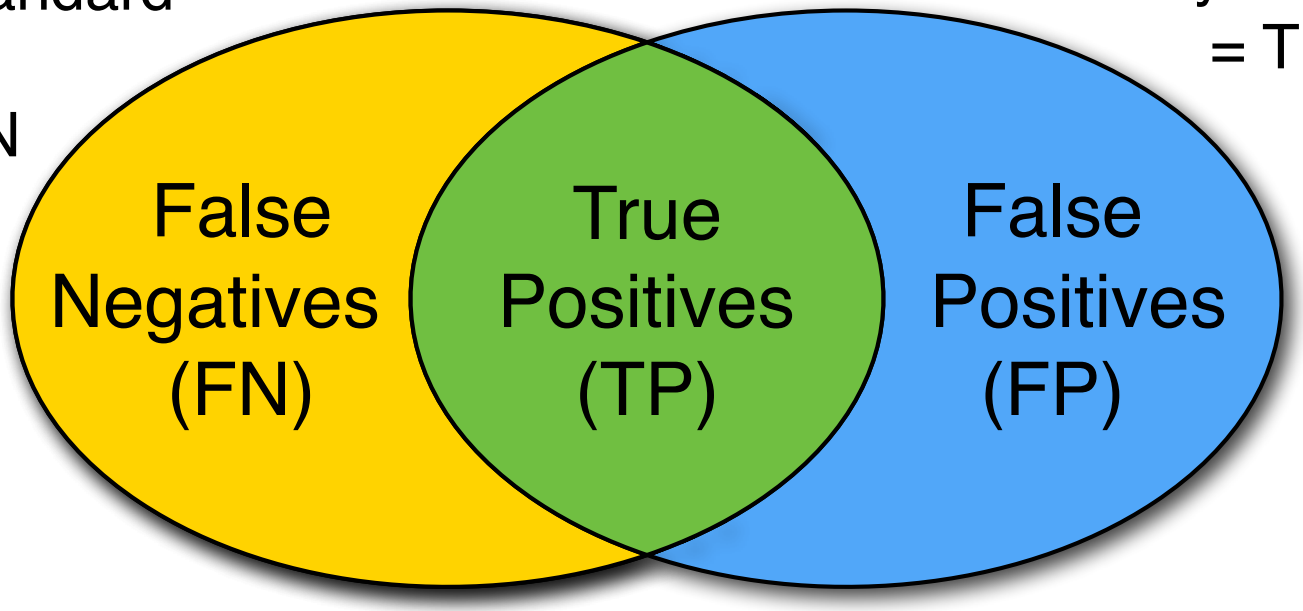
- **Precision:** What percentage of items that were assigned label X do actually have label X in the test data?
- **Recall:** What percentage of items that have label X in the test data were assigned label X by the system?

Particularly useful when there are more than two labels.

True vs. false positives, false negatives

Items labeled X
in the gold standard
('truth')
= TP + FN

Items labeled X
by the system
= TP + FP

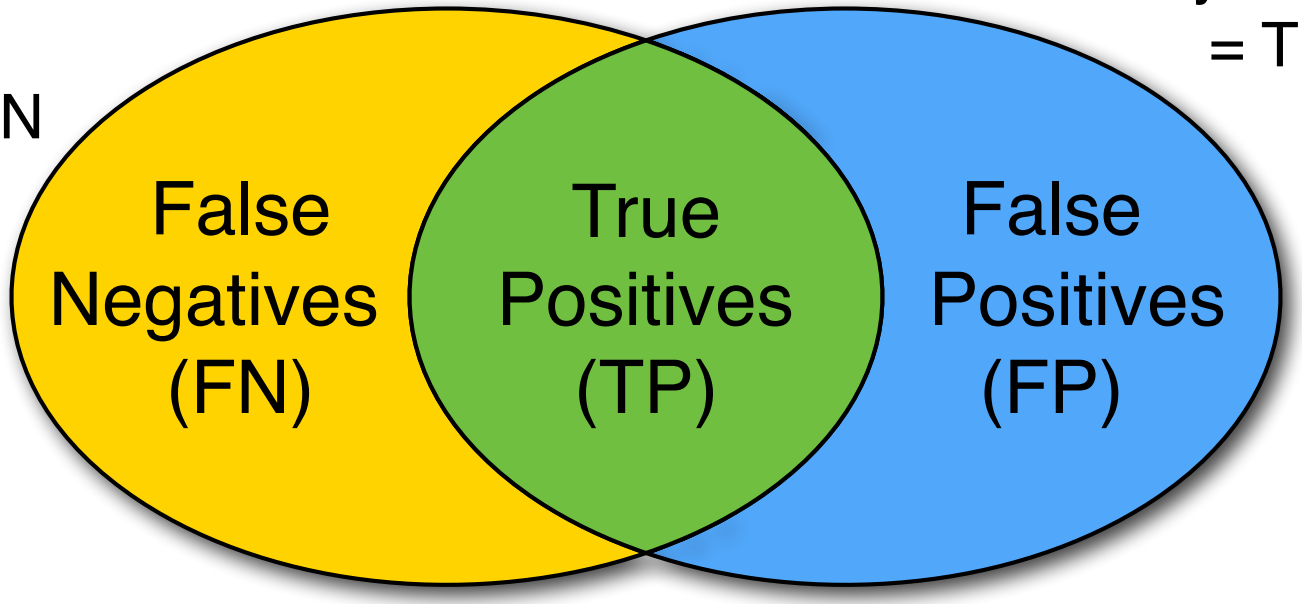


- True positives: Items that were labeled X by the system, and should be labeled X.
- False positives: Items that were labeled X by the system, but should not be labeled X.
- False negatives: Items that were not labeled X by the system, but should be labeled X,

