Chapter 6 - Learning to Classify Text

Jianzhang Zhang

Alibaba Business School
Hangzhou Normal University

May 25, 2022

Table of Contents

Table of Contents

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

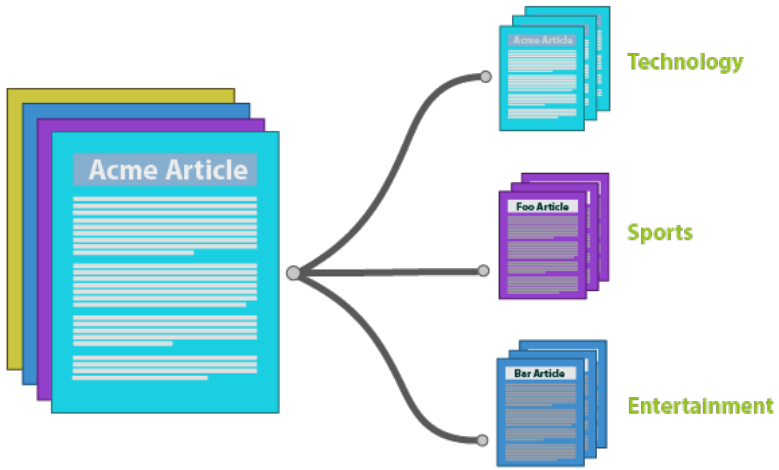


Figure 1: Text classification

There are various applications of classification in NLP, such as deciding what the topic of a news article is from a fixed list (multi-label), word sense disambiguation (multi-class).

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

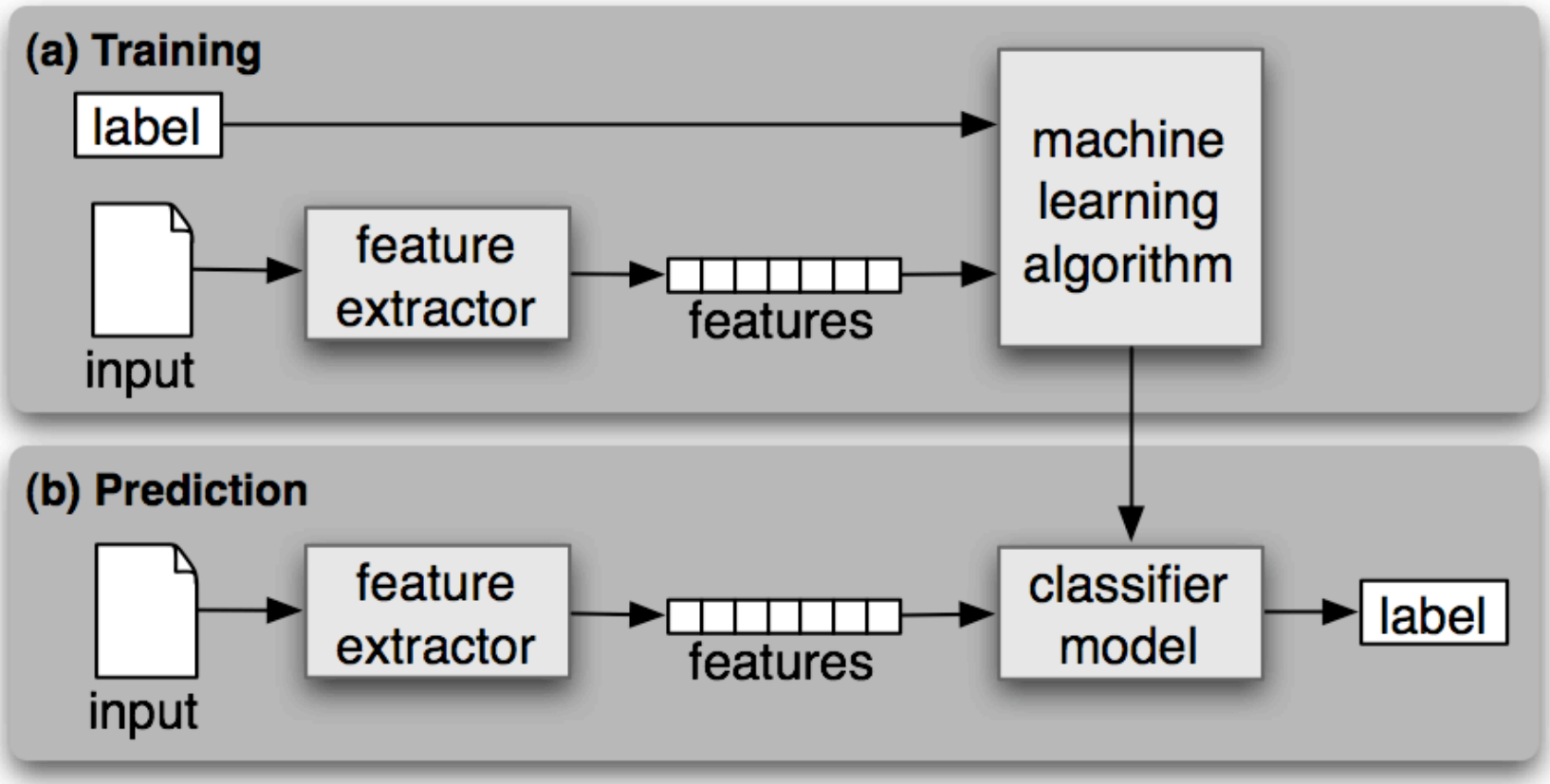


Figure 2: Supervised Classification Framework

Gender Identification

In Chapter 2, we have noticed that names ending in a, e 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.

```
import nltk

def gender_features(word):
  return {'last_letter': word[-1]}

# {'last_letter': 'k'}
gender_features('Shrek')
```

`{'last_letter': 'k'}` is a feature set, maps from feature names to their values.

The second step is to prepare a list of examples and corresponding class labels.

```
import random
from nltk.corpus import names

# [('Aamir', 'male'), ('Aaron', 'male'), ...]
labeled_names = ([(name, 'male') for name in
↪  names.words('male.txt')]
+ [(name, 'female') for name in names.words('female.txt')])

# 将标注数据随机打乱
random.seed(10)
random.shuffle(labeled_names)
```

In the third step, 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**.

```
# [({'last_letter': 'a'}, 'female'), ({'last_letter': 'o'},
↪  'female'), ...,]
featuresets = [(gender_features(n), gender) for (n, gender) in
↪  labeled_names]

train_set, test_set = featuresets[500:], featuresets[:500]
classifier = nltk.NaiveBayesClassifier.train(train_set)
classifier.classify(gender_features('Neo')) # male
```

In the fourth step, we can systematically evaluate the classifier on test set and examine the classifier to determine which features it found most effective for distinguishing the names' genders.

```
print(nltk.classify.accuracy(classifier, test_set)) # 0.77
classifier.show_most_informative_features(5)

'''
Most Informative Features
last_letter = 'a'              female : male =     35.6 : 1.0
last_letter = 'k'              male : female =     32.9 : 1.0
last_letter = 'f'              male : female =     15.4 : 1.0
last_letter = 'p'              male : female =     12.6 : 1.0
last_letter = 'v'              male : female =     11.3 : 1.0
'''
```

These ratios are known as likelihood ratios, and can be useful for comparing different feature-outcome relationships.

