Learning to Classify Text

Detecting patterns is a central part of Natural Language Processing. Words ending in *-ed* tend to be past tense verbs. Frequent use of *will* is indicative of news text. These observable patterns — word structure and word frequency — happen to correlate with particular aspects of meaning, such as tense and topic. But how did we know where to start looking, which aspects of form to associate with which aspects of meaning?

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?

Along the way we will study some important machine learning techniques, including decision trees, naive Bayes' classifiers, and maximum entropy classifiers. We will gloss over the mathematical and statistical underpinnings of these techniques, focusing instead on how and when to use them (see the Further Readings section for more technical background). Before looking at these methods, we first need to appreciate the broad scope of this topic.

1 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 act of tilting to the side, or the act of depositing something in a financial institution.

The basic classification task has a number of interesting variants. For example, in multi-class 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 1.1.

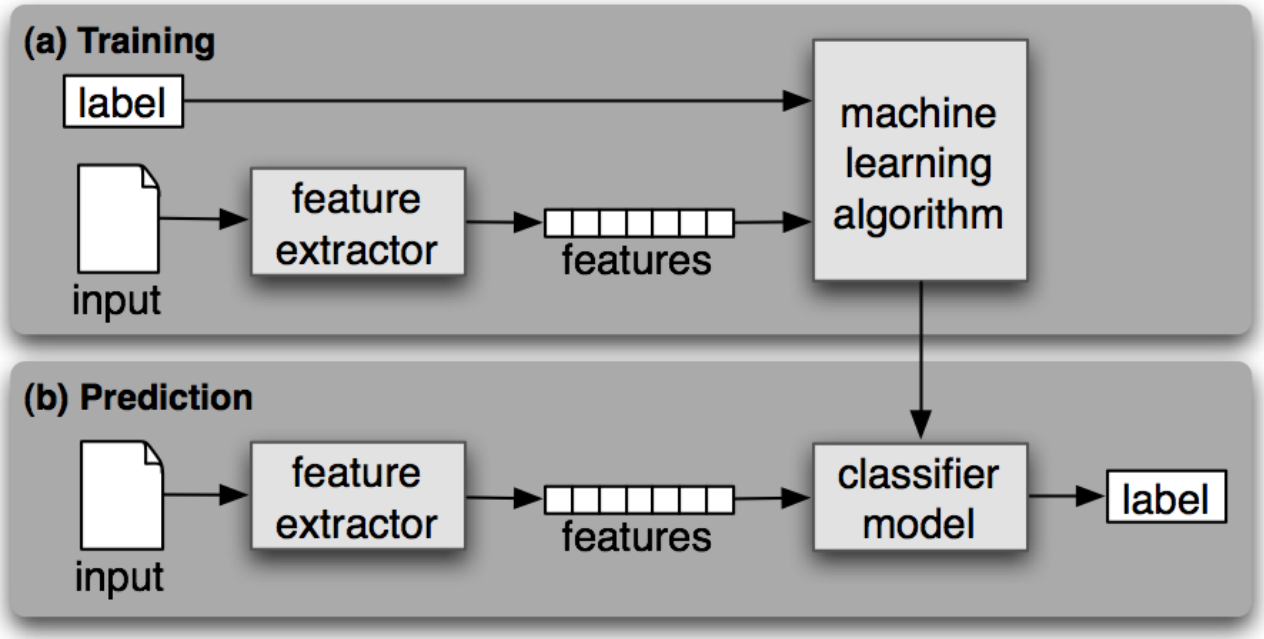

***Figure 1.1**: Supervised Classification. (a) During training, a feature extractor is used to convert each input value to a feature set. These feature sets, which capture the basic information about each input that should be used to classify it, are discussed in the next section. Pairs of feature sets 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.*

In the rest of this section, we will look at how classifiers can be employed to solve a wide variety of tasks. Our discussion is not intended to be comprehensive, but to give a representative sample of tasks that can be performed with the help of text classifiers.