Precision, recall, f-measure

Items labeled X
in the gold standard
('truth')
= TP + FN

Items labeled X
by the system
= TP + FP



Precision: $P = TP / (TP + FP)$

Recall: $R = TP / (TP + FN)$

F-measure: harmonic mean of precision and recall

$$F = (2 \cdot P \cdot R) / (P + R)$$

Confusion matrices

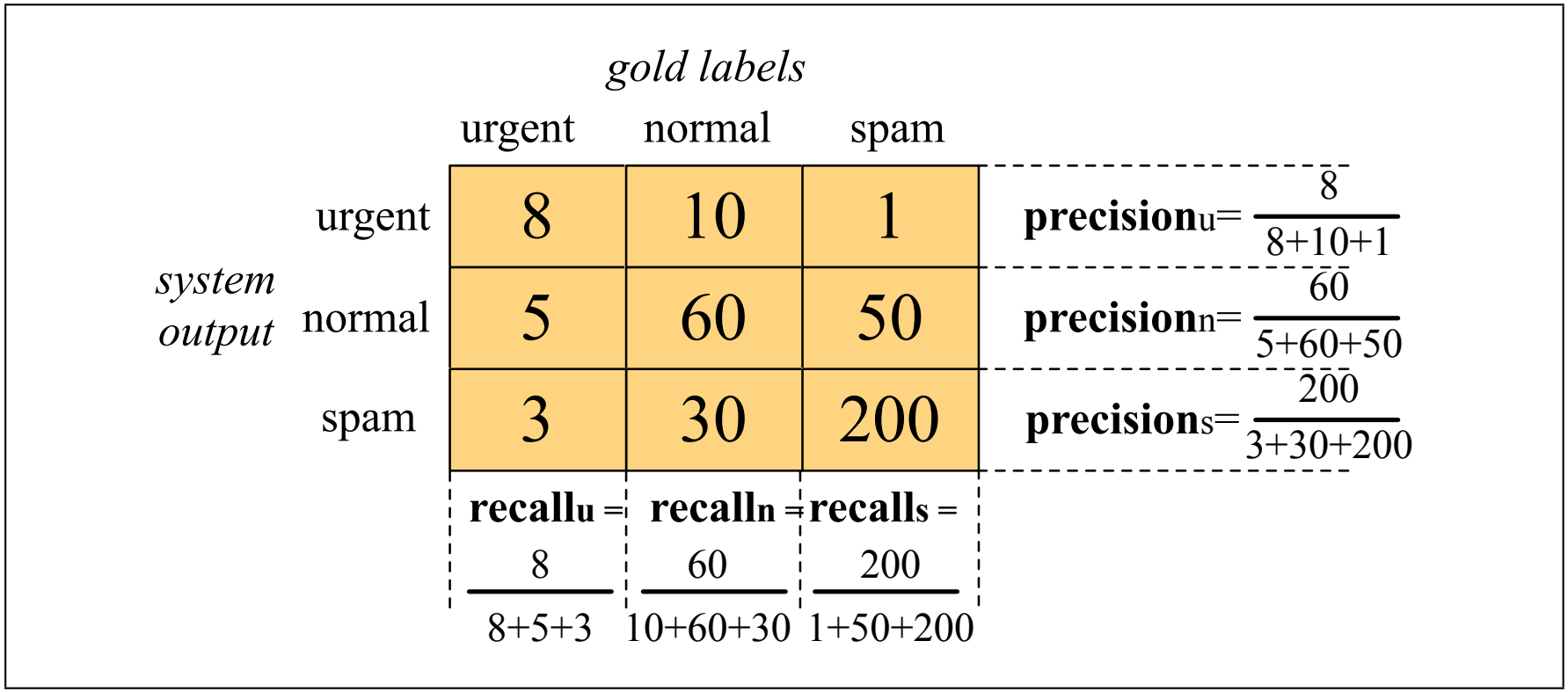


Figure 4.5 Confusion matrix for a three-class categorization task, showing for each pair of classes (c_1, c_2), how many documents from c_1 were (in)correctly assigned to c_2