Choosing The Right Features

Selecting relevant features and deciding how to encode them for a learning method can have an enormous impact on the learning method's ability to extract a good model.

Although it's often possible to get decent performance by using a fairly simple and obvious set of features, there are usually significant gains to be had by using carefully constructed features based on a thorough understanding of the task at hand.

特征并非越多越好，当训练集较小时，使用过多的特征，会使得最终模型对训练数据产生过拟合，即，模型过分依赖于训练数据过于具体的特点，但是这些具体特点不一定能推广到其他未见数据上，导致模型在其他未见数据上表现不好。

下面的特征抽取函数为判断姓名性别任务抽取了非常丰富的特征，但在测试集上的准确率比前一个模型（只关注姓名最后一个字母是什么）低 1%。

```
def gender_features2(name):
  features = {}
  # 首字母
  features["first_letter"] = name[0].lower()
  # 尾字母
  features["last_letter"] = name[-1].lower()
  for letter in 'abcdefghijklmnopqrstuvwxyz':
    # a-z计数
    features["count({})".format(letter)] =
    ↪  name.lower().count(letter)
    # 是否包含a-z
    features["has({})".format(letter)] = (letter in name.lower())
  return features

featuresets = [(gender_features2(n), gender) for (n, gender) in
↪  labeled_names]
train_set, test_set = featuresets[500:], featuresets[:500]
classifier = nltk.NaiveBayesClassifier.train(train_set)
# 0.756 < 0.77
print(nltk.classify.accuracy(classifier, test_set))
```

Data Splitting

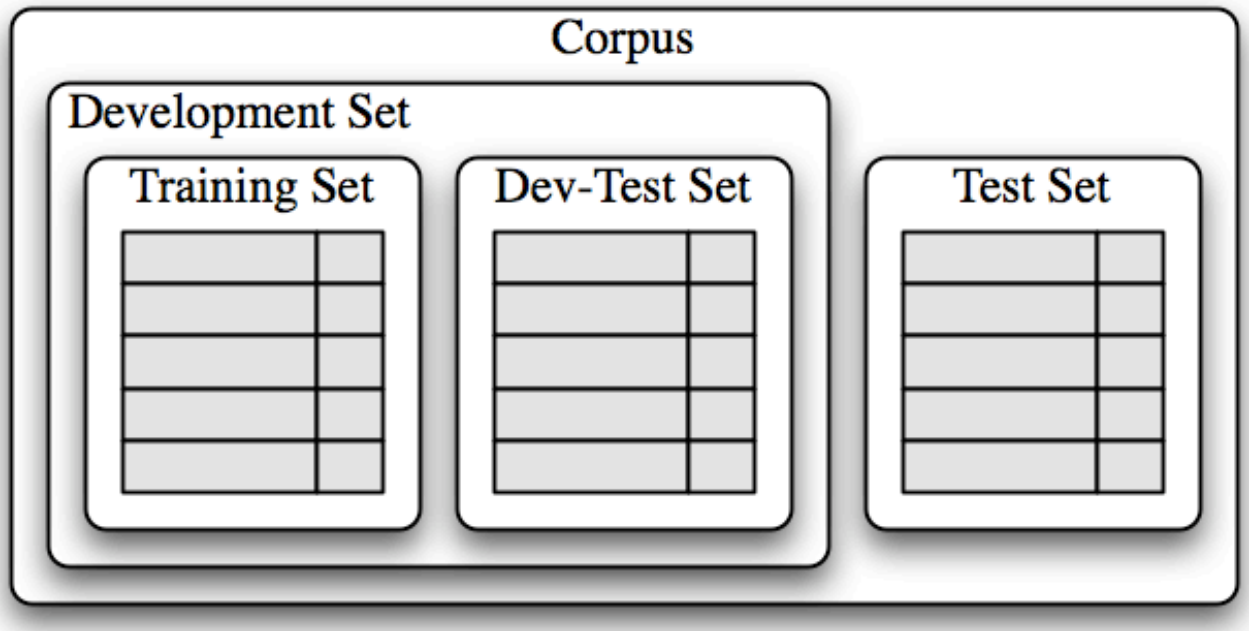


Figure 3: Organization of corpus data for training supervised classifiers

The training set is used to train the model;

The dev-test set is used to perform error analysis;

The test set serves in our final evaluation of the system.

Error Analysis

```
# train and examine individual error cases
train_set = [(gender_features(n), gender) for (n, gender) in
↪  train_names]
devtest_set = [(gender_features(n), gender) for (n, gender) in
↪  devtest_names]
test_set = [(gender_features(n), gender) for (n, gender) in
↪  test_names]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, devtest_set)) # 0.763

errors = []
for (name, tag) in devtest_names:
  guess = classifier.classify(gender_features(name))
  if guess != tag:
    errors.append((tag, guess, name))
for (tag, guess, name) in sorted(errors):
  print('correct={:<8} guess={:<8s} name={:<30}'.format(tag,
  ↪  guess, name))
```

```
correct=female   guess=male     name=Aeriell
correct=female   guess=male     name=Alisun
correct=female   guess=male     name=Allsun
correct=female   guess=male     name=Allyn
correct=female   guess=male     name=Amabel
correct=female   guess=male     name=Amargo
correct=female   guess=male     name=Beilul
correct=female   guess=male     name=Bird
correct=female   guess=male     name=Blair
correct=female   guess=male     name=Britt
correct=female   guess=male     name=Cam
correct=female   guess=male     name=Caril
correct=female   guess=male     name=Carilyn
correct=female   guess=male     name=Carin
correct=female   guess=male     name=Carleen
correct=female   guess=male     name=Caro
correct=female   guess=male     name=Carolann
correct=female   guess=male     name=Cathleen
correct=female   guess=male     name=Cathryn
```

Figure 4: Some error cases in name gender identification task

Some suffixes that are more than one letter can be indicative of name genders. For example, names ending in *yn* appear to be predominantly female, despite the fact that names ending in *n* tend to be male.