1.1 Gender Identification

In 4 we saw that male and female names have some distinctive characteristics. 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. 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 returned dictionary, known as a **feature set**, maps from feature names to their values. Feature names are case-sensitive strings that typically provide a short human-readable description of the feature, as in the example 'last_letter'. Feature values are values with simple types, such as booleans, numbers, and strings.

Note

Most classification methods require that features be encoded using simple value types, such as booleans, numbers, and strings. But note that just because a feature has a simple type, this does not necessarily mean that the feature's value is simple to express or compute. Indeed, it is even possible to use very complex and informative values, such as the output of a second supervised classifier, as features.

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
>>> labeled_names = ([(name, 'male') for name in names.words('male.txt')] +
... [(name, 'female') for name in names.words('female.txt')])
>>> import random
>>> random.shuffle(labeled_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), gender) for (n, gender) in labeled_names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

We will learn more about the naive Bayes classifier later in the chapter. 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. Although this science fiction movie is set in 2199, it still conforms with our expectations about names and genders. We can systematically evaluate the classifier on a much larger quantity of unseen data:

```
>>> print(nltk.classify.accuracy(classifier, test_set))
0.77
```

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   =     33.2 : 1.0
        last_letter = 'k'            male : female =     32.6 : 1.0
        last_letter = 'p'            male : female =     19.7 : 1.0
        last_letter = 'v'            male : female =     18.6 : 1.0
        last_letter = 'f'            male : female =     17.3 : 1.0
```

This listing shows that the names in the training set that end in "a" are female 33 times more often than they are male, but names that end in "k" are male 32 times more often than they are female. These ratios are known as **likelihood ratios**, and can be useful for comparing different feature-outcome relationships.

Note

Your Turn: Modify the gender_features() function to provide the classifier with features encoding the length of the name, its first letter, and any other features that seem like they might be informative. Retrain the classifier with these new features, and test its accuracy.

When working with large corpora, constructing a single list that contains the features of every instance can use up a large amount of memory. In these cases, use the function nltk.classify.apply_features, which returns an object that acts like a list but does not store all the feature sets in memory:

```
>>> from nltk.classify import apply_features
>>> train_set = apply_features(gender_features, labeled_names[500:])
>>> test_set = apply_features(gender_features, labeled_names[:500])
```

1.2 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. Much of the interesting work in building a classifier is deciding what features might be relevant, and how we can represent them. 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.

Typically, feature extractors are built through a process of trial-and-error, guided by intuitions about what information is relevant to the problem. It's common to start with a "kitchen sink" approach, including all the

features that you can think of, and then checking to see which features actually are helpful. We take this approach for name gender features in 1.2.

```
def gender_features2(name):
    features = {}
    features["first_letter"] = name[0].lower()
    features["last_letter"] = name[-1].lower()
    for letter in 'abcdefghijklmnopqrstuvwxyz':
        features["count({})".format(letter)] = name.lower().count(letter)
        features["has({})".format(letter)] = (letter in name.lower())
    return features
```

```
>>> gender_features2('John')
{'count(j)': 1, 'has(d)': False, 'count(b)': 0, ...}
```

Example 1.2 (code_gender_features_overfitting.py): **Figure 1.2**: A Feature Extractor that Overfits Gender Features. The feature sets returned by this feature extractor contain a large number of specific features, leading to overfitting for the relatively small Names Corpus.

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 1.2, 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), gender) for (n, gender) in labeled_names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print(nltk.classify.accuracy(classifier, test_set))
0.768
```

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 = labeled_names[1500:]
>>> devtest_names = labeled_names[500:1500]
>>> test_names = labeled_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 below, 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 1.3.

