REVA UNIVERSITY
Bengaluru, India

nirf

Natural Language Processing

Unit 4
Classifying Text:

Agenda

- Supervised Classification
- Examples for Supervised Classification
- Evaluation
- Decision Trees
- Naive Bayes Classifiers
- Maximum Entropy Classifiers
- Modelling Linguistic Patterns

Introduction

- The goal of this chapter is to answer the following questions:

1. How can we identify particular features of language data that are salient for classifying it?

2. How can we construct models of language that can be used to perform language processing tasks automatically?

3. What can we learn about language from these models?

- We will study some important machine learning techniques like-
  -Decision trees,
  -Naive Bayes classifiers
  -Maximum entropy classifiers.

Supervised Classification:

- Classification is the task of choosing the correct class label for a given input.
- In basic classification tasks, each input is considered in isolation from all other inputs, and the set of labels is defined in advance.

- Some examples of classification tasks are:

  - Deciding whether an email is spam or not.
  - Deciding what the topic of a news article is, from a fixed list of topic areas such as "sports," "technology," and "politics."
  - Deciding whether a given occurrence of the word *bank* is used to refer to a river bank, a financial institution

- The basic classification task has a number of interesting variants.
- For example-
  In multiclass classification, each instance may be assigned multiple labels;
  In open-class classification, the set of labels is not defined in advance;
  and in sequence classification, a list of inputs are jointly classified.

- A classifier is called supervised if it is built based on training corpora containing the correct label for each input.

  The framework used by supervised classification is shown in Figure 6-1.

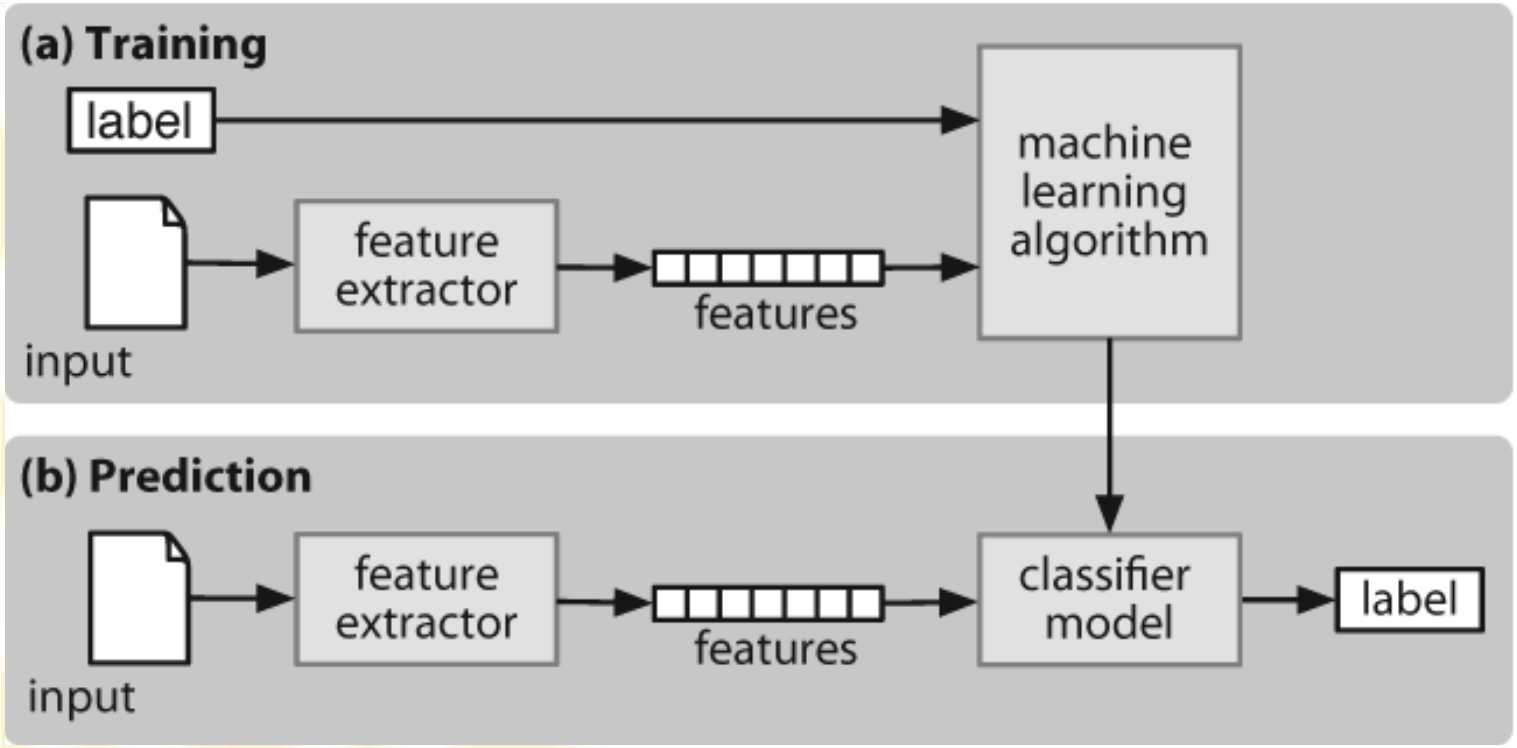

Figure 6-1. Supervised classification.

(a) During training, a feature extractor is used to convert each input value to a feature set. These feature sets, which capture the basic information about each input that should be used to classify it. Pairs of feature sets and labels are fed into the machine learning algorithm to generate a model.

(b) During prediction, the same feature extractor is used to convert unseen inputs to feature sets. These feature sets are then fed into the model, which generates predicted labels.

Gender Identification

- In Section 2.4, we saw that male and female names have some distinctive characteristics.
- Names ending in *a*, *e*, y and *i* are likely to be female.
- While names ending in *k*, *o*, *r*, *s*, and *t* are likely to be male.

Let's build a classifier to model these differences more precisely.

- **The first step in creating a classifier is deciding what features of the input are relevant, and how to encode those features.**
  For this example:
  we'll start by just looking at the final letter of a given name.
  The following feature extractor function builds a dictionary containing relevant information about a given name:

```
>>> def gender_features(word):
... return {'last_letter': word[-1]}
>>> gender_features('Shrek')
{'last_letter': 'k'}
```

- The dictionary that is returned by this function is called a **feature set** and maps from features' names to their values.