We therefore adjust our feature extractor to include features for two-letter suffixes:

```
def gender_features(word):
  return {'suffix1': word[-1:], 'suffix2': word[-2:]}

train_set = [(gender_features(n), gender) for (n, gender) in
↪  train_names]
devtest_set = [(gender_features(n), gender) for (n, gender) in
↪  devtest_names]
classifier = nltk.NaiveBayesClassifier.train(train_set)
# 76.6% > 76.3%
print(nltk.classify.accuracy(classifier, devtest_set))
```

This error analysis procedure can then be repeated. Each time the error analysis procedure is repeated, we should select a different dev-test/training split, to ensure that the classifier does not start to reflect idiosyncrasies in the dev-test set.

Table of Contents

Document Classification

For this example, we choose the Movie Reviews Corpus, which categorizes each review as positive or negative.

```
from nltk.corpus import movie_reviews

# load and shuffle data
documents = [(list(movie_reviews.words(fileid)), category) for
↪  category in movie_reviews.categories() for fileid in
↪  movie_reviews.fileids(category)]
random.shuffle(documents)

# define feature extractor using the frequent words
all_words = nltk.FreqDist(w.lower() for w in
↪  movie_reviews.words())
word_features = list(all_words)[:2000]
```

```
def document_features(document):
  document_words = set(document)
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in
    ↪  document_words)
  return features

print(document_features(movie_reviews.words('pos/cv957_8737.txt'
↪  )))
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))
classifier.show_most_informative_features(5)
```

Part-of-Speech Tagging

We can train a classifier employing suffixes to determine POS tagging of a word.

```
# find the most common suffixes
from nltk.corpus import brown
suffix_fdist = nltk.FreqDist()
for word in brown.words():
  word = word.lower()
  suffix_fdist[word[-1:]] += 1
  suffix_fdist[word[-2:]] += 1
  suffix_fdist[word[-3:]] += 1

common_suffixes = [suffix for (suffix, count) in
↪  suffix_fdist.most_common(100)]
print(common_suffixes)
```

```
# define feature extractor
def pos_features(word):
  features = {}
  for suffix in common_suffixes:
    features['endswith({})'.format(suffix)] =
    ↪  word.lower().endswith(suffix)
  return features

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]
# 此处训练特别慢，建议训练完成后，将模型保存到本地
classifier = nltk.DecisionTreeClassifier.train(train_set)
nltk.classify.accuracy(classifier, test_set) # 0.63
classifier.classify(pos_features('cats')) # NNS
print(classifier.pseudocode(depth=4))
```

```
if endswith(the) == False:
  if endswith(,) == False:
    if endswith(s) == False:
      if endswith(.) == False: return '.'
      if endswith(.) == True: return '.'
    if endswith(s) == True:
      if endswith(is) == False: return 'PP$'
      if endswith(is) == True: return 'BEZ'
  if endswith(,) == True: return ','
if endswith(the) == True: return 'AT'
```

Figure 5: The pseudocode of decision tree

实际的分类器在此处显示的语句下方包含更多嵌套的 `if-then` 语句，`depth=4` 参数仅显示决策树的顶部。

Sequence Classification

In order to capture the dependencies between related classification tasks, we can use joint classifier models, which choose an appropriate labeling for a collection of related inputs.

In the case of POS tagging, a variety of different sequence classifier models can be used to jointly choose part-of-speech tags for all the words in a given sentence.

One sequence classification strategy is consecutive classification or greedy sequence classification (连续分类或贪婪序贯分类):

① Find the most likely class label for the first input;

② Use that answer to help find the best label for the next input;

③ Repeat the process until all of the inputs have been labeled.

The Bigram tagger in Chapter 5 has employed this strategy.

First, we must augment our feature extractor function to take a history argument, which provides a list of the tags that we've predicted for the sentence so far.

```
def pos_features(sentence, i, history):
  features = {"suffix(1)": sentence[i][-1:],
              "suffix(2)": sentence[i][-2:],
              "suffix(3)": sentence[i][-3:]}
  # 将target word的前一个词的pos tag作为特征
  if i == 0:
    features["prev-word"] = "<START>"
    features["prev-tag"] = "<START>"
  else:
    features["prev-word"] = sentence[i-1]
    features["prev-tag"] = history[i-1]
  return features
```

```
class ConsecutivePosTagger(nltk.TaggerI):

    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = pos_features(untagged_sent, i, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
        self.classifier = nltk.NaiveBayesClassifier.train(train_set)

    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = pos_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)

tagged_sents = brown.tagged_sents(categories='news')

size = int(len(tagged_sents) * 0.1)
train_sents, test_sents = tagged_sents[size:], tagged_sents[:size]
tagger = ConsecutivePosTagger(train_sents)
print(tagger.evaluate(test_sents))
```

Figure 6: Consecutive (Bi-gram) POS Tagger

Table of Contents

- Sentence Segmentation;
- Identifying Dialogue Act Types;
- Recognizing Textual Entailment;

For the above three application examples, please refer to the experimental jupyter file on the course website.

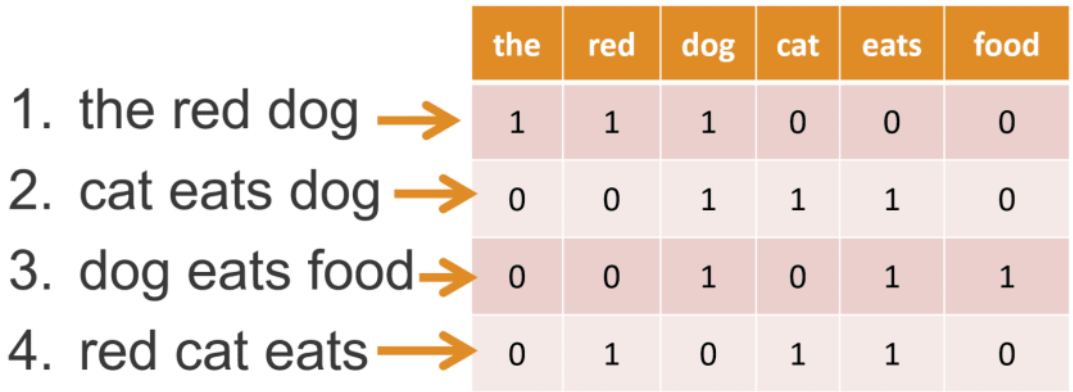
If you plan to train classifiers with large amounts of training data or a large number of features, I recommend that you explore scikit-learn.

Table of Contents

It's difficult to work with text data while building Machine learning models since these models need well-defined numerical data. The process of **Vectorization** converts text data into numerical data/vector.

Bag-of-Words(BoW) and **Word Embedding** are two well-known methods for converting text data to numerical data.