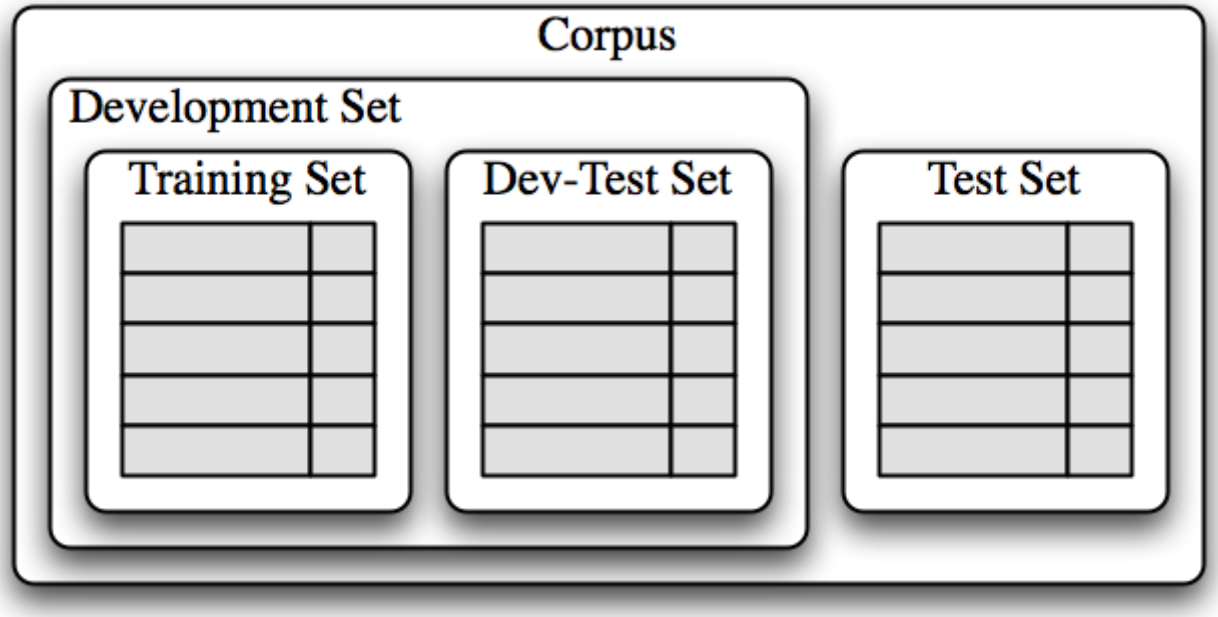

***Figure 1.3**: Organization of corpus data for training supervised classifiers. The corpus data is divided into two sets: the development set, and the test set. The development set is often further subdivided into a training set and a dev-test set.*

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

```
>>> 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) ❶
```

```
>>> print(nltk.classify.accuracy(classifier, devtest_set))
0.75
```

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) )
```

We can then examine individual error cases where the model predicted the wrong label, and try to determine what additional pieces of information would allow it to make the right decision (or which existing pieces of information are tricking it into making the wrong decision). The feature set can then be adjusted accordingly. The names classifier that we have built generates about 100 errors on the dev-test corpus:

```
>>> for (tag, guess, name) in sorted(errors):
...     print('correct={:<8} guess={:<8s} name={:<30}'.format(tag, guess, name))
correct=female   guess=male    name=Abigail
  ...
correct=female   guess=male    name=Cindelyn
  ...
correct=female   guess=male    name=Katheryn
correct=female   guess=male    name=Kathryn
```

```
  ...
correct=male    guess=female   name=Aldrich
  ...
correct=male    guess=female   name=Mitch
  ...
correct=male    guess=female   name=Rich
  ...
```

Looking through this list of errors makes it clear that 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; and names ending in *ch* are usually male, even though names that end in *h* tend to be female. We therefore adjust our feature extractor to include features for two-letter suffixes:

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

Rebuilding the classifier with the new feature extractor, we see that the performance on the dev-test dataset improves by almost 2 percentage points (from 76.5% to 78.2%):

```
>>> 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)
>>> print(nltk.classify.accuracy(classifier, devtest_set))
0.782
```

This error analysis procedure can then be repeated, checking for patterns in the errors that are made by the newly improved classifier. 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.

But once we've used the dev-test set to help us develop the model, we can no longer trust that it will give us an accurate idea of how well the model would perform on new data. It is therefore important to keep the test set separate, and unused, until our model development is complete. At that point, we can use the test set to evaluate how well our model will perform on new input values.

1.3 Document Classification

In 1, we saw several examples of corpora where documents have been labeled 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 (1.4). 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 2000 most frequent words in the overall corpus ❶. We can then define a feature extractor ❷ that simply checks whether each of these words is present in a given document.