Now that we've defined a feature extractor, we need to prepare a list of examples and corresponding class labels:

```
>>> from nltk.corpus import names
>>> import random
>>> names = ([(name, 'male') for name in names.words('male.txt')] +
... [(name, 'female') for name in names.words('female.txt')])
>>> random.shuffle(names)
```

Next, we use the feature extractor to process the names data, and divide the resulting list of feature sets into a training set and a test set. The training set is used to train a new "naive Bayes" classifier.

```
>>> featuresets = [(gender_features(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

- For now, let's just test it out on some names that did not appear in its training data:

```
>>> classifier.classify(gender_features('Neo'))
'male'
>>> classifier.classify(gender_features('Trinity'))
'female'
```

- Observe that these character names from *The Matrix* are correctly classified.

- We can systematically evaluate the classifier on a much larger quantity of unseen data:

```
>>> print nltk.classify.accuracy(classifier, test_set)
0.758
```

- Finally, we can examine the classifier to determine which features it found most effective for distinguishing the names' genders:

```
>>> classifier.show_most_informative_features(5)
Most Informative Features
        last_letter = 'a' female : male = 38.3 : 1.0
        last_letter = 'k' male : female = 31.4 : 1.0
        last_letter = 'f' male : female = 15.3 : 1.0
        last_letter = 'p' male : female = 10.6 : 1.0
        last_letter = 'w' male : female = 10.6 : 1.0
```

Choosing the Right Features

- Much of the interesting work in building a classifier is deciding what features might be relevant, and how we can represent them.
- Typically, feature extractors are built through a process of trial-and-error, guided by intuitions about what information is relevant to the problem.
- We take this approach for name gender features in Example 6-1.

Example 6-1. A feature extractor that overfits gender features. The featuresets returned by this feature extractor contain a large number of specific features, leading to overfitting for the relatively small Names Corpus.

```
def gender_features2(name):
    features = {}
    features["firstletter"] = name[0].lower()
    features["lastletter"] = name[–1].lower()
    for letter in 'abcdefghijklmnopqrstuvwxyz':
        features["count(%s)" % letter] = name.lower().count(letter)
        features["has(%s)" % letter] = (letter in name.lower())
    return features
>>> gender_features2('John')
{'count(j)': 1, 'has(d)': False, 'count(b)': 0, ...}
```

- However, there are usually limits to the number of features that you should use with a given learning algorithm—if you provide too many features, then the algorithm will have a higher chance of relying on idiosyncrasies of your training data that don't generalize well to new examples. This problem is known as **overfitting**, and can be especially problematic when working with small training sets.

- For example, if we train a naive Bayes classifier using the feature extractor shown in Example 6-1, it will overfit the relatively small training set, resulting in a system whose accuracy is about 1% lower than the accuracy of a classifier that only pays attention to the final letter of each name:

```
>>> featuresets = [(gender_features2(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
0.748
```

- Once an initial set of features has been chosen, a very productive method for refining the feature set is **error analysis**.
- First, we select a **development set**, containing the corpus data for creating the model. This development set is then subdivided into the **training set** and the **dev-test** set.

```
>>> train_names = names[1500:]
>>> devtest_names = names[500:1500]
>>> test_names = names[:500]
```

- The training set is used to train the model, and the dev-test set is used to perform error analysis.
- The test set serves in our final evaluation of the system. For reasons discussed later, it is important that we employ a separate dev-test set for error analysis, rather than just using the test set.
- The division of the corpus data into different subsets is shown in Figure 6-2.

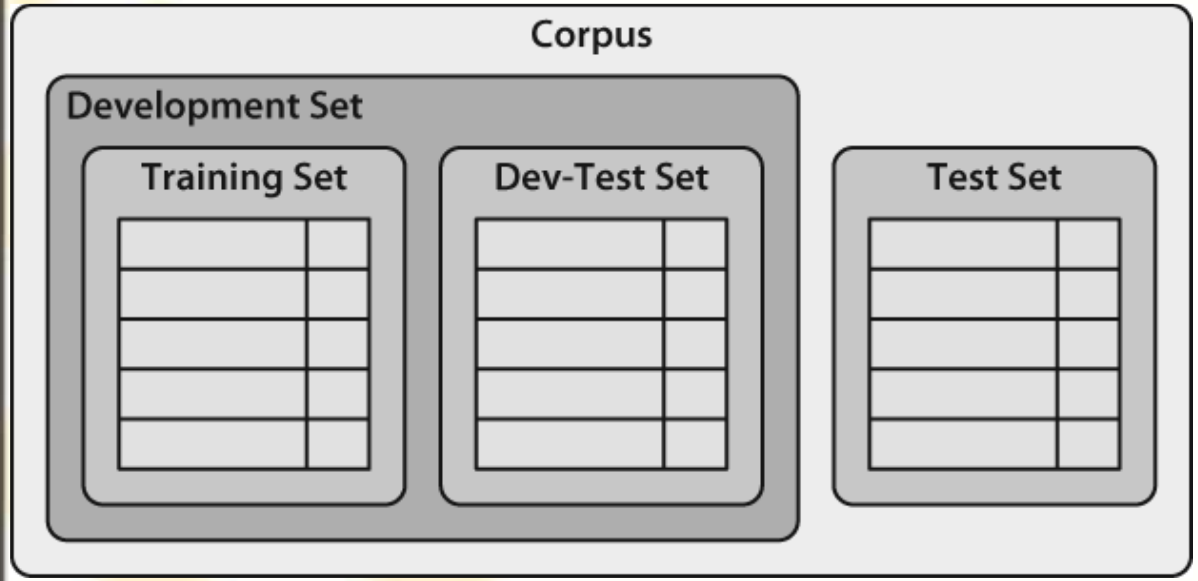

Figure 6-2. Organization of corpus data for training supervised classifiers. The corpus data is divided into two sets: the development set and the test set. The development set is often further subdivided into a training set and a dev-test set

- Having divided the corpus into appropriate datasets, we train a model using the training set , and then run it on the dev-test set .

```
>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> devtest_set = [(gender_features(n), g) for (n,g) in devtest_names]
>>> test_set = [(gender_features(n), g) for (n,g) in test_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, devtest_set)
0.765
```

- Using the dev-test set, we can generate a list of the errors that the classifier makes when predicting name genders:

```
>>> errors = []
>>> for (name, tag) in devtest_names:
... guess = classifier.classify(gender_features(name))
... if guess != tag:
... errors.append( (tag, guess, name) )
```

Document Classification

- In Section 2.1, we saw several examples of corpora where documents have been labelled with categories.

- Using these corpora, we can build classifiers that will automatically tag new documents with appropriate category labels. First, we construct a list of documents, labeled with the appropriate categories.

- For this example, we've chosen the Movie Reviews Corpus, which categorizes each review as positive or negative.

