Lecture 4:
Introduction to Classification for NLP

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center
Review: Lecture 03

Language models define a probability distribution over all strings \( w = w^{(1)} \ldots w^{(K)} \) in a language:

\[
\sum_{w \in L} P(w) = 1
\]

N-gram language models define the probability of a string \( w = w^{(1)} \ldots w^{(K)} \) as the product of the probabilities of each word \( w^{(i)} \), conditioned on the \( n-1 \) preceding words:

\[
P_{n-gram}(w^{(1)} \ldots w^{(K)}) = \prod_{i=1..K} P(w^{(i)} | w^{(i-1)}, \ldots, w^{(i-n+1)})
\]

Unigram: \( P_{unigram}(w^{(1)} \ldots w^{(K)}) = \prod_{i=1..K} P(w^{(i)}) \)

Bigram: \( P_{bigram}(w^{(1)} \ldots w^{(K)}) = \prod_{i=1..K} P(w^{(i)} | w^{(i-1)}) \)

Trigram: \( P_{trigram}(w^{(1)} \ldots w^{(K)}) = \prod_{i=1..K} P(w^{(i)} | w^{(i-1)}, w^{(i-2)}) \)
How do we…

... estimate the parameters of a language model?
Relative frequency estimation (aka Maximum Likelihood estimation)

... compute the probability of the first n–1 words?
By padding the start of the sentence with n–1 BOS tokens

... obtain one distribution over strings of any length?
By adding an EOS token to the end of each sentence.

... handle unknown words?
By replacing rare words in training and unknown words with an UNK tokens

... evaluate language models?
Intrinsically with perplexity of test data, extrinsically e.g. with word error rate
Overview: Lecture 04

Part 1: Review and Overview
Part 2: What is classification?
Part 3: The Naive Bayes classifier
Part 4: Running & evaluating classification experiments
Part 5: Features for Sentiment analysis

Reading:
Chapter 4, 3rd edition of Jurafsky and Martin
Lecture 04’s questions

What is classification?
   What is binary/multiclass/multilabel classification?

What is supervised learning?
   And why do we want to learn classifiers
   (instead of writing down some rules, say)?

Feature engineering: from data to vectors

How is the Naive Bayes Classifier defined?

How do you evaluate a classifier?
Lecture 04, Part 2: What is Classification?
Spam Detection

Spam detection is a **binary classification** task: Assign one of two labels (e.g. \{SPAM, NOSPAM\}) to the input (here, an email message)
Spam Detection

A **classifier** is a **function** that maps inputs to a predefined (finite) set of class **labels**:

- Spam Detector: Email $\mapsto \{\text{SPAM, NOSPAM}\}$
- Classifier: Input $\mapsto \{\text{LABEL}_1, \ldots, \text{LABEL}_K\}$
The importance of generalization

We need to be able to classify items our classifier has never seen before.
The importance of adaptation

The classifier needs to adapt/change based on the feedback (supervision) it receives.
Text classification more generally

This is a **multiclass** classification task:
Assign **one of K labels** to the input
\{**SPAM**, **CONFERENCES**, **VACATIONS**, …\}
Classification more generally

But: The data we want to classify could be *anything*:

- Emails, words, sentences, images, image regions, sounds, database entries, sets of measurements, ....

We assume that *any* data point can be represented as a vector.
Classification more generally

Before we can use a classifier on our data, we have to map the data to "feature" vectors.
Feature engineering as a prerequisite for classification

To talk about classification mathematically, we assume each input item is represented as a ‘feature’ vector \( x = (x_1, \ldots, x_N) \)

- Each element in \( x \) is one feature.
- The number of elements/features \( N \) is fixed, and may be very large.
- \( x \) has to capture all the information about the item that the classifier needs.

But the raw data points (e.g. documents to classify) are typically not in vector form.

Before we can train a classifier, we therefore have to first define a suitable feature function that maps raw data points to vectors.

In practice, feature engineering (designing suitable feature functions) is very important for accurate classification.
From texts to vectors

In NLP, input items are documents, sentences, words, ….
⇒ How do we represent these items as vectors?