In BOW model, a word is encoded by a one-hot vector, e.g., the word *cat* can be represented as `[0, 0, 0, 1, 0, 0]`. In word embedding model, a word is encoded by a dense float vector, e.g., the word *cat* might be represented as `[0.6, 0.9, 0.4, 0.7, 0.3, 0.2]` by a word embedding model.



There are a few versions of Bag of Words, corresponding to different words scoring methods (**Count, Binary, TF-IDF**).

$$tf\text{-}idf_{i,j} = tf_{i,j} \times \log(\frac{N}{df_i})$$

After vectorization, we can calculate the cosine similarity of two documents as follow:

$$CosineSim(D_1, D_2) = \frac{\mathbf{D_1} \cdot \mathbf{D_2}}{\|\mathbf{D_1}\|\|\mathbf{D_2}\|} = \frac{\sum_1^n D_{1i} D_{2i}}{\sqrt{\sum_1^n D_{1i}^2}\sqrt{\sum_1^n D_{2i}^2}}$$

$\mathbf{D_1}$ and $\mathbf{D_2}$ are vector representations of two documents, e.g., TF-IDF vectors or embedding vectors. If $\mathbf{D_1}$ and $\mathbf{D_2}$ are normalized vector (vectors of length 1), the cosine similarity is equal to the dot product.

Table of Contents

The Test Set

When large amounts of annotated data are available, it is common to err on the side of safety by using 10% of the overall data for evaluation.

下面以 POS tagging 任务为例说明:

```
# 1. 从同一体裁（如news）中选择10%作为测试集，
↪  会使得开发集和测试集的样例非常相似，进而影响模型的评估结果
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]
```

上面代码中使用了 random.shuffle 方法，会使得测试集和训练集使用来自相同文档的句子，进一步加剧了测试集和训练集的相似性，进而影响模型的评估结果。

```
# 2. 对于同一体裁的标注语料，
↪  可以选择让训练数据和测试数据分别来自不同的文档
file_ids = brown.fileids(categories='news')
size = int(len(file_ids) * 0.1)
train_set = brown.tagged_sents(file_ids[size:])
test_set = brown.tagged_sents(file_ids[:size])
# 3. 如果想要进行更严格的评估，可以让训练集和测试集使用不同体裁的文档
train_set = brown.tagged_sents(categories='news')
test_set = brown.tagged_sents(categories='fiction')
```

Accuracy measures the percentage of inputs in the test set that the classifier correctly labeled.

在不均衡数据集中会产生误导：在词义消歧中，bank 一词在金融语料测试数据中出现 20 次，其中 19 次含义为银行，如果分类器的 accuracy 为 95%，则意义不大。在文档检索中，不相关的文档数量远大于相关文档的数量，此时分类器若把所有文档均预测为不相关，便可以达到接近 100% 的 accuracy。

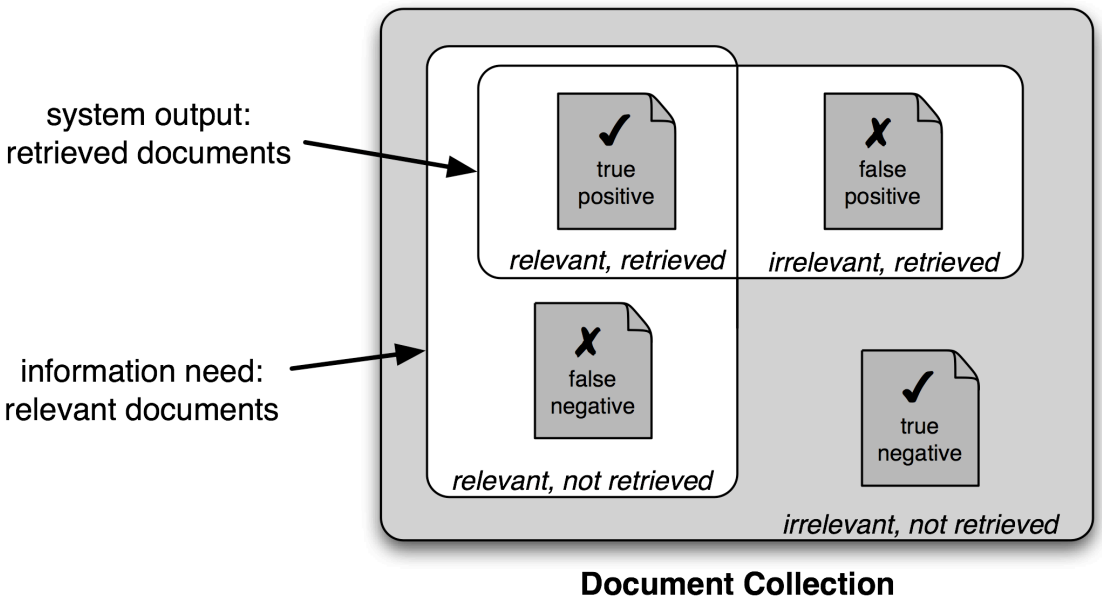


Figure 7: True and False Positives and Negatives

- **True positives**: relevant items -> relevant;
- **True negatives**: irrelevant items -> irrelevant;
- **False positives** (or Type I errors): irrelevant items -> relevant;
- **False negatives** (or Type II errors): relevant items -> irrelevant.

$$Precision = \frac{TP}{TP + FP}$$

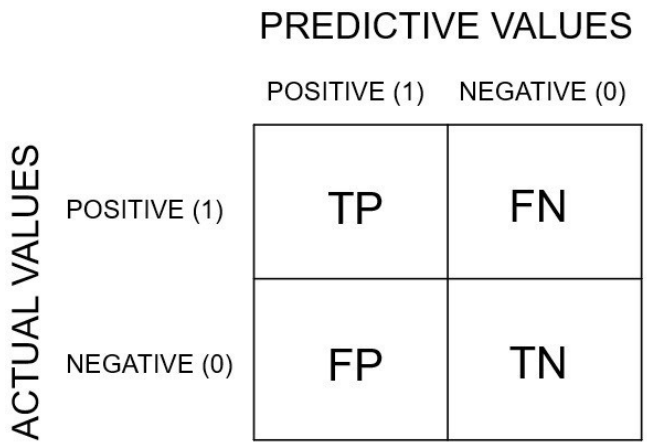
$$Recall = \frac{TP}{TP + FN}$$

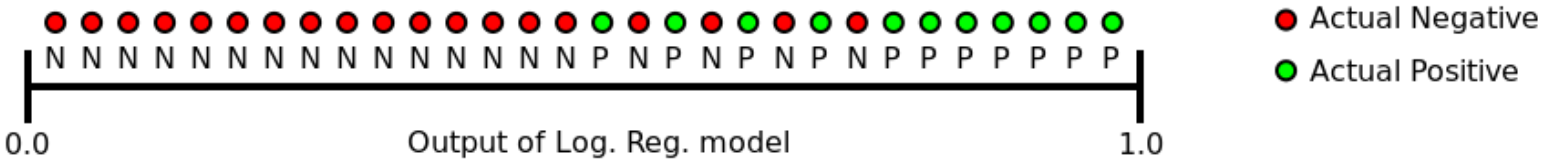
$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