```
>>> from nltk.corpus import movie_reviews
>>> documents = [(list(movie_reviews.words(fileid)), category)
... for category in movie_reviews.categories()
... for fileid in movie_reviews.fileids(category)]
>>> random.shuffle(documents)
```

- Next, we define a feature extractor for documents, so the classifier will know which aspects of the data it should pay attention to (see Example 6-2).

- For document topic identification, we can define a feature for each word, indicating whether the document contains that word.

- To limit the number of features that the classifier needs to process, we begin by constructing a list of the 2,000 most frequent words in the overall corpus . We can then define a feature extractor that simply checks whether each of these words is present in a given document.

- Example 6-2. A feature extractor for document classification, whose features indicate whether or not individual words are present in a given document.

```
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
word_features = all_words.keys()[:2000]
        def document_features(document):
        document_words = set(document)
        features = {}
        for word in word_features:
                features['contains(%s)' % word] = (word in document_words)
        return features
>>> print document_features(movie_reviews.words('pos/cv957_8737.txt'))
{'contains(waste)': False, 'contains(lot)': False, ...}
```

- Now that we've defined our feature extractor, we can use it to train a classifier to label new movie reviews (Example 6-3).

- To check how reliable the resulting classifier is, we compute its accuracy on the test set . And once again, we can use show_most_infor mative_features() to find out which features the classifier found to be most informative .

Example 6-3. Training and testing a classifier for document classification.

```
featuresets = [(document_features(d), c) for (d,c) in documents]
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
0.81
>>> classifier.show_most_informative_features(5)
Most Informative Features
contains(outstanding) = True pos : neg = 11.1 : 1.0
contains(seagal) = True neg : pos = 7.7 : 1.0
contains(wonderfully) = True pos : neg = 6.8 : 1.0
contains(damon) = True pos : neg = 5.9 : 1.0
contains(wasted) = True neg : pos = 5.8 : 1.0
```

Apparently in this corpus, a review that mentions *Seagal* is almost 8 times more likely to be negative than positive, while a review that mentions *Damon* is about 6 times more likely to be positive

Part-of-Speech Tagging

In Chapter 5, we built a regular expression tagger that chooses a part-of-speech tag for a word by looking at the internal makeup of the word.

However, this regular expression tagger had to be handcrafted. Instead, we can train a classifier to work out which suffixes are most informative. Let's begin by finding the most common suffixes:

```
>>> from nltk.corpus import brown
>>> suffix_fdist = nltk.FreqDist()
>>> for word in brown.words():
... word = word.lower()
... suffix_fdist.inc(word[-1:])
... suffix_fdist.inc(word[-2:])
... suffix_fdist.inc(word[-3:])
>>> common_suffixes = suffix_fdist.keys()[:100]
>>> print common_suffixes
['e', ',', '.', 's', 'd', 't', 'he', 'n', 'a', 'of', 'the',
'y', 'r', 'to', 'in', 'f', 'o', 'ed', 'nd', 'is', 'on', 'l',
'g', 'and', 'ng', 'er', 'as', 'ing', 'h', 'at', 'es', 'or',
're', 'it', '``', 'an', '"', 'm', ';', 'i', 'ly', 'ion', ...]
```

Next, we'll define a feature extractor function that checks a given word for these suffixes:

```
>>> def pos_features(word):
... features = {}
... for suffix in common_suffixes:
... features['endswith(%s)' % suffix] = word.lower().endswith(suffix)
... return features
```

- Feature extraction functions behave like tinted glasses, highlighting some of the properties (colors) in our data and making it impossible to see other properties.
- The classifier will rely exclusively on these highlighted properties when determining how to label inputs.
- In this case, the classifier will make its decisions based only on information about which of the common suffixes (if any) a given word has.

Now that we've defined our feature extractor, we can use it to train a new "decision tree" classifier (to be discussed in Section 6.4):

```
>>> tagged_words = brown.tagged_words(categories='news')
>>> featuresets = [(pos_features(n), g) for (n,g) in tagged_words]
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.DecisionTreeClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
0.627051218299353 51
>>> classifier.classify(pos_features('cats'))
'NNS
```

Evaluation

- In order to decide whether a classification model is accurately capturing a pattern, we must evaluate that model.
- The result of this evaluation is important for deciding how trustworthy the model is, and for what purposes we can use it.
- Evaluation can also be an effective tool for guiding us in making future improvements to the model.

The Test Set

- Most evaluation techniques calculate a score for a model by comparing the labels that it generates for the inputs in a **test set** (or **evaluation set**) with the correct labels for those inputs.

- When building the test set, there is often a trade-off between the amount of data available for testing and the amount available for training.

- For classification tasks that have a small number of well-balanced labels and a diverse test set, a meaningful evaluation can be performed with as few as 100 evaluation instances.

- But if a classification task has a large number of labels or includes very infrequent labels, then the size of the test set should be chosen to ensure that the least frequent label occurs at least 50 times.

- Additionally, if the test set contains many closely related instances—such as instances drawn from a single document—then the size of the test set should be increased to ensure that this lack of diversity does not skew the evaluation results.

- Another consideration when choosing the test set is the degree of similarity between instances in the test set and those in the development set.

- The more similar these two datasets are, the less confident we can be that evaluation results will generalize to other datasets.

- For example, consider the part-of-speech tagging task. At one extreme, we could create the training set and test set by randomly assigning sentences from a data source that reflects a single genre, such as news:

```
>>> import random
>>> from nltk.corpus import brown
>>> tagged_sents = list(brown.tagged_sents(categories='news'))
>>> random.shuffle(tagged_sents)
>>> size = int(len(tagged_sents) * 0.1)
>>> train_set, test_set = tagged_sents[size:], tagged_sents[:size]
```

- In this case, our test set will be *very* similar to our training set.

Accuracy

- The simplest metric that can be used to evaluate a classifier, **accuracy**, measures the percentage of inputs in the test set that the classifier correctly labeled.
- For example, a name gender classifier that predicts the correct name 60 times in a test set containing 80 names would have an accuracy of 60/80 = 75%.
- The function nltk.classify.accuracy() will calculate the accuracy of a classifier model on a given test set:

```
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print 'Accuracy: %4.2f' % nltk.classify.accuracy(classifier, test_set)
0.75
```

- When interpreting the accuracy score of a classifier, it is important to consider the frequencies of the individual class labels in the test set.
- For example, consider a classifier that determines the correct word sense for each occurrence of the word bank.
- If we evaluate this classifier on financial newswire text, then we may find that the financial institution sense appears 19 times out of 20. In that case, an accuracy of 95% would hardly be impressive, since we could achieve that accuracy with a model that always returns the financial-institution sense.
- However, if we instead evaluate the classifier on a more balanced corpus, where the most frequent word sense has a frequency of 40%, then a 95% accuracy score would be a much

Precision and Recall

- Another instance where accuracy scores can be misleading is in “search” tasks, such as
  information retrieval, where we are attempting to find documents that are relevant to a particular task.

- Since the number of irrelevant documents far outweighs the number of relevant documents, the accuracy score for a model that labels every document as irrelevant would be very close to 100%.

- It is therefore conventional to employ a different set of measures for search tasks, based
  on the number of items in each of the four categories shown in Figure 6-3:

- **True positives** are relevant items that we correctly identified as relevant.
- **True negatives** are irrelevant items that we correctly identified as irrelevant.
- **False positives** (or **Type I errors**) are irrelevant items that we incorrectly identified as relevant.
- **False negatives** (or **Type II errors**) are relevant items that we incorrectly identified as irrelevant

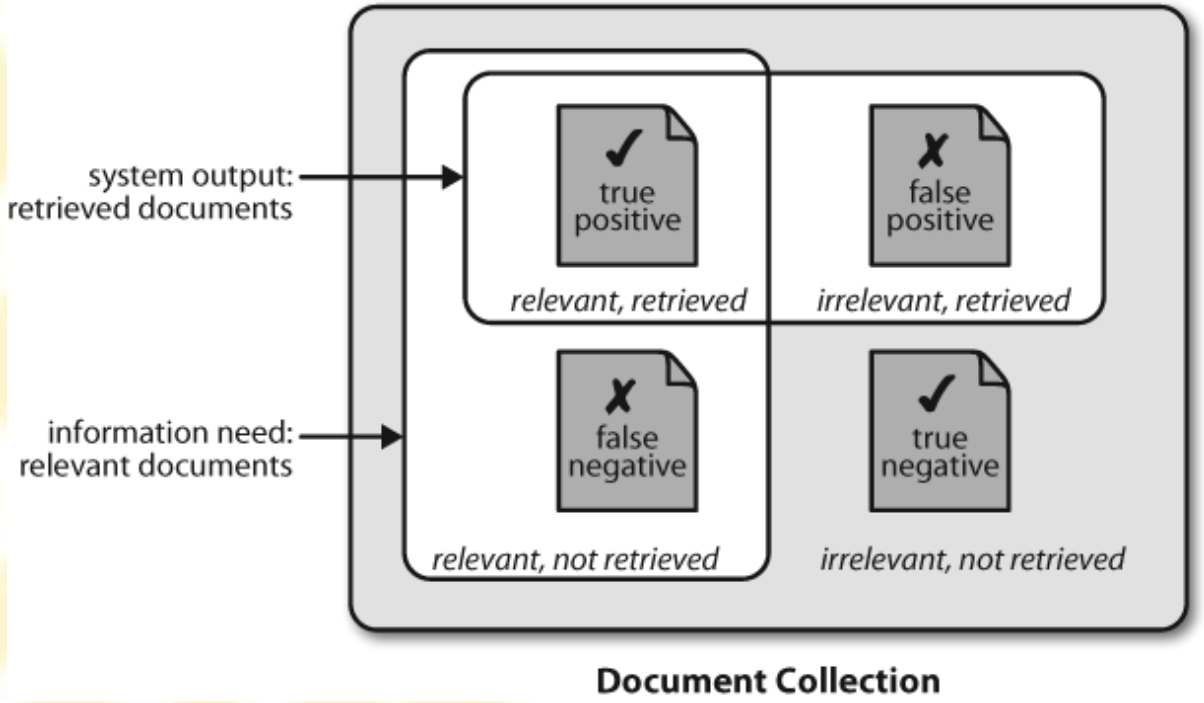

Figure 6-3. True and false positives and negatives

Given these four numbers, we can define the following metrics:

• **Precision**, which indicates how many of the items that we identified were relevant,
is *TP*/(*TP*+*FP*).

• **Recall**, which indicates how many of the relevant items that we identified, is *TP*/(*TP*+*FN*).

• The **F-Measure** (or **F-Score**), which combines the precision and recall to give a single score, is defined to be the harmonic mean of the precision and recall (2 × *Precision* × *Recall*)/(*Precision*+*Recall*).

Confusion Matrices

When performing classification tasks with three or more labels, it can be informative to subdivide the errors made by the model based on which types of mistake it made.

A **confusion matrix** is a table where each cell [*i*,*j*] indicates how often label *j* was predicted
when the correct label was *i*.

Thus, the diagonal entries (i.e., cells [*i*,*j*]) indicate labels that were correctly predicted, and the off-diagonal entries indicate errors.

In the following example, we generate a confusion matrix for the unigram tagger developed in Section 5.4:

```
>>> def tag_list(tagged_sents):
... return [tag for sent in tagged_sents for (word, tag) in sent]
>>> def apply_tagger(tagger, corpus):
... return [tagger.tag(nltk.tag.untag(sent)) for sent in corpus]
>>> gold = tag_list(brown.tagged_sents(categories='editorial'))
>>> test = tag_list(apply_tagger(t2,
brown.tagged_sents(categories='editorial')))
>>> cm = nltk.ConfusionMatrix(gold, test)
```

| N |
| N I A J N V N |
| N N T J . S , B P |
----+--+
NN | <11.8%> 0.0% . 0.2% . 0.0% . 0.3% 0.0% |
IN | 0.0% <9.0%> . . . 0.0% . . . |
AT | . . <8.6%> . . . . . |
JJ | 1.6% . . <4.0%> . . . 0.0% 0.0% |
. | . . . . <4.8%> . . . . |
NNS | 1.5% . . . . <3.2%> . . 0.0% |
, | . . . . . <4.4%> . . |
B | 0.9% . . 0.0% . . . <2.4%> . |
NP | 1.0% . . 0.0% . . . . <1.9%>|
----+--+
(row = reference; col = test)

The confusion matrix indicates that common errors include a substitution of NN for JJ (for 1.6% of words), and of NN for NNS (for 1.5% of words). Note that periods (.)
indicate cells whose value is 0, and that the diagonal entries—which correspond to
correct classifications—are marked with angle brackets.

Cross-Validation

- In order to evaluate our models, we must reserve a portion of the annotated data for the test set.
- If the test set is too small, our evaluation may not be accurate. However, making the test set larger usually means making the training set smaller, which can have a significant impact on performance if a limited amount of annotated data is available.

- One solution to this problem is to perform multiple evaluations on different test sets, then to combine the scores from those evaluations, a technique known as **crossvalidation**.

- In particular, we subdivide the original corpus into *N* subsets called **folds**. For each of these folds, we train a model using all of the data *except* the data in that fold, and then test that model on the fold. Even though the individual folds might be too small to give accurate evaluation scores on their own, the combined evaluation score is based on a large amount of data and is therefore quite reliable.

- A second, and equally important, advantage of using cross-validation is that it allows us to examine how widely the performance varies across different training sets. If we get very similar scores for all *N* training sets, then we can be fairly confident that the score is accurate. On the other hand, if scores vary widely across the *N* training sets, then we should probably be skeptical about the accuracy of the evaluation score.

Decision Trees

A **decision tree** is a simple flowchart that selects labels for input values. This flowchart consists of **decision nodes**, which check feature values, and **leaf nodes**, which assign labels.

To choose the label for an input value, we begin at the flowchart's initial decision node, known as its **root node**. This node contains a condition that checks one of the input value's features, and selects a branch based on that feature's value.

Following the branch that describes our input value, we arrive at a new decision node, with a new condition on the input value's features.

We continue following the branch selected by each node's condition, until we arrive at a leaf node which provides a label for the input value.

Figure 6-4 shows an example decision tree model for the name gender task.

Once we have a decision tree, it is straightforward to use it to assign labels to new input values. What's less straightforward is how we can build a decision tree that models a given training set. But before we look at the learning algorithm for building decision trees, we'll consider a simpler task: picking the best "decision stump" for a corpus.

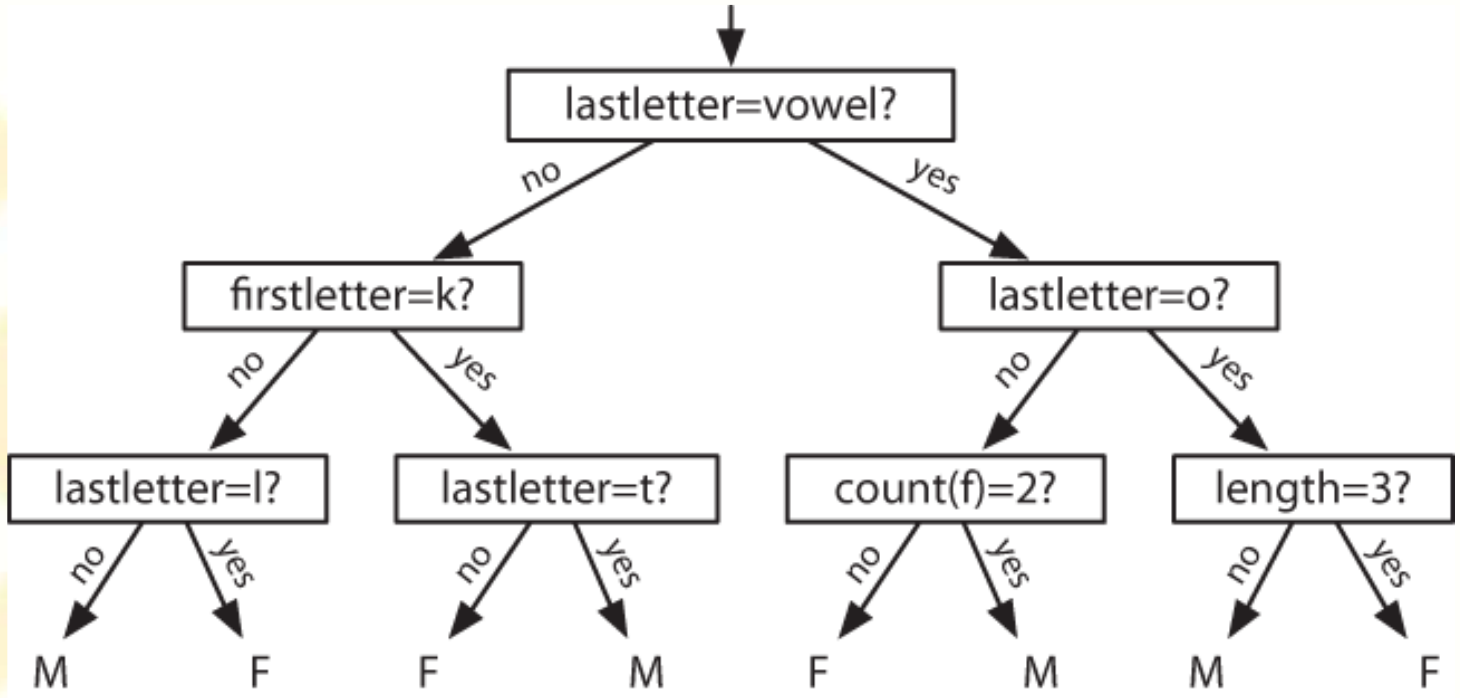

Figure 6-4. Decision Tree model for the name gender task.

Entropy and Information Gain

There are several methods for identifying the most informative feature for a decision stump. One popular alternative, called **information gain**, measures how much more organized the input values become when we divide them up using a given feature.

To measure how disorganized the original set of input values are, we calculate entropy of their labels, which will be high if the input values have highly varied labels, and low if many input values all have the same label.

In particular, entropy is defined as the sum of the probability of each label times the log probability of that same label:

(1) $H = \Sigma_{l \in \text{labels}} P(l) \times \log_2 P(l)$.

For example, Figure 6-5 shows how the entropy of labels in the name gender prediction task depends on the ratio of male to female names. Note that if most input values have the same label (e.g., if P(male) is near 0 or near 1), then entropy is low.

In particular, labels that have low frequency do not contribute much to the entropy (since $P(l)$ is small), and labels with high frequency also do not contribute much to the entropy (since $\log_2 P(l)$ is small).

On the other hand, if the input values have a wide variety of labels, then there are many labels with a "medium" frequency, where neither $P(l)$ nor $\log_2 P(l)$ is small, so the entropy is high. Example 6-8 demonstrates how to calculate the entropy of a list of labels.

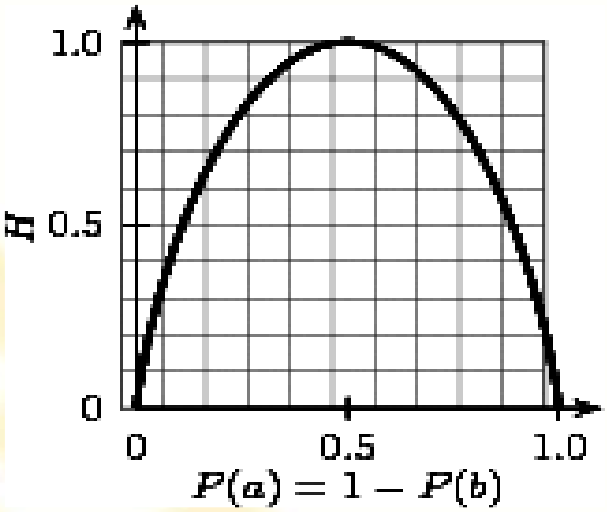

Figure 6-5. The entropy of labels in the name gender prediction task, as a function of the percentage of names in a given set that are male.

Example 6-8. Calculating the entropy of a list of labels.