**Bag-of-Words representation:** (this ignores word order)
Assume that each element \( x_i \) in \( (x_1, \ldots, x_N) \) corresponds to one word type \( (v_i) \) in the vocabulary \( V = \{v_1, \ldots, v_N\} \)

There are many different ways to represent a piece of text as a vector over the vocabulary, e.g.:
- If \( x_i \in \{0, 1\} \): Does word \( v_i \) occur (yes: \( x_i = 1 \), no: \( x_i = 0 \)) in the input document?
- If \( x_i \in \{0, 1, 2, \ldots\} \): How often does word \( v_i \) occur in the input document?

[We will see many other ways to map text to vectors this semester]
Now, back to classification....:

A classifier is a function \( f(x) \) that maps input items \( x \in X \) to class labels \( y \in Y \) (

\( X \) is a vector space, \( Y \) is a finite set)

**Binary classification:**
Each input item is mapped to exactly one of 2 classes

**Multi-class classification:**
Each input item is mapped to exactly one of \( K \) classes (\( K > 2 \))

**Multi-label classification:**
Each input item is mapped to \( N \) of \( K \) classes (\( N \geq 1 \), varies per input item)
Classification as supervised machine learning

Classification tasks: Map inputs to a fixed set of class labels

Underlying assumption: Each input really has one (or N) correct labels
Corollary: The correct mapping is a function (aka the ‘target function’)

How do we obtain a classifier (model) for a given task?
— If the target function is very simple (and known), implement it directly
— Otherwise, if we have enough correctly labeled data,
estimate (aka. learn/train) a classifier based on that labeled data.

Supervised machine learning:

Given (correctly) labeled training data, obtain a classifier that predicts these labels as accurately as possible.

Learning is supervised because the learning algorithm can get feedback about how accurate its predictions are from the labels in the training data.
Supervised learning: Training

Labeled Training Data

\[ D^{\text{train}} = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\} \]

Learned model \[ g(x) \]

Give the learning algorithm examples in \( D^{\text{train}} \)

The learning algorithm returns a model \( g(x) \)
Supervised learning: Testing

Reserve some labeled data for testing

Labeled Test Data

$D_{\text{test}}$

$(x'_1, y'_1)$

$(x'_2, y'_2)$

$\ldots$

$(x'_M, y'_M)$
Supervised learning: Testing

Raw Test Data $X^{\text{test}}$
- $x'_1$
- $x'_2$
- ....
- $x'_M$

Labeled Test Data $D^{\text{test}}$
- $(x'_1, y'_1)$
- $(x'_2, y'_2)$
- ...
- $(x'_M, y'_M)$

Test Labels $Y^{\text{test}}$
- $y'_1$
- $y'_2$
- ...

CS47 Natural Language Processing (J. Hockenmaier) https://courses.grainger.illinois.edu/cs447/
Supervised learning: **Testing**

Apply the learned model to the raw test data to obtain **predicted labels** for the test data.

**Raw Test Data**

$X_{\text{test}}$

$x'_{1}$

$x'_{2}$

....

$x'_{M}$

**Learned model** $g(x)$

**Predicted Labels**

$g(X_{\text{test}})$

$g(x'_{1})$

$g(x'_{2})$

....

$g(x'_{M})$

**Test Labels**

$Y_{\text{test}}$

$y'_{1}$

$y'_{2}$

....

$y'_{M}$
Supervised learning: Testing

Evaluate the learned model by comparing the predicted labels against the (correct) test labels.

Raw Test Data:
- $X^{\text{test}}$
- $x'_1$
- $x'_2$
- ....
- $x'_M$

Learned model:
- $g(x)$

Predicted Labels:
- $g(x'_1)$
- $g(x'_2)$
- ....
- $g(x'_M)$

Test Labels:
- $Y^{\text{test}}$
- $y'_1$
- $y'_2$
- ....
- $y'_M$
Supervised machine learning

**The supervised learning task** (for classification):
Given (correctly) labeled data $D = \{(x_i, y_i)\}$, where each item $x_i$ is a vector $(x_1 \ldots x_N)$ with label $y_i$ (which we assume is given by the target function $f(x_i) = y_i$), return a classifier $g(x_i)$ that predicts these labels as accurately as possible (i.e. such that $g(x_i) = y_i = f(x_i)$)

To make this more concrete, we need to specify:
— what *class of functions* $g(x_i)$ to consider
  (many classifiers assume $g(x_i)$ is a linear function)
— what *learning algorithm* we will use to learn $g(x_i)$
  (many learning algorithms assume a particular class of functions)
Classifiers in vector spaces

Binary classification:
Learn a function $f$ that best *separates* the positive and negative examples:

- Assign $y = 1$ to all $x$ where $f(x) > 0$
- Assign $y = 0$ to all $x$ where $f(x) < 0$

**Linear classifier:** $f(x) = wx + b$ is a linear function of $x$
Lecture 04, Part 3:
The Naive Bayes Classifier
Probabilistic classifiers

We want to find the *most likely* class $y$ for the input $x$:

$$y^* = \arg\max_y P(Y = y \mid X = x)$$

$P(Y = y \mid X = x)$:

The probability that the class label is $y$ when the input feature vector is $x$

$$y^* = \arg\max_y f(y)$$

Let $y^*$ be the $y$ that maximizes $f(y)$
Modeling $P(Y | X)$ with Bayes Rule

**Bayes Rule** relates $P(Y | X)$ to $P(X | Y)$ and $P(Y)$:

$$
P(Y | X) = \frac{P(Y, X)}{P(X)} = \frac{P(X | Y)P(Y)}{P(X)} \propto P(X | Y)P(Y)
$$

Bayes rule: The **posterior** $P(Y | X)$ is proportional to the **prior** $P(Y)$ times the **likelihood** $P(X | Y)$.
Using Bayes Rule for our classifier

\[ y^* = \arg \max_y P(Y | X) \]

\[ = \arg \max_y \frac{P(X | Y)P(Y)}{P(X)} \]

[Bayes Rule]

\[ = \arg \max_y P(X | Y)P(Y) \]

[\(P(X)\) doesn’t change \(\arg \max_y\)]
Modeling $P(Y = y)$

$P(Y = y)$ is the “prior” class probability

We can estimate this as the fraction of documents in the training data that have class $y$:

$$\hat{P}(Y = y) = \frac{\# \text{documents } \langle x_i, y_i \rangle \in D_{train} \text{ with } y_i = y}{\# \text{documents } \langle x_i, y_i \rangle \in D_{train}}$$
Modeling $P(X = x \mid Y = y)$

$P(X = x \mid Y = y)$ is the “likelihood” of the input $x$

$x = \langle x_1, \ldots, x_n \rangle$ is a vector

Each $x_i$ represents a word (type) in our vocabulary

Let’s make a (naive) independence assumption:

$P(X = \langle x_1, \ldots, x_n \rangle \mid Y = y) := \prod_{i=1..n} P(X_i = x_i \mid Y = y)$

With this independence assumption, we now need to define (and multiply together) all $P(X_i = x_i \mid Y = y)$
The Naive Bayes Classifier

Assign class $y^*$ to input $x = (x_1 \ldots x_n)$ if

$$y^* = \arg\max_y P(Y = y) \prod_{i=1 \ldots n} P(X_i = x_i | Y = y)$$

$P(Y = y)$ is the **prior class probability** (estimated as the fraction of items in the training data with class $y$)

$P(X_i = x_i | Y = y)$ is the (class-conditional) **likelihood** of the feature $x_i$ conditioned on the class $y$. There are different ways to model this probability.
Capture **whether a word occurs** in a document or not:

\[ P(X_i = x_i \mid Y = y) \]

is a **Bernoulli** distribution \((x_i \in \{0,1\})\)

- \(P(X_i = 1 \mid Y = y)\): probability that word \(v_i\) **occurs** in a document of class \(y\).
- \(P(X_i = 0 \mid Y = y)\): probability that word \(v_i\) **does not occur** in a document of class \(y\).

**Estimation:**

Compute the fraction of documents of class \(y\) with/without \(x_i\):

\[
\hat{P}(X_i = 1 \mid Y = y) = \frac{\text{# docs } \langle x_i, y_i \rangle \in D_{\text{train}} \text{ with } y_i = y \text{ in which } x_i \text{ occurs}}{\text{# docs } \langle x_i, y_i \rangle \in D_{\text{train}} \text{ with } y_i = y}
\]

\[
\hat{P}(X_i = 0 \mid Y = y) = \frac{\text{# docs } \langle x_i, y_i \rangle \in D_{\text{train}} \text{ with } y_i = y \text{ in which } x_i \text{ does not occur}}{\text{# docs } \langle x_i, y_i \rangle \in D_{\text{train}} \text{ with } y_i = y}
\]
What if we want to capture how often a word appears in a document?