An ROC curve (receiver operating characteristic curve, 受试者工作特征曲线) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

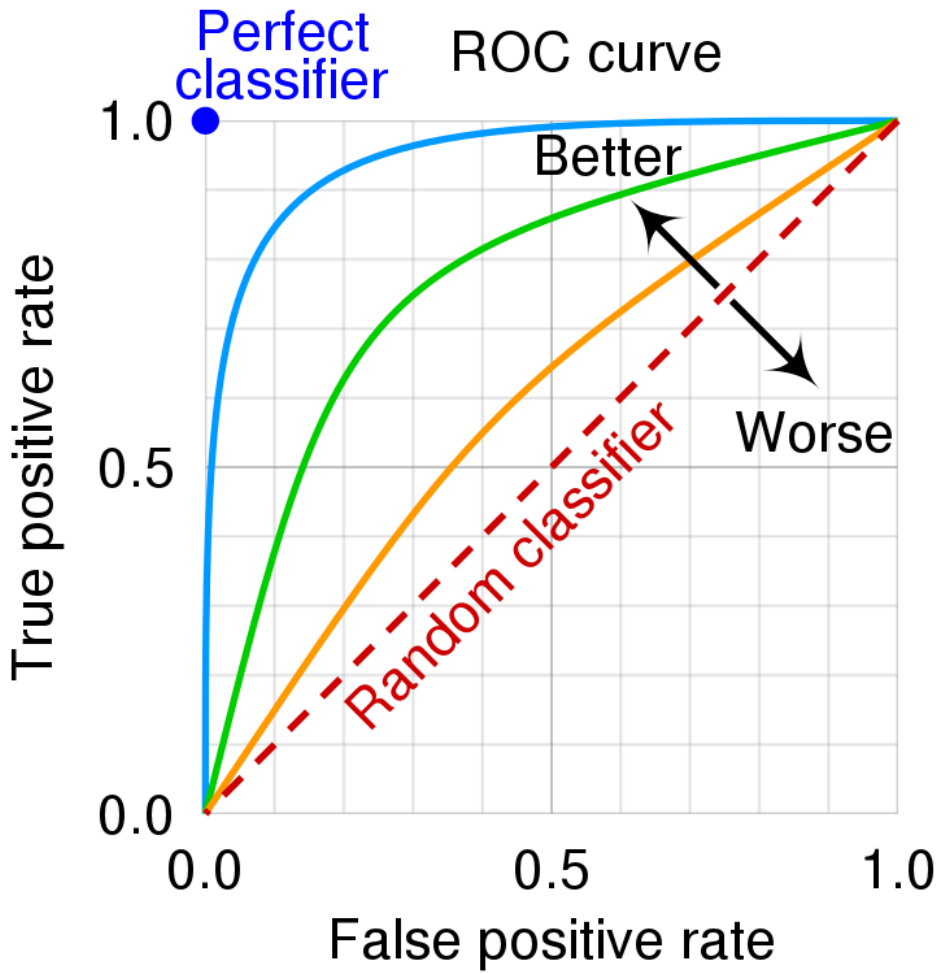
$$True\ Positive\ Rate, TPR = \frac{TP}{TP + FN}$$

$$False\ Positive\ Rate, FPR = \frac{FP}{FP + TN}$$





An ROC curve plots TPR vs. FPR at different classification thresholds. AUC stands for "Area under the ROC Curve". For different classifiers, we prefer the one with the biggest AUC as it has the best discrimination ability.



Error Analysis

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`.

	NN	IN	AT	JJ	.	NNS	,	VB	NP
NN	<11.8%>	0.0%	.	0.2%	.	0.0%	.	0.3%	0.0%
IN	0.0%	<9.0%>	.	.	.	0.0%	.	.	.
AT	.	.	<8.6%>	.	.	.	.	.	.
JJ	1.7%	.	.	<3.9%>	.	.	.	0.0%	0.0%
.	.	.	.	.	<4.8%>	.	.	.	.
NNS	1.5%	.	.	.	.	<3.2%>	.	.	0.0%
,	.	.	.	.	.	.	<4.4%>	.	.
VB	0.9%	.	.	0.0%	.	.	.	<2.4%>	.
NP	1.0%	.	.	0.0%	.	.	.	.	<1.8%>

(row = reference; col = test)

Figure 8: Confusion matrix for the bigram tagger

Cross-Validation

If the test set is too small, then 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.