```
import math
def entropy(labels):
freqdist = nltk.FreqDist(labels)
probs = [freqdist.freq(l) for l in nltk.FreqDist(labels)]
return -sum([p * math.log(p,2) for p in probs])
>>> print entropy(['male', 'male', 'male', 'male'])
0.0
>>> print entropy(['male', 'female', 'male', 'male'])
0.811278124459
>>> print entropy(['female', 'male', 'female', 'male'])
1.0
>>> print entropy(['female', 'female', 'male', 'female'])
0.811278124459
>>> print entropy(['female', 'female', 'female', 'female'])
0.0
```

Decision trees have a number of useful qualities. To begin with, they're simple to understand, and easy to interpret. This is especially true near the top of the decision tree, where it is usually possible for the learning algorithm to find very useful features. Decision trees are especially well suited to cases where many hierarchical categorical distinctions can be made.
For example, decision trees can be very effective at capturing phylogeny trees. However, decision trees also have a few disadvantages. One problem is that, since each branch in the decision tree splits the training data, the amount of training data available to train nodes lower in the tree can become quite small.

As a result, these lower decision nodes may **overfit** the training set, learning patterns that reflect idiosyncrasies of the training set rather than linguistically significant patterns in the underlying problem. One solution to this problem is to stop dividing nodes once the amount of training data becomes too small. Another solution is to grow a full decision tree, but then to **prune** decision nodes that do not improve performance on a dev-test.

A second problem with decision trees is that they force features to be checked in a specific order, even when features may act relatively independently of one another. For example, when classifying documents into topics (such as sports, automotive, or murder mystery), features such as hasword(football) are highly indicative of a specific label, regardless of what the other feature values are. Since there is limited space near the top of the decision tree, most of these features will need to be repeated on many different

Naive Bayes Classifiers

In **naive Bayes** classifiers, every feature gets a say in determining which label should be assigned to a given input value. To choose a label for an input value, the naive Bayes classifier begins by calculating the **prior probability** of each label, which is determined by checking the frequency of each label in the training set. The contribution from each feature is then combined with this prior probability, to arrive at a likelihood estimate for each label. The label whose likelihood estimate is the highest is then assigned to the input value. Figure 6-6 illustrates this process.

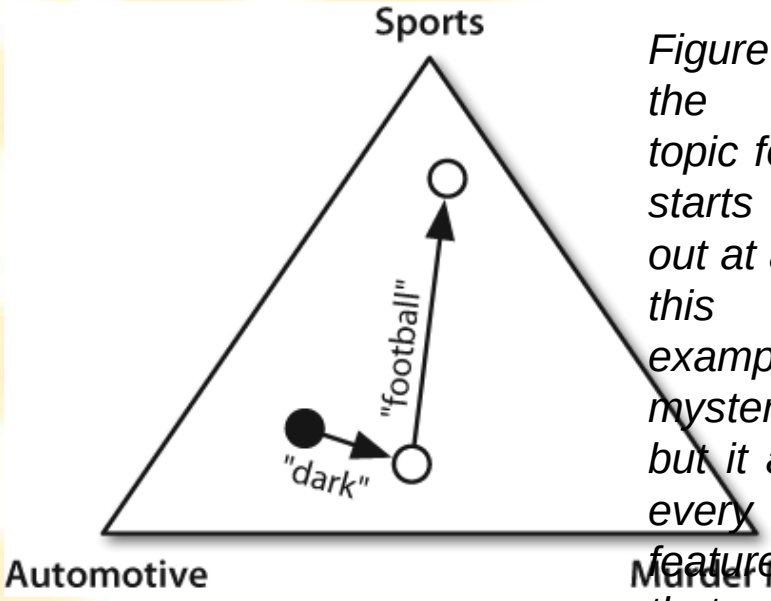

Figure 6-6. An abstract illustration of the procedure used by the naive Bayes classifier to cho
the
topic for a document. In the training corpus, most documents are automotive, so the class
starts
out at a point closer to the "automotive" label. But it then considers the effect of each feature
this
example, the input document contains the word dark*, which is a weak indicator for mur*
mysteries,
but it also contains the word football*, which is a strong indicator for sports documents. A*
every
feature has made its contribution, the classifier checks which label it is closest to, and assi
that
label to the input.

Individual features make their contribution to the overall decision by “voting against”
labels that don’t occur with that feature very often. In particular, the likelihood score
for each label is reduced by multiplying it by the probability that an input value with
that label would have the feature.

For example, if the word *run* occurs in 12% of the sports documents, 10% of the murder mystery documents, and 2% of the automotive documents, then the likelihood score for the sports label will be multiplied by 0.12, the likelihood score for the murder mystery label will be multiplied by 0.1, and the likelihood score for the automotive label will be multiplied by 0.02.

The overall effect will be to reduce the score of the murder mystery label slightly more than the score of the sports label, and to significantly reduce the automotive label with respect to the other two labels. This process is illustrated

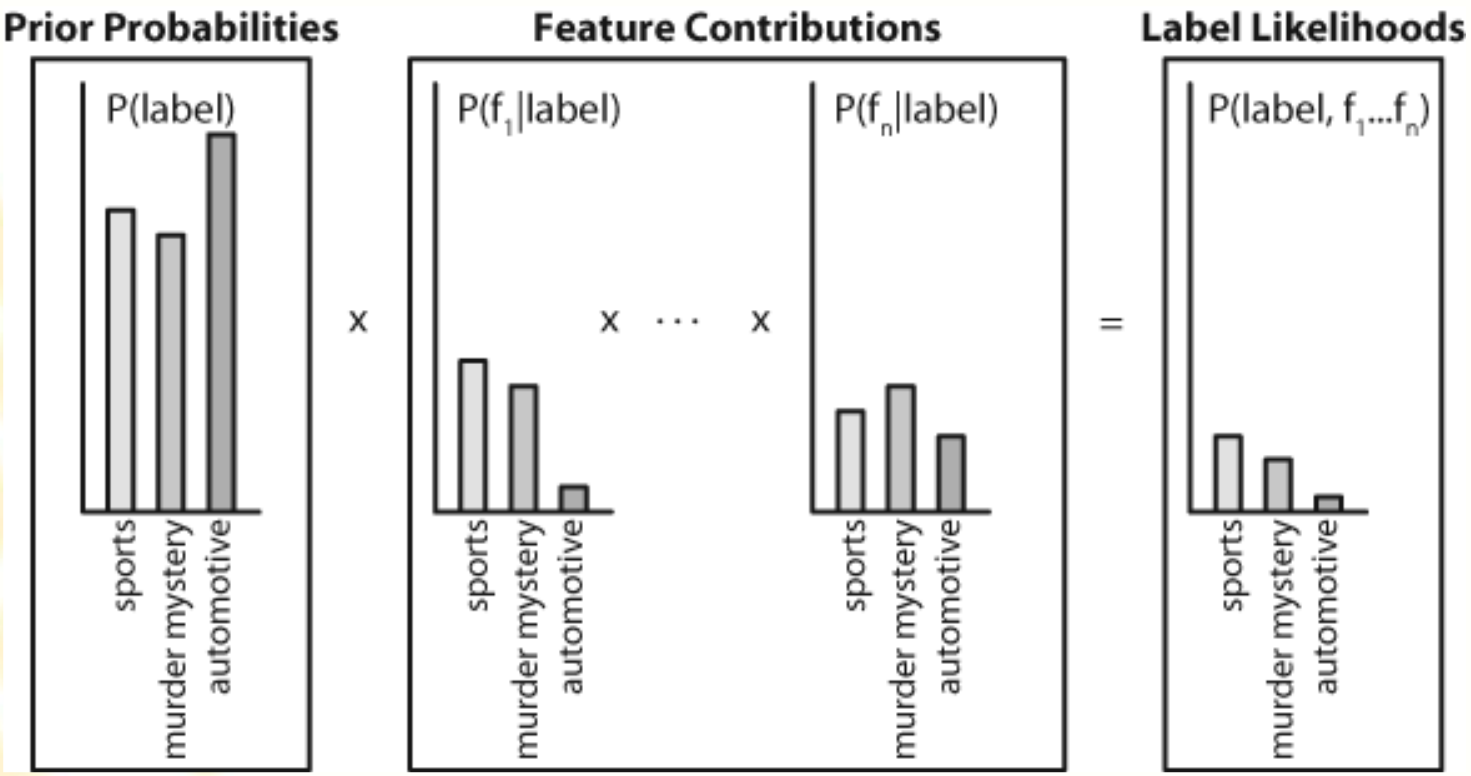

Figure 6-7. Calculating label likelihoods with naive Bayes. Naive Bayes begins by calculating the prior probability of each label, based on how frequently each label occurs in the training data. Every feature then contributes to the likelihood estimate for each label, by multiplying it by the probability that input values with that label will have that feature. The resulting likelihood score can be thought of as an estimate of the probability that a randomly selected value from the training set would have both the given label and the set of features, assuming that the feature probabilities are all independent

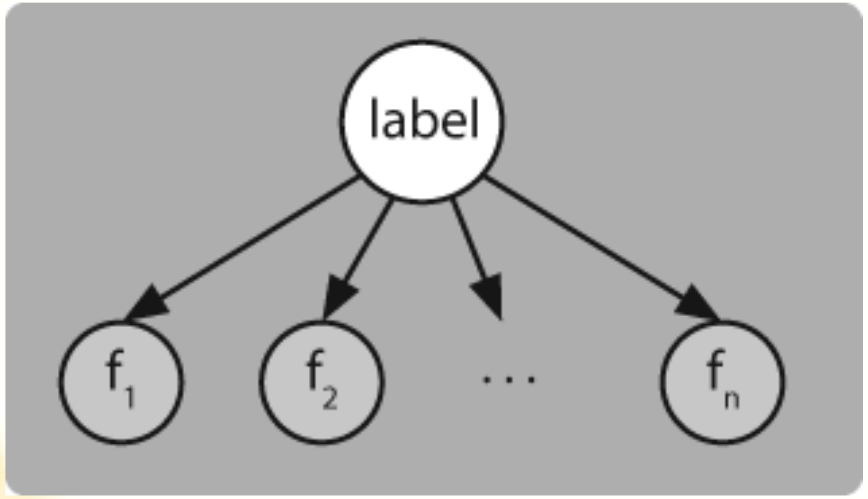

Figure 6-8. A Bayesian Network Graph illustrating the generative process that is assumed by the naive
Bayes classifier. To generate a labeled input, the model first chooses a label for the input, and then it
generates each of the input's features based on that label. Every feature is assumed to be entirely
independent of every other feature, given the label.

Maximum Entropy Classifiers

The **Maximum Entropy** classifier uses a model that is very similar to the model employed
by the naive Bayes classifier. But rather than using probabilities to set the model's parameters, it uses search techniques to find a set of parameters that will maximize the performance of the classifier.

In particular, it looks for the set of parameters that maximizes the **total likelihood** of the training corpus, which is defined as:

$$P(features) = \Sigma x \in \text{corpus } P(label(x)|features(x))$$

Where $P(label|features)$, the probability that an input whose features are *features* will have class label *label*, is defined as:

$$P(label|features) = P(label, features)/\Sigma label\ P(label, features)$$

Because of the potentially complex interactions between the effects of related features, there is no way to directly calculate the model parameters that maximize the likelihood of the training set.

Therefore, Maximum Entropy classifiers choose the model parameters using **iterative optimization** techniques, which initialize the model's parameters to random values, and then repeatedly refine those parameters to bring them closer to the optimal solution.

These iterative optimization techniques guarantee that each refinement of the parameters will bring them closer to the optimal values, but do not necessarily provide a means of determining when those optimal values have been reached.

Because the parameters for Maximum Entropy classifiers are selected using iterative optimization techniques, they can take a long time to learn.

This is especially true when the size of the training set, the number of features, and the number of labels are all large.

The Maximum Entropy Model

The Maximum Entropy classifier model is a generalization of the model used by the naive Bayes classifier.

Like the naive Bayes model, the Maximum Entropy classifier calculates the likelihood of each label for a given input value by multiplying together the parameters that are applicable for the input value and label.
The naive Bayes classifier model defines a parameter for each label, specifying its prior probability, and a parameter for each (feature, label) pair, specifying the contribution of individual features toward a label's likelihood.

In contrast, the Maximum Entropy classifier model leaves it up to the user to decide what combinations of labels and features should receive their own parameters.
In particular, it is possible to use a single parameter to associate a feature with more than one label; or to associate more than one feature with a given label.
This will sometimes allow the model to "generalize" over some of the differences between related labels or features.
Each combination of labels and features that receives its own parameter is called a **joint-feature**. Note that joint-features are properties of *labeled* values, whereas (simple) features are properties of *unlabeled* values.

Typically, the joint-features that are used to construct Maximum Entropy models exactly mirror those that are used by the naive Bayes model.

In particular, a joint-feature is defined for each label, corresponding to w[*label*], and for each combination of (simple) feature and label, corresponding to *w*[*f*, *label*].

Given the joint-features for a Maximum Entropy model, the score assigned to a label for a given input is simply the product of the parameters associated with the joint-features that apply to that input and label:

$$P(input, label) = \prod_{joint\text{-}features(input,label)} w[joint\text{-}feature]$$

Maximizing Entropy

The intuition that motivates Maximum Entropy classification is that we should build a model that captures the frequencies of individual joint-features, without making any unwarranted assumptions.

An example will help to illustrate this principle. Suppose we are assigned the task of picking the correct word sense for a given word, from a list of 10 possible senses (labeled A–J).

At first, we are not told anything more about the word or the senses. There are many probability distributions that we could choose for the 10 senses, such as:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
(i)	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%
(ii)	5%	15%	0%	30%	0%	8%	12%	0%	6%	24%
(iii)	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%

Although any of these distributions *might* be correct, we are likely to choose distribution
(i), because without any more information, there is no reason to believe that any word
sense is more likely than any other. On the other hand, distributions *(ii)* and *(iii)*

One way to capture this intuition that distribution *(i)* is more "fair" than the other two is to invoke the concept of entropy. In the discussion of decision trees, we described entropy as a measure of how "disorganized" a set of labels was.

In particular, if a single label dominates then entropy is low, but if the labels are more evenly distributed then entropy is high.

In our example, we chose distribution *(i)* because its label probabilities are evenly distributed—in other words, because its entropy is high.

In general, the **Maximum Entropy principle** states that, among the distributions that are consistent with what we know, we should choose the distribution whose entropy is highest.

Next, suppose that we are told that sense A appears 55% of the time. Once again, there are many distributions that are consistent with this new piece of information, such as:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
(iv)	55%	45%	0%	0%	0%	0%	0%	0%	0%	0%
(v)	55%	5%	5%	5%	5%	5%	5%	5%	5%	5%
(vi)	55%	3%	1%	2%	9%	5%	0%	25%	0%	0%

But again, we will likely choose the distribution that makes the fewest unwarranted assumptions—in this case, distribution *(v)*.

Finally, suppose that we are told that the word *up* appears in the nearby context 10% of the time, and that when it does appear in the context there's an 80% chance that sense A or C will be used. In this case, we will have a harder time coming up with an appropriate distribution by hand; however, we can verify that the following distribution looks appropriate:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
(vii) +up	5.1%	0.25%	2.9%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%
–up	49.9%	4.46%	4.46%	4.46	4.46%	4.46%	4.46%	4.46%	4.46%	4.46%

In particular, the distribution is consistent with what we know: if we add up the probabilities
in column A, we get 55%; if we add up the probabilities of row 1, we get 10%;
and if we add up the boxes for senses A and C in the +up row, we get 8% (or 80% of the +up cases). Furthermore, the remaining probabilities appear to be "evenly distributed."