```
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
word_features = list(all_words)[:2000] ❶

def document_features(document): ❷
    document_words = set(document) ❸
    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')))
{'contains(waste)': False, 'contains(lot)': False, ...}
```

Example 1.4 (code_document_classify_fd.py): **Figure 1.4**: A feature extractor for document classification, whose features indicate whether or not individual words are present in a given document.

Note

The reason that we compute the set of all words in a document in ❸, rather than just checking if word in document, is that checking whether a word occurs in a set is much faster than checking whether it occurs in a list (4.7).

Now that we've defined our feature extractor, we can use it to train a classifier to label new movie reviews (1.5). To check how reliable the resulting classifier is, we compute its accuracy on the test set ❶. And once again, we

can use show_most_informative_features() to find out which features the classifier found to be most informative ❷.

```
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)) ❶
0.81
>>> classifier.show_most_informative_features(5) ❷
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
```

Example 1.5 (code_document_classify_use.py): **Figure 1.5**: Training and testing a classifier for document classification.

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.

1.4 Part-of-Speech Tagging

In 5. we built a regular expression tagger that chooses a part-of-speech tag for a word by looking at the internal make-up of the word. However, this regular expression tagger had to be hand-crafted. Instead, we can train a

classifier to work out which suffixes are most informative. Let's begin by finding out what the most common suffixes are:

```
>>> 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)
['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 which checks a given word for these suffixes:

```
>>> def pos_features(word):
...     features = {}
...     for suffix in common_suffixes:
...         features['endswith({})'.format(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 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)
0.62705121829935351
```

```
>>> classifier.classify(pos_features('cats'))
'NNS'
```

One nice feature of decision tree models is that they are often fairly easy to interpret — we can even instruct NLTK to print them out as pseudocode:

```
>>> print(classifier.pseudocode(depth=4))
if endswith(,) == True: return ','
if endswith(,) == False:
```

```
if endswith(the) == True: return 'AT'
if endswith(the) == False:
 if endswith(s) == True:
  if endswith(is) == True: return 'BEZ'
  if endswith(is) == False: return 'VBZ'
 if endswith(s) == False:
  if endswith(.) == True: return '.'
  if endswith(.) == False: return 'NN'
```

Here, we can see that the classifier begins by checking whether a word ends with a comma — if so, then it will receive the special tag ",". Next, the classifier checks if the word ends in "the", in which case it's almost certainly a determiner. This "suffix" gets used early by the decision tree because the word "the" is so common. Continuing on, the classifier checks if the word ends in "s". If so, then it's most likely to receive the verb tag VBZ (unless it's the word "is", which has a special tag BEZ), and if not, then it's most likely a noun (unless it's the punctuation mark "."). The actual classifier contains further nested if-then statements below the ones shown here, but the depth=4 argument just displays the top portion of the decision tree.

Naïve Bayes Classifier Algorithm

- Naïve Bayes algorithm is a supervised learning algorithm, which is based on **Bayes theorem** and used for solving classification problems.
- It is mainly used in *text classification* that includes a high-dimensional training dataset.
- **It is a probabilistic classifier, which means it predicts on the basis of the probability of an object**.
- Some popular examples of Naïve Bayes Algorithm are **spam filtration, Sentimental analysis, and classifying articles**.

Why is it called Naïve Bayes?

The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:

- **Naïve**: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
- **Bayes**: It is called Bayes because it depends on the principle of Bayes' Theorem.

Bayes' Theorem:

- Bayes' theorem is also known as **Bayes' Rule** or **Bayes' law**, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.
- The formula for Bayes' theorem is given as:

$$P(A|B)= \frac{P(B|A)P(A)}{P(B)}$$

Where,

P(A | B) is Posterior probability: Probability of hypothesis A on the observed event B.