Let’s represent each document as a vector of word frequencies \( x_i = C(v_i): \)

**Vocabulary** \( V = \{ \) apple, banana, coffee, drink, eat, fish \( \} \)

**A document:** “fish fish eat eat fish”

**Vector representation** of this document: \( x = \langle 0, 0, 0, 0, 2, 3 \rangle \)

\( P(X_i = x_i \mid Y = y) \): probability that word \( v_i \) occurs with frequency \( x_i = C(v_i) \) in a document of class \( y \).

We can model this by treating \( P(X \mid Y) \) as a **Multinomial distribution**
Multinomial Distribution: Rolling Dice

Before we look at language, let’s assume we’re rolling dice, where the probability of getting any one side (e.g. a 4) when rolling the die once is equal to that of any other side (e.g. a 6).

A multinomial computes the probability of, say, getting two 5s and three 6s if you roll a die five times:

\[
P(\langle 0,0,0,0,2,3 \rangle) = \frac{5!}{0!0!0!0!2!3!} (\frac{1}{6})^2 (\frac{1}{6})^3
\]

#of sequences of three 6s and two 5s: 5!/(0!0!0!0!2!3!)

Prob. of getting a 5 (or a 6) when you roll a die once = 1/6

#Occurrences of 5 and 3: 2 and 3

Prob. of any one sequence of three 6s and two 5s: (1/6)^2 (1/6)^3

NB: Note that we can ignore the probabilities of any sides (i.e. 1, 2, 3, 4) that didn’t come up in our trial (unlike in the Bernoulli model)
\[ P(X_i = x_i \mid Y = y) \text{ as Multinomial} \]

We want to know \( P(X = \langle 0,0,0,0,2,3 \rangle \mid Y = y) \)
where \( \langle 0,0,0,0,2,3 \rangle = \langle C(\text{apple}), \ldots, C(\text{eat}), C(\text{fish}) \rangle \)

Unlike the sides of a dice, words don’t have uniform probability (cf. Zipf’s Law)

So we need to estimate the **class-conditional unigram probability** \( P(\text{apple} \mid Y = y) \) of each word \( v_i \{\text{apple, \ldots, fish}\} \) in documents of class \( y \) …

… and multiply that probability \( x_i \) times
(\( x_i \) = frequency of \( v_i \) in our document):
\[
P(\langle 0,0,0,0,2,3 \rangle \mid Y = y) = P(\text{eat} \mid Y = y)^2 P(\text{fish} \mid Y = y)^3
\]

Or more generally: \( P(X = x \mid Y = y) = \prod P(v_i \mid Y = y)^{x_i} \)
Unigram probabilities $P(v_i \mid Y = y)$

We can estimate the **unigram probability** $P(v_i \mid Y = y)$ of word $v_i$ in all documents of class $y$ as

$$
\hat{P}(v_i \mid Y = y) = \frac{\#v_i \text{ in all docs } \in D_{\text{train}} \text{ of class } y}{\#\text{words in all docs } \in D_{\text{train}} \text{ of class } y}
$$

or **with add-one smoothing** (with $N$ words in vocab $V$):

$$
\hat{P}(v_i \mid Y = y) = \frac{\#v_i \text{ in all docs } \in D_{\text{train}} \text{ of class } y + 1}{\#\text{words in all docs } \in D_{\text{train}} \text{ of class } y + N}
$$
Lecture 04, Part 4: Running and Evaluating Classification Experiments
Evaluating Classifiers

Evaluation setup:
Split data into separate training, (development) 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:** What fraction of items in the test data were classified 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?

Precision and Recall are particularly useful when there are more than two labels.
True vs. false positives, false negatives

- **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.

\[
\text{False Negatives (FN)} = \text{Items labeled X in the gold standard (‘truth’)}
\]

\[
\text{True Positives (TP)} = \text{Items labeled X by the system}
\]

\[
\text{False Positives (FP)} = \text{True Positives (TP)} + \text{False Positives (FP)}
\]
**Precision, Recall, F-Measure**

- **False Negatives (FN)**: Items labeled X in the gold standard (‘truth’)
  \[ \text{FN} = \text{TP} + \text{FN} \]

- **True Positives (TP)**: Items labeled X by the system
  \[ \text{TP} = \text{TP} + \text{FN} \]

- **False Positives (FP)**: Items labeled X by the system
  \[ \text{FP} = \text{TP} + \text{FP} \]