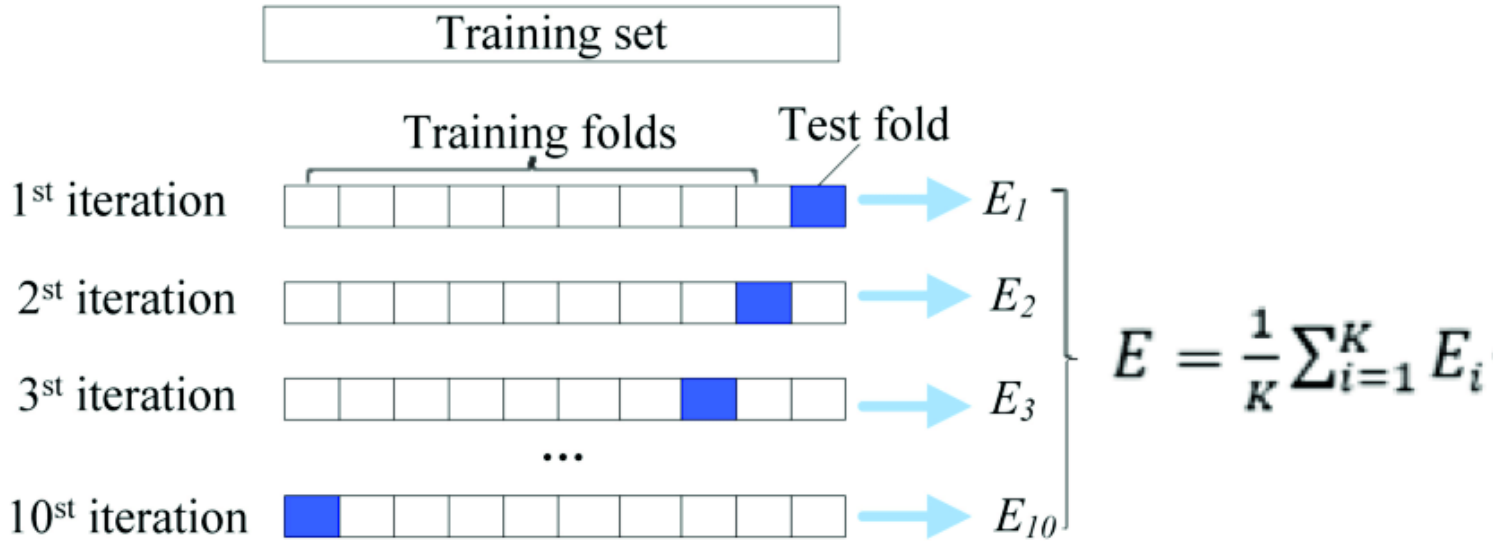


Figure 9: 10 Fold Cross-Validation

Cross-validation also allow us to examine how widely the performance varies across different training sets.

Table of Contents

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.

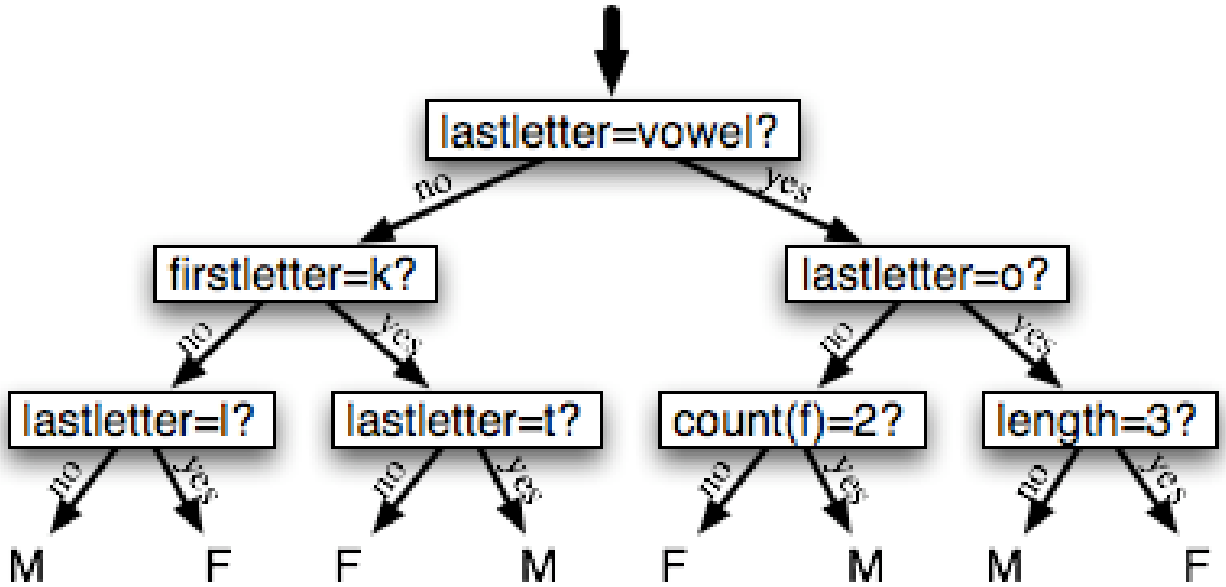


Figure 10: Decision Tree model for the name gender task

To choose the label for an input value, we begin at the flowchart's initial decision node (root node) and 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.

Before studying the learning algorithm for building decision trees, we'll consider a simpler task: picking the best "**decision stump**" for a corpus.

A decision stump is a decision tree with a single node that decides how to classify inputs based on a single feature.

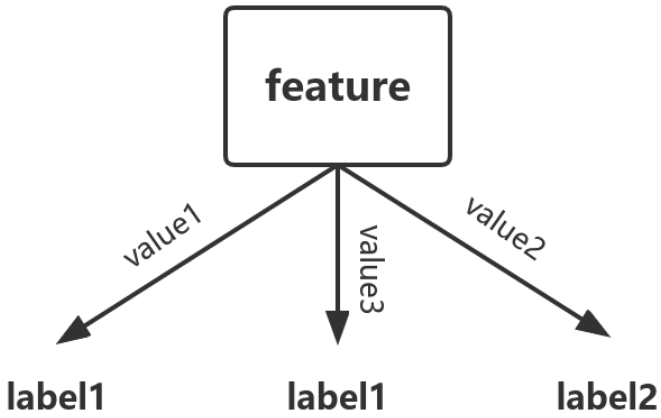


Figure 11: Decision stump example

We could firstly build a decision stump for each possible feature and pick the one achieving the highest accuracy on the training data. Then we assign a label to each leaf based on the most frequent label for the selected examples in the training set (i.e., the examples where the selected feature has that value).

Given the algorithm for choosing decision stumps, the algorithm for growing larger decision trees is straightforward.

1. Select the overall best decision stump for the classification task;
2. Check the accuracy of each of the leaves on the training set;
3. Leaves that do not achieve sufficient accuracy are then replaced by new decision stumps, trained on the subset of the training corpus that is selected by the path to the leaf;

For example, we could grow the decision tree in Figure 11 by replacing the leftmost leaf with a new decision stump, trained on the subset of the training set names that do not start with a "k" and do not end with an "l".

One popular method for identifying the most informative feature for a decision is **information gain (IG)**.

IG measures how much more organized the input values become when we divide them up using a given feature. (衡量有序程度的变化)

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. (衡量有序程度)

$$H = -\sum_{l \ in \ labels} P(l) \times \log_2 P(l)$$

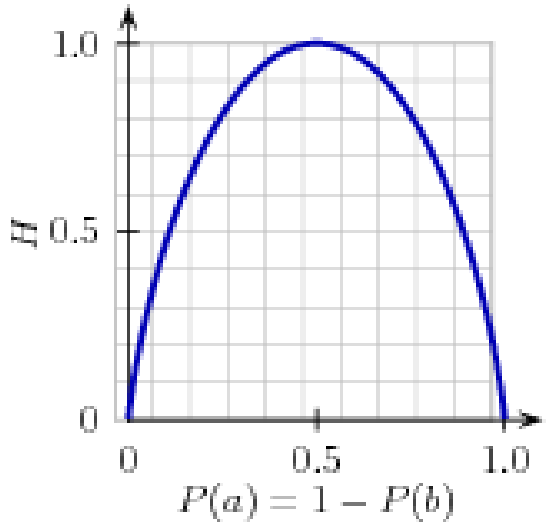


Figure 12: The entropy of labels in the name gender prediction task

If most input values have the same label (e.g., if $P(male)$ is near 0 or near 1), then entropy is low.