Confusion matrices

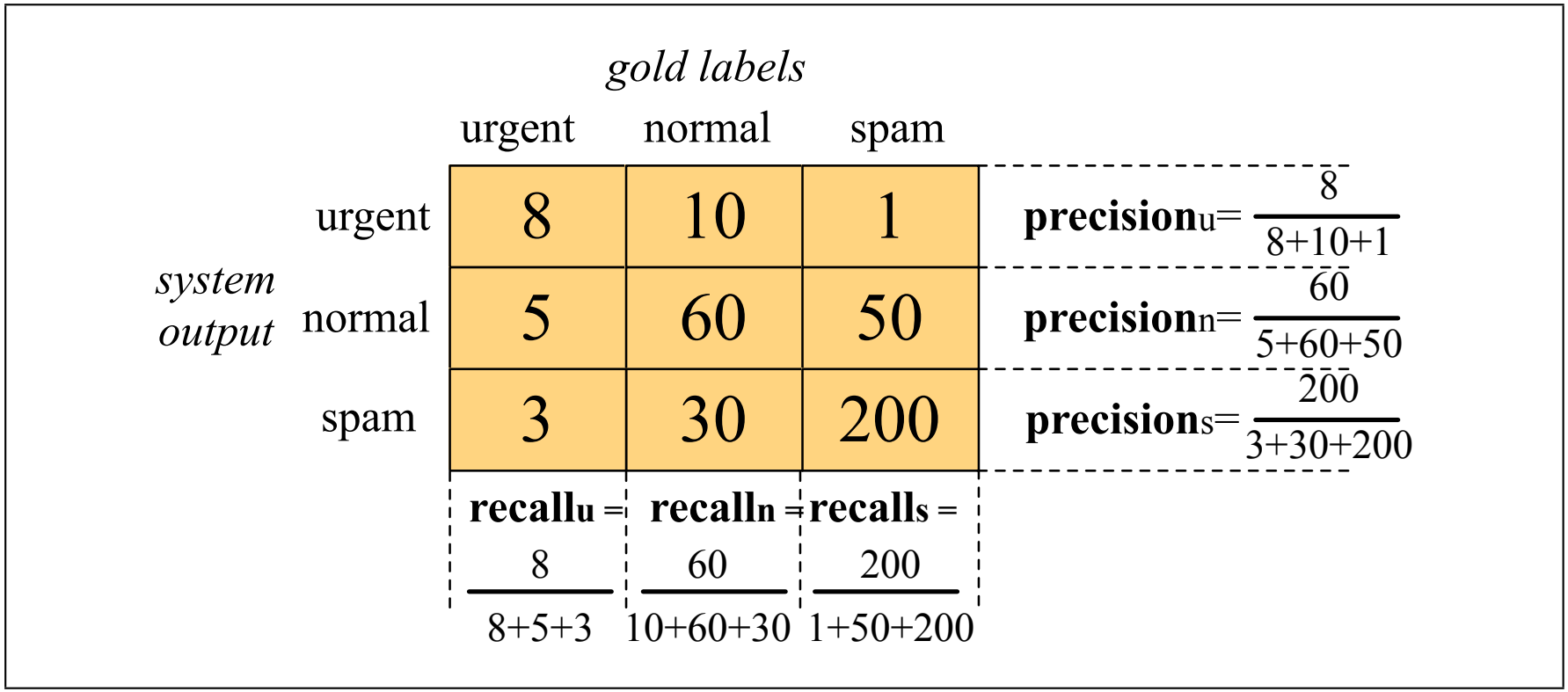


Figure 4.5 Confusion matrix for a three-class categorization task, showing for each pair of classes (c_1, c_2), how many documents from c_1 were (in)correctly assigned to c_2