(The revised or updated probability of an event occurring after taking into consideration new information.)

P(B | A) is Likelihood probability: Probability of the evidence given that the probability of a hypothesis is true.

(Probability of getting result for a given value of the parameters)

P(A) is Prior Probability: Probability of hypothesis before observing the evidence. (**The probability of an event before new data is collected)**

P(B) is Marginal Probability: Probability of Evidence.

(The marginal probability is the probability of a single event occurring, independent of other events).

(Conditional probability is the probability that an event occurs given that another specific event has already occurred)

Working of Naïve Bayes' Classifier:

Working of Naïve Bayes' Classifier can be understood with the help of the below example:

Suppose we have a dataset of **weather conditions** and corresponding target variable "**Play**". So using this dataset we need to decide that whether we should play or not on a particular day according to the weather conditions. So to solve this problem, we need to follow the below steps:

1. Convert the given dataset into frequency tables.
2. Generate Likelihood table by finding the probabilities of given features.
3. Now, use Bayes theorem to calculate the posterior probability.

Problem: If the weather is sunny, then the Player should play or not?

Solution: To solve this, first consider the below dataset:

	Outlook	Play
0	Rainy	Yes
1	Sunny	Yes
2	Overcast	Yes
3	Overcast	Yes
4	Sunny	No

5	Rainy	Yes
6	Sunny	Yes
7	Overcast	Yes
8	Rainy	No
9	Sunny	No
10	Sunny	Yes
11	Rainy	No
12	Overcast	Yes
13	Overcast	Yes

Frequency table for the Weather Conditions:

Weather	Yes	No
Overcast	5	0
Rainy	2	2
Sunny	3	2
Total	10	4

Likelihood table weather condition:

Weather	No	Yes	
Overcast	0	5	5/14= 0.35
Rainy	2	2	4/14=0.29

Sunny	2	3	5/14=0.35
All	4/14=0.29	10/14=0.71	

Applying Bayes' theorem:

P(Yes | Sunny)= P(Sunny | Yes)*P(Yes)/P(Sunny)

P(Sunny | Yes)= 3/10= 0.3

P(Sunny)= 0.35

P(Yes)=0.71

So P(Yes | Sunny) = 0.3*0.71/0.35= **0.60**

P(No | Sunny)= P(Sunny | No)*P(No)/P(Sunny)

P(Sunny | NO)= 2/4=0.5

P(No)= 0.29

P(Sunny)= 0.35

So P(No|Sunny)= 0.5*0.29/0.35 = **0.41**

So as we can see from the above calculation that **P(Yes|Sunny)>P(No|Sunny)**

Hence on a Sunny day, Player can play the game.

Advantages of Naïve Bayes Classifier:

- Naïve Bayes is one of the fast and easy ML algorithms to predict a class of datasets.
- It can be used for Binary as well as Multi-class Classifications.
- It performs well in Multi-class predictions as compared to the other Algorithms.
- It is the most popular choice for **text classification problems**.

Disadvantages of Naïve Bayes Classifier:

- Naive Bayes assumes that all features are independent or unrelated, so it cannot learn the relationship between features.

Applications of Naïve Bayes Classifier:

- It is used for **Credit Scoring**.
- It is used in **medical data classification**.
- t is used in Text classification such as **Spam filtering** and **Sentiment analysis**.

Types of Naïve Bayes Model:

There are three types of Naive Bayes Model, which are given below:

- **Gaussian**: The Gaussian model assumes that features follow a normal distribution. This means if predictors take continuous values instead of discrete, then the model assumes that these values are sampled from the Gaussian distribution.
- **Multinomial**: The Multinomial Naïve Bayes classifier is used when the data is multinomial distributed. It is primarily used for document classification problems, it means a particular document belongs to which category such as Sports, Politics, education, etc.
  The classifier uses the frequency of words for the predictors.
- **Bernoulli**: The Bernoulli classifier works similar to the Multinomial classifier, but the predictor variables are the independent Booleans variables. Such as if a particular word is present or not in a document. This model is also famous for document classification tasks.