Generative Versus Conditional Classifiers

An important difference between the naive Bayes classifier and the Maximum Entropy classifier concerns the types of questions they can be used to answer.

The naive Bayes classifier is an example of a **generative** classifier, which builds a model that predicts
P(*input*, *label*), the joint probability of an (*input*, *label*) pair.

As a result, generative models can be used to answer the following questions:

1. What is the most likely label for a given input?
2. How likely is a given label for a given input?
3. What is the most likely input value?
4. How likely is a given input value?
5. How likely is a given input value with a given label?
6. What is the most likely label for an input that might have one of two values (but we don't know which)?

The Maximum Entropy classifier, on the other hand, is an example of a **conditional** classifier. Conditional classifiers build models that predict *P*(*label*|*input*)—the probability of a label *given* the input value.

Thus, conditional models can still be used to answer questions 1 and 2. However, conditional models *cannot* be used to answer the remaining questions 3–6.

In general, generative models are strictly more powerful than conditional models, since we can calculate the conditional probability *P*(*label*|*input*) from the joint probability *P*(*input, label*), but not vice versa.

However, this additional power comes at a price. Because the model is more powerful, it has more "free parameters" that need to be learned. However, the size of the training set is fixed.

Thus, when using a more powerful model, we end up with less data that can be used to train each parameter's value, making it harder to find the best parameter values.

As a result, a generative model may not do as good a job at answering questions 1 and 2 as a conditional model, since the conditional model can focus its efforts on those two questions. However, if we do need answers to questions like 3–6, then we have no choice but to use a generative model.

The difference between a generative model and a conditional model is analogous to the
difference between a topographical map and a picture of a skyline.

Although the topographical map can be used to answer a wider variety of questions, it is significantly more difficult to generate an accurate topographical map than it is to generate an accurate skyline.

Modeling Linguistic Patterns

Classifiers can help us to understand the linguistic patterns that occur in natural language, by allowing us to create explicit **models** that capture those patterns.

Typically, these models are using supervised classification techniques, but it is also possible to build analytically motivated models.

Either way, these explicit models serve two important purposes: they help us to understand linguistic patterns, and they can be used to make predictions about new language data.

The extent to which explicit models can give us insights into linguistic patterns depends largely on what kind of model is used.

Some models, such as decision trees, are relatively transparent, and give us direct information about which factors are important in making decisions and about which factors are related to one another.

Other models, such as multilevel neural networks, are much more opaque. Although it can be possible to gain insight by studying them, it typically takes a lot more work.

But all explicit models can make predictions about new **unseen** language data that was not included in the corpus used to build the model.

These predictions can be evaluated to assess the accuracy of the model. Once a model is deemed sufficiently accurate, it can then be used to automatically predict information about new language data.

These predictive models can be combined into systems that perform many useful language processing tasks, such as document classification, automatic translation, and question answering.

What Do Models Tell Us?

It's important to understand what we can learn about language from an automatically constructed model.

One important consideration when dealing with models of language is the distinction between descriptive models and explanatory models.

Descriptive models capture patterns in the data, but they don't provide any information about *why* the data contains those patterns.

For example, as we saw in Table 3-1, the synonyms *absolutely* and *definitely* are not interchangeable: we say *absolutely adore* not *definitely adore*, and *definitely prefer*, not *absolutely prefer*. In contrast, explanatory models attempt to capture properties and relationships that cause the linguistic patterns.

For example, we might introduce the abstract concept of "polar adjective" as an adjective that has an extreme meaning, and categorize some adjectives, such as *adore* and *detest* as polar. Our explanatory model would contain the constraint that *absolutely* can combine only with polar adjectives, and *definitely* can only combine with non-polar adjectives.

In summary, descriptive models provide information about correlations in the data, while explanatory models go further to postulate causal relationships.

Most models that are automatically constructed from a corpus are descriptive models;
in other words, they can tell us what features are relevant to a given pattern or construction,
but they can't necessarily tell us how those features and patterns relate to one another.

If our goal is to understand the linguistic patterns, then we can use this information about which features are related as a starting point for further experiments designed to tease apart the relationships between features and patterns.

On the other hand, if we're just interested in using the model to make predictions (e.g., as part of a language processing system), then we can use the model to make predictions about
new data without worrying about the details of underlying causal relationships.

REVA UNIVERSITY
Bengaluru, India

THANK YOU