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).

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.

```python
import math
def entropy(labels):
  # 频数统计
  freqdist = nltk.FreqDist(labels)
  # 计算每个标签的频率值
  probs = [freqdist.freq(l) for l in freqdist]
  # 计算熵值
  return -sum(p * math.log(p,2) for p in probs)

labels_list = ['male'] * 98 + ['female'] * 2
print(entropy(labels_list)) # 0.14
labels_list = ['male'] * 50 + ['female'] * 50
print(entropy(labels_list)) # 1.0
```

Once we have calculated the entropy of the original set of input values' labels, we can determine how much more organized the labels become once we apply the decision stump:

1. Calculate the entropy for each of the decision stump's leaves;
2. Take the average of those leaf entropy values (weighted by the number of samples in each leaf);
3. The information gain is then equal to the original entropy minus this new, reduced entropy.

The higher the information gain, the better job the decision stump does of dividing the input values into coherent groups.

Advantages: simple to understand, and easy to interpret.

Disadvantages: the amount of training data available to train nodes lower in the tree can become quite small, as each branch in the decision tree splits the training data. Thus, these lower decision nodes may **overfit** the training set (stop dividing nodes once the amount of training data becomes too small; grow a full decision tree, but then to **prune** decision nodes that do not improve performance on a dev-test).

Table of Contents

伯努利分布：抛一次硬币，正面朝上的概率；

$$P(X = x|\theta) = \theta^x(1-\theta)^(1-x)$$

二项分布：抛 n 次硬币，正面朝上出现了 m 次的概率；

$$P(X = m|\theta, n) = \frac{n!}{m!(n-m)!}\theta^m(1-\theta)^{(n-m)}$$

categorical 分布：抛一次骰子，第 k 面朝上的概率；

$$P(X = x_k|\theta_1, \theta_2, ..., \theta_K) = \prod_{k=1}^{K}\theta_k^{x_k}$$

$$\sum_{k=1}^{K}\theta_k = 1$$

$$\sum_{k=1}^{K}x_k = 1,\ x_k \in \{0, 1\}$$

多项分布：抛 n 次骰子，第 1 面朝上出现了 m_1 次，第 2 面朝上出现了 m_2 次..... 第 K 面朝上出现了 m_K 次的概率。

$$P(X_1 = m_1.X_2 = m_2, ..., X_K = m_K|\theta_1, \theta_2, ..., \theta_K, n) = \frac{n!}{m_1!, ..., m_K!} \prod_{k=1}^{K} \theta_k^{m_k}$$

$$\sum_{k=1}^{K} \theta_k = 1$$

$$\sum_{k=1}^{K} m_k = n$$

在朴素贝叶斯分类器中，每个特征都对标签分配都有投票权，对于一个输入样本，标签分配过程为：

① 计算每个标签的先验概率；

② 将每个特征的贡献与标签的先验概率结合，得出标签的似然估计。

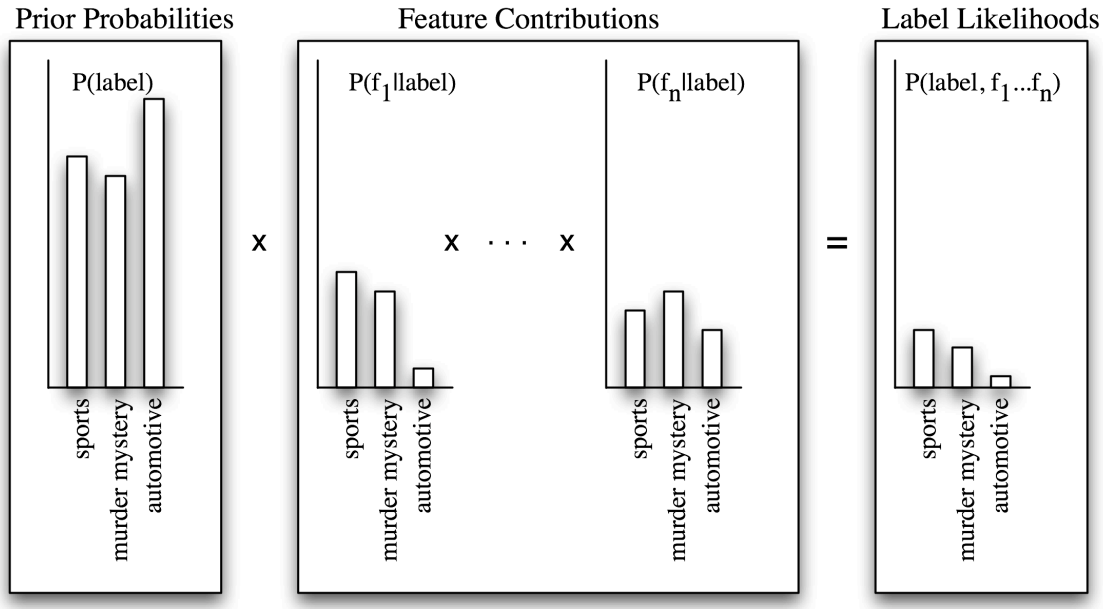


Figure 13: Calculating label likelihoods with naive Bayes.

$$P(label) \cdot \prod_{features} P(feature|label) = P(features, label) \propto P(label|features)$$

基于条件独立假设 $P(AB|C) = P(A|C)P(B|C)$ (**Naive** because this assumption is unrealistic)，可以根据贝叶斯公式和全概率公式计算 $P(label \mid features)$ (即，给定一个用 $features$ 表示的输入，该输入被分配标签 $label$ 的条件概率) 如下：

$$
\begin{aligned}
P(C_k \mid \mathbf{x}) &= \frac{P(\mathbf{x}, C_k)}{P(\mathbf{x})} \\
&= \frac{P(C_k) \cdot P(\mathbf{x} \mid C_k)}{\sum_k P(\mathbf{x}, C_k)} \\
&= \frac{P(C_k) \cdot P(\mathbf{x} \mid C_k)}{\sum_k P(C_k) \cdot P(\mathbf{x} \mid C_k)} \\
&\propto P(C_k) \cdot \prod_{i=1}^{n} P(x_i \mid C_k)
\end{aligned}
$$

$$
P(C_k) = \frac{Count(C_k)}{N}, \; P(x_i \mid C_k) = \frac{Count(x_i, C_k)}{Count(C_k)}
$$