Macro-average vs Micro-average

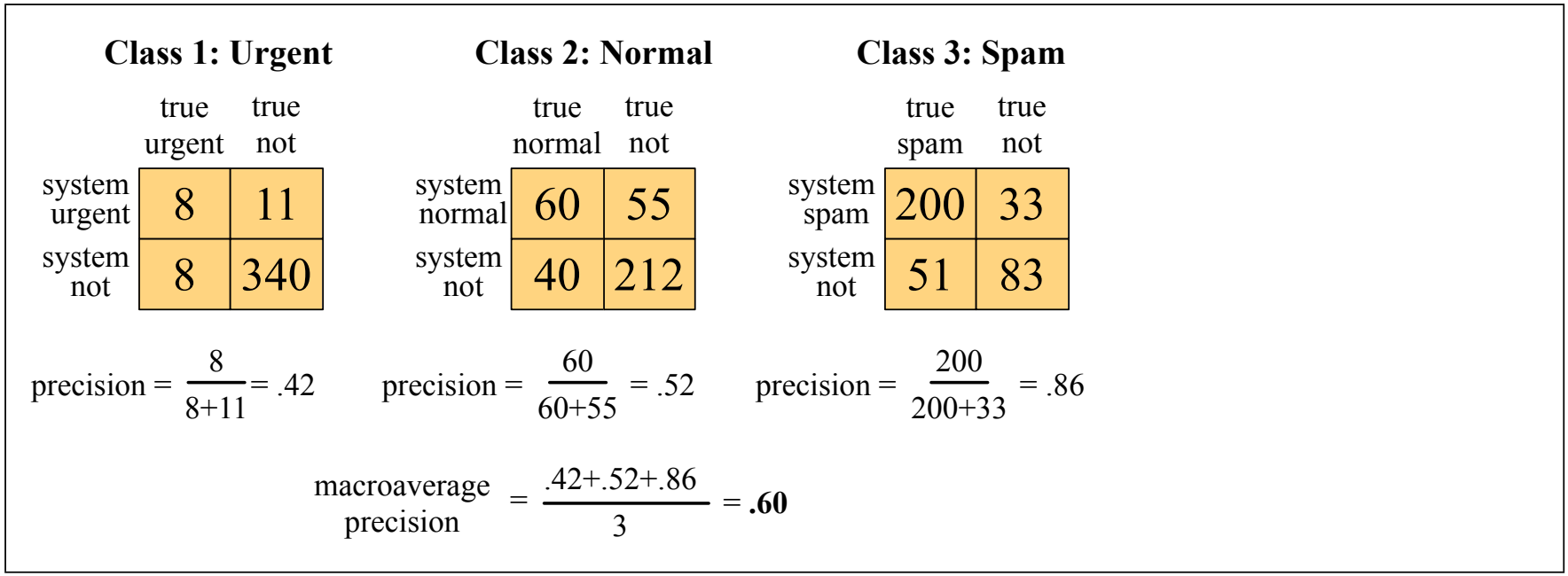


Figure 4.6 Separate contingency tables for the 3 classes from the previous figure, showing the pooled contingency table and the microaveraged and macroaveraged precision.

Macro-average: average the precision over all classes
(regardless of how common each class is)

Micro-average vs Macro-average

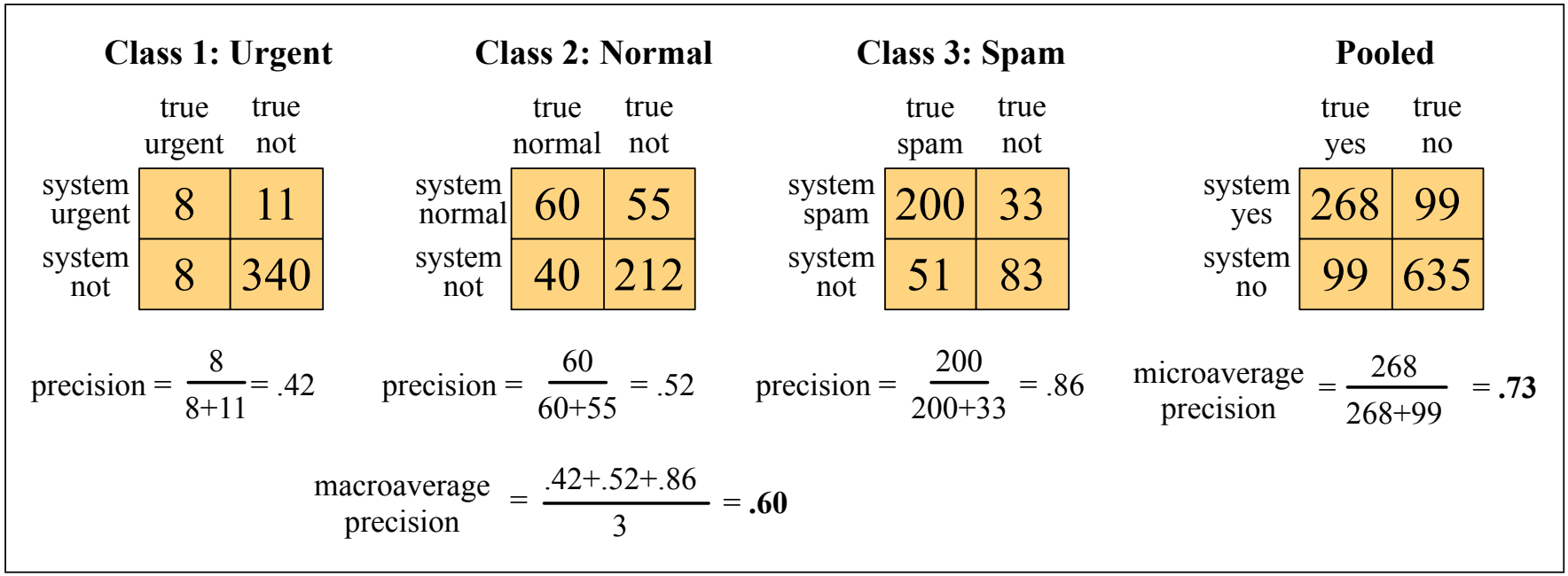


Figure 4.6 Separate contingency tables for the 3 classes from the previous figure, showing the pooled contingency table and the microaveraged and macroaveraged precision.

Macro-average: average the precision over all classes
(regardless of how common each class is)

Micro-average: average the precision over all items
(regardless of which class they have)

Logistic Regression

Probabilistic classifiers

A probabilistic classifier returns the *most likely* class *y* for input **x**:

$$y^* = \text{argmax}_y P(Y = y \mid \mathbf{X} = \mathbf{x})$$

[Last class:] **Naive Bayes** uses Bayes Rule:

$$y^* = \text{argmax}_y P(y \mid \mathbf{x}) = \text{argmax}_y P(\mathbf{x} \mid y) P(y)$$

Naive Bayes models the joint distribution: $P(\mathbf{x} \mid y)\, P(y) = P(\mathbf{x}, y)$

Joint models are also called **generative** models because we can view them as stochastic processes that *generate* (labeled) items:

Sample/pick a label *y* with *P*(*y*), and then an item **x** with *P*(**x**|*y*)

[Today's class:] **Logistic Regression** models $P(y \mid \mathbf{x})$ directly

This is also called a **discriminative** or **conditional** model, because it only models the probability of the class given the input, and not of the raw data itself.