**Precision:**
\[
P = \frac{\text{TP}}{\text{TP} + \text{FP}}
\]

**Recall:**
\[
R = \frac{\text{TP}}{\text{TP} + \text{FN}}
\]

**F-measure:** harmonic mean of precision and recall
\[
F = \frac{2 \cdot P \cdot R}{P + R}
\]
Confusion Matrices

A confusion matrix tabulates how many items that are labeled with class y in the gold data are labeled with class y’ by the classifier.

\[
\begin{array}{ccc}
\text{gold labels} & \text{urgent} & \text{normal} & \text{spam} \\
\text{system output} & 8 & 10 & 1 \\
\text{urgent} & 5 & 60 & 50 \\
\text{normal} & 3 & 30 & 200 \\
\text{spam} & & & \\
\end{array}
\]

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 \(c_2\).

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.

4.8 Test sets and Cross-validation

The training and testing procedure for text classification follows what we saw with language modeling (Section ??): we use the training set to train the model, then use the development test set (also called a devset) to perhaps tune some parameters, and in general decide what the best model is. Once we come up with what we think is the best model, we run it on the (hitherto unseen) test set to report its performance.

While the use of a devset avoids overfitting the test set, having a fixed training set, devset, and test set creates another problem: in order to save lots of data for training, the test set (or devset) might not be large enough to be representative. It would be better if we could somehow use all our data both for training and test. We do this by cross-validation: we randomly choose a training and test set division of our data, train our classifier, and then compute the error rate on the test set. Then we repeat with a different randomly selected training set and test set. We do this sampling process 10 times and average these 10 runs to get an average error rate. This is called 10-fold cross-validation.

The only problem with cross-validation is that because all the data is used for testing, we need the whole corpus to be blind; we can’t examine any of the data to suggest possible features and in general see what’s going on. But looking at the corpus is often important for designing the system. For this reason, it is common...
Confusion Matrices

This can be useful for understanding what kinds of mistakes a (multi-class) classifier makes.

<table>
<thead>
<tr>
<th></th>
<th>urgent</th>
<th>normal</th>
<th>spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>urgent</td>
<td>8</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>normal</td>
<td>5</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>spam</td>
<td>3</td>
<td>30</td>
<td>200</td>
</tr>
</tbody>
</table>

**gold labels**

**system output**

**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

This can be useful for understanding what kinds of mistakes a (multi-class) classifier makes.

![Confusion Matrix](image)

Only 8/16 ‘urgent’ messages are classified correctly.
Confusion Matrices

This can be useful for understanding what kinds of mistakes a (multi-class) classifier makes.

<table>
<thead>
<tr>
<th></th>
<th>gold labels</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>urgent</td>
<td>normal</td>
</tr>
<tr>
<td>urgent</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>normal</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td>spam</td>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

Only 8/16 ‘urgent’ messages are classified correctly. But 200/251 ‘spam’ messages are classified correctly.
Confusion Matrices

This can be useful for understanding what kinds of mistakes a (multi-class) classifier makes.

Only 8/16 ‘urgent’ messages are classified correctly.
But 200/251 ’spam’ messages are classified correctly.
And only 8/19 messages labeled ‘urgent’ are actually urgent.

<table>
<thead>
<tr>
<th></th>
<th>gold labels</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>urgent</td>
<td>normal</td>
<td>spam</td>
</tr>
<tr>
<td>urgent</td>
<td>8</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>normal</td>
<td>5</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>spam</td>
<td>3</td>
<td>30</td>
<td>200</td>
</tr>
</tbody>
</table>

Figure 4.5
Confusion matrix for a three-class categorization task, showing for each pair of classes (c₁, c₂), how many documents from c₁ were (in)correctly assigned to c₂.

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.

4.8 Test sets and Cross-validation

The training and testing procedure for text classification follows what we saw with language modeling (Section ??): we use the training set to train the model, then use the development test set (also called a devset) to perhaps tune some parameters, and in general decide what the best model is. Once we come up with what we think is the best model, we run it on the (hitherto unseen) test set to report its performance.

While the use of a devset avoids overfitting the test set, having a fixed training set, devset, and test set creates another problem: in order to save lots of data for training, the test set (or devset) might not be large enough to be representative. It would be better if we could somehow use all our data both for training and test. We do this by cross-validation: we randomly choose a training and test set division of our data, train our classifier, and then compute the error rate on the test set. Then we repeat with a different randomly selected training set and test set. We do this sampling process 10 times and average these 10 runs to get an average error rate. This is called 10-fold cross-validation.