其中，$\mathbf{x} = (x_1, x_2, ..., x_n)$，表示特征向量，$n$ 为特征数量，m 表示训练样本总数，$Count(C_k)$ 表示类别为 k 的训练样本数，$Count(x_i, C_k)$ 表示类别标签为 C_k 的样本中，包含特征 x_i 的样本数量，:

$$
\begin{aligned}
P(C_k \mid \mathbf{x}) &= \frac{P(\mathbf{x}, C_k)}{P(\mathbf{x})} \\
&= \frac{P(C_k) \cdot P(\mathbf{x} \mid C_k)}{\sum_k P(\mathbf{x}, C_k)} \\
&= \frac{P(C_k) \cdot P(\mathbf{x} \mid C_k)}{\sum_k P(C_k) \cdot P(\mathbf{x} \mid C_k)} \\
&\propto P(C_k) \cdot \prod_{i=1}^{n} P(x_i \mid C_k)
\end{aligned}
$$

$$
P(C_k) = \frac{Count(C_k)}{m}, \quad P(x_i \mid C_k) = \frac{Count(x_i, C_k)}{Count(C_k)}
$$

Table 1: Sample-feature matrix

	x_1	x_2	...	x_n	**class**
S_1	v_{11}	v_{12}	...	v_{1n}	C_2
S_2	v_{21}	v_{22}	...	v_{2n}	C_3
...	...	...	...	...	...
S_m	v_{m1}	v_{m2}	...	v_{mn}	C_k

Table 2: Boy-characteristic matrix

	Height	*Weight*	*Appearance*	*Income*	**class**
Boy_1	178	70	*handsome*	$[50, 100]$	Y
Boy_2	185	75	*normal*	$[30, 50]$	Y
...	...	...	...	...	...
Boy_m	168	60	*ugly*	$[10, 20]$	N

Height 和 *Weight* 是（近似）服从正态分布的特征，*Appearance* 是服从 *Categorical* 分布的特征，*Income* 原本是连续的，但是按照区间划分后可以看做是服从 *Categorical* 分布的特征。

如果每个特征 x_i 都服从**高斯分布**（在类别 k 中，该特征取值的均值为 μ_k，方差为 σ_k），则：

$$P(x_i = v \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(v-\mu_k)^2}{2\sigma_k^2}}$$

如果特征集整体服从**多项式分布**，则：

$$P(\mathbf{x} \mid C_k) = \frac{(\sum_{i=1}^{n} x_i)!}{\prod_{i=1}^{n} x_i!} \prod_{i=1}^{n} p_{ki}^{x_i}$$

其中，x_i 表示第 i 个特征出现的次数，p_{ki} 表示在类别 k 中，第 i 个特征出现的概率。

想象一下，现在对于每一个类别 C_k，都有一个 n 面体的骰子，每个面对应 *vocabulary* 中的一个词语，如果一个文档 *Doc* 包含 100 个词语，那么我们就投掷这个骰子 100 次，该文档的生成概率 $P(\mathbf{x} \mid C_k)$ 如上。

	football	*Trump*	...	*love*	**class**
Doc_1	10	0	...	1	*sports*
Doc_2	0	5	...	1	*political*
...	...	...	...	...	...
Doc_m	1	0	...	8	*romantic*

Figure 14: Faceted dice of class C_k

如果每个特征 x_i 都服从**伯努利（0-1）分布**（p_{ki} 表示在第 k 个类别中，特征 x_i 发生的概率），则：

$$P(\mathbf{x} \mid C_k) = \prod_{i=1}^{n} p_{ki}^{x_i}(1 - p_{ki})^{(1-x_i)}$$

想象一下，现在对于每一个类别 C_k，都有 n 个硬币（不一定均匀），每个硬币对应 $vocabulary$ 中的一个词语，正面朝上表示文档中出现该词语，如果一个文档 Doc 不重复的词语数为 100 个，那么我们会把 n 枚硬币都抛掷一次，最后正面朝上的硬币个数是 100，该文档的生成概率 $P(\mathbf{x} \mid C_k)$ 如上。

	football	*Trump*	...	*love*	**class**
Doc_1	1	0	...	0	*sports*
Doc_2	0	1	...	0	*political*
...	...	...	...	...	...
Doc_m	0	0	...	1	*romantic*

Smoothing (平滑): 如果在训练集中 $count(feature, label)$ 的计数为0，乘积计算会导致 $P(label \mid features) = 0$，无论其他 $feature$ 对该 $label$ 的指示性有多强，该标签也不会被分配给当前样本，为避免这种情况，在计数时通常进行平滑操作，如 $count(feature, label) + 0.5$。

独立性假设的缺陷：现实中，各特征之间并非完全独立的，忽略这种特征依赖性会导致分类器重复计算相互之间高度依赖特征的影响（对同一个信息内容赋予过多的权重），使得分类结果产生更大偏差。

$$P(label_1 \mid features) \propto P(label_1) \cdot \prod_{features} P(feature|label_1) \cdot P(f_{new} \mid label_1)$$

$$P(label_2 \mid features) \propto P(label_2) \cdot \prod_{features} P(feature|label_2) \cdot P(f_{new} \mid label_2)$$

考虑一种极端情况，假如新增特征 f_{new} 与某个已有特征完全重复，则 $\frac{P(label_1 \mid features)}{P(label_2 \mid features)}$ 将发生变化，使分类结果以不合理的方式更偏向某一个标签。

思考 1：如果所有特征都服从 $Categorical$ 分布，$P(\mathbf{x} \mid C_k)$ 该如何计算。

思考 2：某男生的特点是{不帅，性格好，身高矮，上进}，请你基于下图中的训练数据，用朴素贝叶斯分类器判断一下女生是嫁还是不嫁。

帅？	性格好？	身高？	上进？	嫁与否
帅	不好	矮	不上进	不嫁
不帅	好	矮	上进	不嫁
帅	好	矮	上进	嫁
不帅	好	高	上进	嫁
帅	不好	矮	上进	不嫁
帅	不好	矮	上进	不嫁
帅	好	高	不上进	嫁
不帅	好	中	上进	嫁
帅	好	中	上进	嫁
不帅	不好	高	上进	嫁
帅	好	矮	不上进	不嫁
帅	好	矮	不上进	不嫁

Table of Contents

Explicit models (supervised classifiers, analytically motivated models) serve two important purposes: ① help us to understand linguistic patterns; ② 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 more transparent than other models, e.g., neural networks.

Descriptive models provide information about correlations in the data (what features are relevant to a given pattern or construction), while **explanatory models** go further to postulate causal relationships (how those features and patterns relate to one another). Most models that are automatically constructed from a corpus are descriptive models.

For better understanding the linguistic patterns, we can use the information about which features are related as a starting point for further experiments designed to tease apart the relationships between features and patterns.

THE END