P(Y | **X**) with Logistic Regression

Task: Model $P(y \mid \mathbf{x})$ for any input (feature) vector **x**=(x_1,…,x_n)
Idea: Learn feature weights **w**=(w_1,…,w_n) (and a bias term *b*) to capture how important each feature x_i is for predicting the class *y*

For **binary** classification ($y \in \{0,1\}$), (standard) logistic regression uses the sigmoid function:

$$P(Y=1 \mid \mathbf{x}) = \sigma(\mathbf{w}\mathbf{x}+b) = \frac{1}{1+\exp(-(\mathbf{w}\mathbf{x}+b))}$$

Parameters to learn: one feature weight vector **w** and one bias term *b*

For **multiclass** classification ($y \in \{0,1,...,K\}$), multinomial logistic regression uses the softmax function:

$$P(Y=y_i \mid \mathbf{x}) = \text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{K}\exp(z_j)} = \frac{\exp(-(\mathbf{w}_i\mathbf{x}+b_i))}{\sum_{j=1}^{K}\exp(-(\mathbf{w}_j\mathbf{x}+b_j))}$$

Parameters to learn: one feature weight vector **w** and one bias term *b* ***per class.***

Binary logistic regression is just a special case of multinomial logistic regression

Binary logistic regression needs a distribution over $y \in \{0,1\}$:

$$P(Y=1 \mid \mathbf{x}) = \frac{1}{1+\exp(-(\mathbf{w}\mathbf{x}+b))}$$

$$P(Y=0 \mid \mathbf{x}) = \frac{\exp(-(\mathbf{w}\mathbf{x}+b))}{1+\exp(-(\mathbf{w}\mathbf{x}+b))} = 1 - P(Y=1 \mid \mathbf{x})$$

Compare with **Multinomial logistic regression** over $y \in \{0,1\}$:

$$P(Y=1 \mid \mathbf{x}) = \frac{\exp(-(\mathbf{w}_1\mathbf{x}+b_1))}{\exp(-(\mathbf{w}_1\mathbf{x}+b_1)) + \exp(-(\mathbf{w}_0\mathbf{x}+b_0))}$$

$$P(Y=0 \mid \mathbf{x}) = \frac{\exp(-(\mathbf{w}_0\mathbf{x}+b_0))}{\exp(-(\mathbf{w}_1\mathbf{x}+b_1)) + \exp(-(\mathbf{w}_0\mathbf{x}+b_0))}$$

➜ Think of binary lr. as multinomial lr. with $\exp(-(\mathbf{w}_1\mathbf{x}+b_1)) = 1$
(i.e. where $\mathbf{w}_1$ is set to the null vector and $b_1 := 0$)

Using Logistic Regression

How do we create a (binary) logistic regression classifier?

1) **Design**: Decide how to map raw inputs to feature vectors **x**
2) **Training**: Learn parameters **w** and *b* on training da
3) **Testing**: Use the classifier to classify unseen inputs

Feature Design: from raw inputs to feature vectors **x**

In a generative model, we have to learn a model for *P*(**x** | y).
To guarantee that we get a proper distribution ($\sum_{\mathbf{x}} P(\mathbf{x} \mid y) = 1$), we have to assume that the features (elements of **x**) are independent (more precisely, conditionally independent given y),
In a conditional model, we only have to learn *P*(y | **x**), not for *P*(**x** | y).

Advantage: Because we don't need a distribution over **x**, we do not need to assume that our features $x_1,\ldots,x_n$ are independent.

Feature Design:
From raw inputs to feature vectors **x**

Feature design for generative models (Naive Bayes):

- In a generative model, we have to learn a model for $P(\mathbf{x} \mid y)$.
- Getting a proper distribution ($\sum_{\mathbf{x}} P(\mathbf{x} \mid y) = 1$) is difficult
- NB assumes that the features (elements of **x**) are independent* and defines $P(\mathbf{x} \mid y) = \prod_i P(x_i \mid y)$ via a multinomial or Bernoulli

(*more precisely, conditionally independent given y)

- Different kinds of feature values (boolean, integer, real) require different kinds of distributions $P(x_i \mid y)$ (Bernoulli, multinomial, etc.)

Feature design for conditional models (Logistic Regression):

- In a conditional model, we only have to learn $P(y \mid \mathbf{x})$
- It is much easier to get a proper distribution ($\sum_y P(y \mid \mathbf{x}) = 1$)
- We don't need to assume that our features are independent
- Any numerical feature x_i can be used to compute $\exp(w_j x_i)$

Useful features that are not independent

Different features can *overlap* in the input

(e.g. we can model both unigrams and bigrams, or overlapping bigrams)

Features can capture *properties* of the input

(e.g. whether words are capitalized, in all-caps, contain particular [classes of] letters or characters, etc.)