The only problem with cross-validation is that because all the data is used for testing, we need the whole corpus to be blind; we can’t examine any of the data to suggest possible features and in general see what’s going on. But looking at the corpus is often important for designing the system. For this reason, it is common
Reading off Precision and Recall

![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 \(c_2\)]

<table>
<thead>
<tr>
<th></th>
<th>urgent</th>
<th>normal</th>
<th>spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>urgent</td>
<td>8</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>normal</td>
<td>5</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>spam</td>
<td>3</td>
<td>30</td>
<td>200</td>
</tr>
</tbody>
</table>

**gold labels**

- **precision_u** = \(\frac{8}{8+10+1}\)
- **precision_n** = \(\frac{60}{5+60+50}\)
- **precision_s** = \(\frac{200}{3+30+200}\)

- **recall_u** = \(\frac{8}{8+5+3}\)
- **recall_n** = \(\frac{60}{10+60+30}\)
- **recall_s** = \(\frac{200}{1+50+200}\)

**Figure 4.5**

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

**4.8 Test sets and Cross-validation**

The training and testing procedure for text classification follows what we saw with language modeling (Section ??): we use the training set to train the model, then use the development test set (also called a devset) to perhaps tune some parameters, and in general decide what the best model is. Once we come up with what we think is the best model, we run it on the (hitherto unseen) test set to report its performance.

While the use of a devset avoids overfitting the test set, having a fixed training set, devset, and test set creates another problem: in order to save lots of data for training, the test set (or devset) might not be large enough to be representative. It would be better if we could somehow use all our data both for training and test. We do this by cross-validation: we randomly choose a training and test set division of our data, train our classifier, and then compute the error rate on the test set. Then we repeat with a different randomly selected training set and test set. We do this sampling process 10 times and average these 10 runs to get an average error rate. This is called 10-fold cross-validation.

The only problem with cross-validation is that because all the data is used for testing, we need the whole corpus to be blind; we can't examine any of the data to suggest possible features and in general see what's going on. But looking at the corpus is often important for designing the system. For this reason, it is common
Reading off Precision and Recall

<table>
<thead>
<tr>
<th>Class 1: Urgent</th>
<th>Class 2: Normal</th>
<th>Class 3: Spam</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>system</td>
<td>urgent</td>
<td>not</td>
</tr>
<tr>
<td>urgent</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>not</td>
<td>8</td>
<td>340</td>
</tr>
</tbody>
</table>

precision = \( \frac{8}{8+11} = .42 \)  
precision = \( \frac{60}{60+55} = .52 \)  
precision = \( \frac{200}{200+33} = .86 \)
### Macro-average vs Micro-average

How do we aggregate precision and recall across classes?

<table>
<thead>
<tr>
<th>Class 1: Urgent</th>
<th>Class 2: Normal</th>
<th>Class 3: Spam</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>true</strong>&lt;br&gt;system urgent</td>
<td><strong>true</strong>&lt;br&gt;system normal</td>
<td><strong>true</strong>&lt;br&gt;system spam</td>
</tr>
<tr>
<td>urgent</td>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>not</td>
<td>11</td>
<td>55</td>
</tr>
<tr>
<td><strong>true</strong>&lt;br&gt;system not</td>
<td><strong>true</strong>&lt;br&gt;system not</td>
<td><strong>true</strong>&lt;br&gt;system not</td>
</tr>
<tr>
<td>urgent</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>not</td>
<td>340</td>
<td>212</td>
</tr>
</tbody>
</table>

- **Precision** for each class:
  - Class 1: \( \frac{8}{8+11} = 0.42 \)
  - Class 2: \( \frac{60}{60+55} = 0.52 \)
  - Class 3: \( \frac{200}{200+33} = 0.86 \)

- **Macroaverage precision**: \( \frac{0.42 + 0.52 + 0.86}{3} = 0.60 \)

---

**Macro-average**: average the precision over all \( K \) classes (regardless of how common each class is).
Macro-average vs Micro-average

How do we aggregate precision and recall across classes?

Micro-average: average the precision over all N items (regardless of what class they have)
Macro-average vs. Micro-average

Which average should you report?

**Macro-average** (average P/R of all classes):
Useful if performance on all *classes* is equally important.

**Micro-average** (average P/R of all items):
Useful if performance on all *items* is equally important.
The End
Lecture 04, Part 5: Features for Sentiment Analysis