This also makes it easy to use predefined dictionaries of words (e.g. for sentiment analysis, or gazetteers for names):

Is this word "positive" ('*happy*') or "negative" ('*awful*')?

Is this the name of a person ('Smith') or city ('Boston') [it may be both ('Paris')]

Features can capture *combinations* of properties

(e.g. whether a word is capitalized *and* ends in a full stop)

We can use the outputs of other classifiers as features

(e.g. to combine weak [less accurate] classifiers for the same task, or to get at complex properties of the input that require a learned classifier)

Feature Design and Selection

How do you specify features?

We can't manually enumerate 10,000s of features
(e.g. for every possible bigram: "*an apple*", …, "*zillion zebras*")

Instead we use **feature templates** that define what type of feature we want to use
(e.g. "*any pair of adjacent words that appears >2 times in the training data*")

How do you know which features to use?

Identifying useful sets of feature templates requires **expertise** and a lot of **experimentation** (e.g. ablation studies)
Which specific set of feature (templates) works well depends very much on the particular classification task and dataset.

Feature selection methods prune useless features automatically. This reduces the number of weights to learn.
(e.g. '*of the*' may not be useful for sentiment analysis, but '*very cool*' is)

Learning parameters **w** and *b*

Training objective: Find parameters **w** and *b* that "capture the training data D_{train} as well as possible"

More formally (and since we're being probabilistic):

Find **w** and *b* that assign the largest possible conditional probability to the labels of the items in D_{train}

$$(\mathbf{w}^*, b^*) = \text{argmax}_{(\mathbf{w},b)} \prod_{(\mathbf{x}_i,y_i)\in D_{train}} P(y_i \mid \mathbf{x}_i)$$

⇒ Maximize $P(1 \mid \mathbf{x}_i)$ for any $(\mathbf{x}_i,1)$ with a *positive* label in D_{train}

⇒ Maximize $P(0 \mid \mathbf{x}_i)$ for any $(\mathbf{x}_i,0)$ with a *negative* label in D_{train}

Since $y_i \in \{0,1\}$ we can rewrite this to:

$$(\mathbf{w}^w, b^*) = \text{argmax}_{(\mathbf{w},b)} \prod_{(\mathbf{x}_i,y_i)\in D_{train}} P(1 \mid \mathbf{x}_i)^{y_i} \cdot [1 - P(0 \mid \mathbf{x}_i)]^{1-y_i}$$

For $y_i = 1$, this comes out to: $P(1 \mid \mathbf{x}_i)^1(1 - P(1 \mid \mathbf{x}_i))^0 = P(1 \mid \mathbf{x}_i)$

For $y_i = 0$, this is: $P(1 \mid \mathbf{x}_i)^0(1 - P(1 \mid \mathbf{x}_i))^1 = 1 - P(1 \mid \mathbf{x}_i) = P(0 \mid \mathbf{x}_i)$

Learning = Optimization = Loss Minimization

Learning = parameter estimation = optimization:

Given a particular class of model (logistic regression, Naive Bayes, …) and data D_{train}, find **the *best* parameters** for this class of model on D_{train}

If the model is a probabilistic classifier, think of optimization as Maximum Likelihood Estimation (**MLE**)

*"Best" = return (among all possible parameters for models of this class) parameters that assign the **largest probability** to* D_{train}

In general (incl. for probabilistic classifiers), think of optimization as **Loss Minimization**:

*"Best" = return (among all possible parameters for models of this class) parameters that have the **smallest loss** on* D_{train}

"**Loss**": how bad are the predictions of a model?

*The **loss function** we use to measure loss depends on the class of model*

$L(\hat{y}, y)$: how bad is it to predict $\hat{y}$ if the correct label is y ?

Conditional MLE $\Rightarrow$ Cross-Entropy Loss

Conditional MLE: *Maximize probability* of labels in D_{train}

$$(\mathbf{w}^*, b^*) = \text{argmax}_{(\mathbf{w},b)} \prod_{(\mathbf{x}_i,y_i)\in D_{train}} P(y_i \mid \mathbf{x}_i)$$

$\Rightarrow$ Maximize $P(1 \mid \mathbf{x}_i)$ for any ($\mathbf{x}_i$,1) with a *positive* label in D_{train}

$\Rightarrow$ Maximize $P(0 \mid \mathbf{x}_i)$ for any ($\mathbf{x}_i$,0) with a *negative* label in D_{train}

Equivalently: *Minimize negative log prob.* of labels in D_{train}

$P(y_i \mid \mathbf{x}) = 0 \Leftrightarrow -\log(P(y_i \mid \mathbf{x})) = +\infty$ if y_i is the correct label for $\mathbf{x}$, this is the worst possible model

$P(y_i \mid \mathbf{x}) = 1 \Leftrightarrow -\log(P(y_i \mid \mathbf{x})) = 0$ if y_i is the correct label for $\mathbf{x}$, this is the best possible model

The negative log probability of the correct label is a loss function:

$-\log(P(y_i \mid \mathbf{x}_i))$ is largest ($+\infty$) when we assign all probability to the wrong label,

$-\log(P(y_i \mid \mathbf{x}_i))$ is smallest (0) when we assign all probability to the correct label.

This negative log likelihood loss is also called cross-entropy loss

From loss to per-example cost

Let's define the "cost" of our classifier on the whole dataset as the average loss of each of the *m* training examples:

$$\text{Cost}_{CE}(D_{\text{train}}) = \frac{1}{m} \sum_{i=1..m} -\log P(y_i \mid \mathbf{x}_i)$$

For each example:

$$-\log P(y_i \mid \mathbf{x}_i)$$

$$= -\log(\ P(1 \mid \mathbf{x}_i)^{y_i} \cdot P(0 \mid \mathbf{x}_i)^{1-y_i}\ )$$

[either $y_i = 1$ or $y_i = 0$]

$$= -[\ y_i \log(\ P(1 \mid \mathbf{x}_i)) + (1 - y_i)\log(P(0 \mid \mathbf{x}_i))]$$

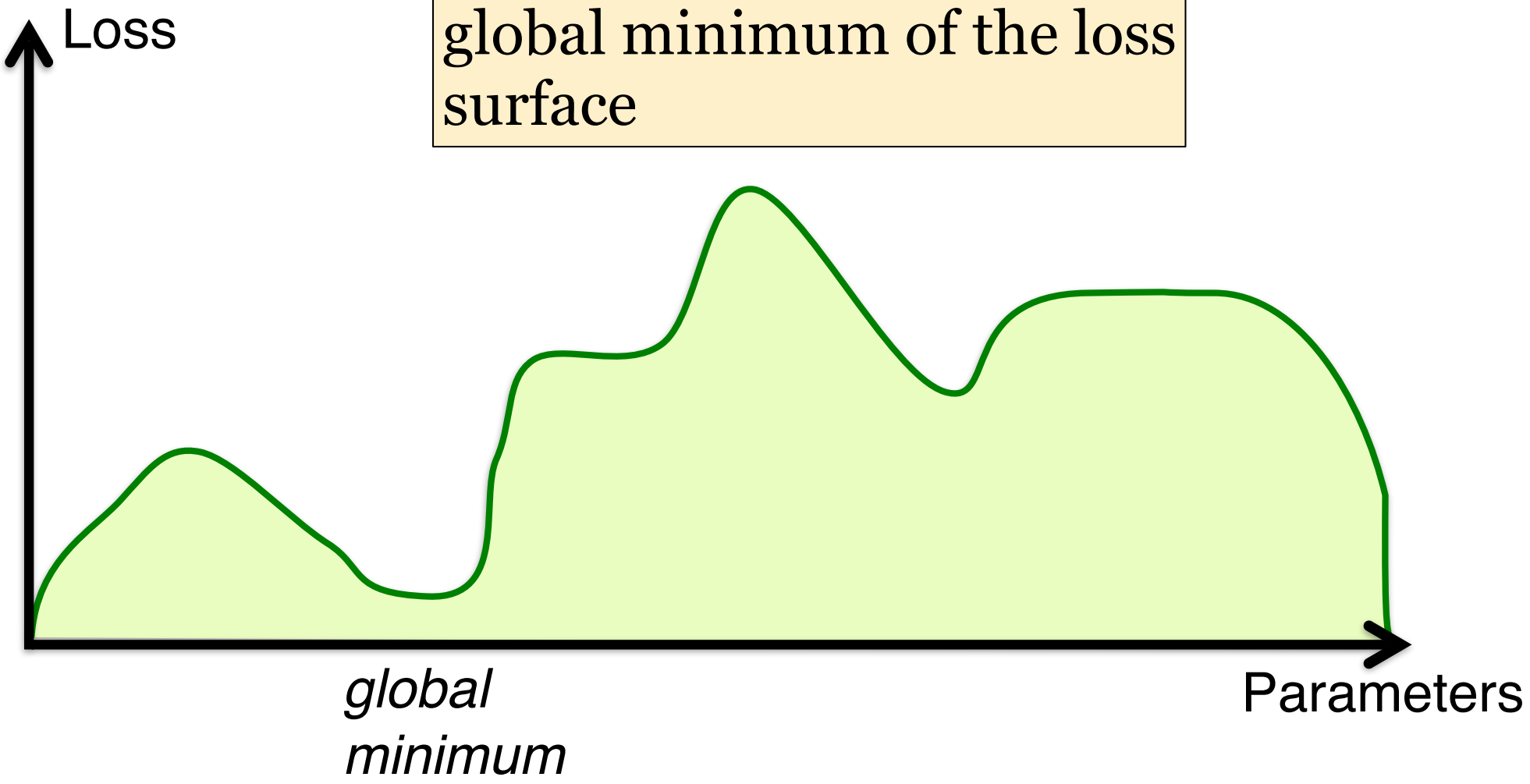
[moving the log inside]

$$= -[\ y_i \log(\sigma(\mathbf{w}\mathbf{x}_i + b) + (1 - y_i)\log(1 - \sigma(\mathbf{w}\mathbf{x}_i + b))]$$

[plugging in definition of $P(1 \mid \mathbf{x}_i)$]

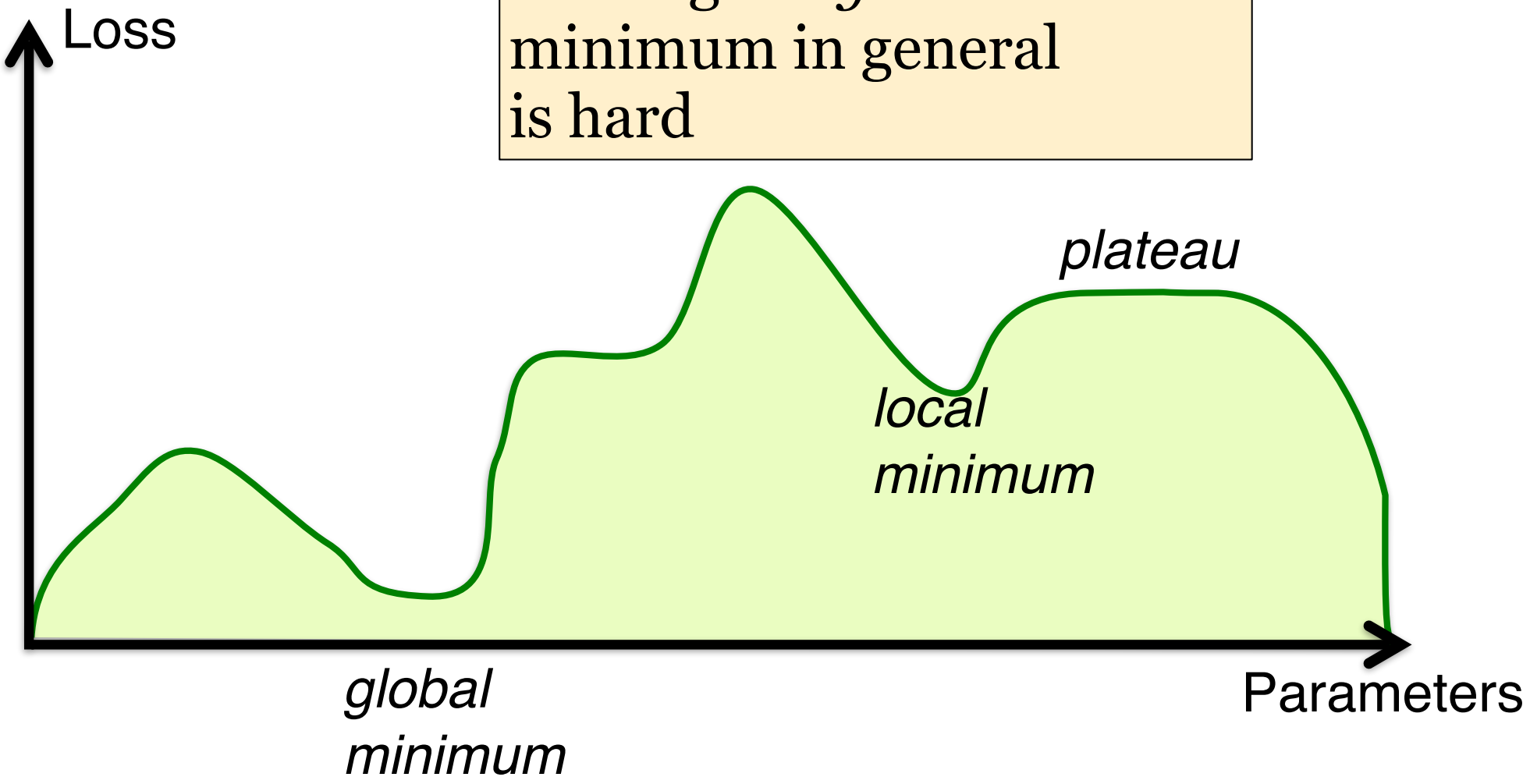
The loss surface

Learning = finding the global minimum of the loss surface



The loss surface

Finding the *global* minimum in general is hard



(Stochastic) Gradient Descent

— We want to find parameters that have minimal cost (loss) on our training data.
— But we don't know the whole loss surface.
— However, the gradient of the cost (loss) of our current parameters tells us how the slope of the loss surface at the point given by our current parameters
— And then we can take a (small) step in the right (downhill) direction (to update our parameters)

Gradient descent:
Compute loss for entire dataset before updating weights

Stochastic gradient descent:
Compute loss for one (randomly sampled) training example before updating weights

Stochastic Gradient Descent

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 $x^{(1)}$, $x^{(2)}$,..., $x^{(n)}$
y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$,..., $y^{(n)}$

$\theta \leftarrow 0$
repeat T times
  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
  Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
  Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$) from the true output $y^{(i)}$?
  $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss ?
  $\theta \leftarrow \theta - \eta\, g$ # go the other way instead
return θ

Gradient for Logistic Regression

Computing the gradient of the loss for example $\mathbf{x}_i$ and weight $\mathbf{w}_j$ is very simple (x_{ji}: j-th feature of $\mathbf{x}_i$)

$$\frac{\delta L(\mathbf{w}, b)}{\delta w_j} = [\sigma(\mathbf{w}\mathbf{x}_i + b) - y_i]x_{ji}$$