Learning to Classify Text

6. Learning to Classify Text

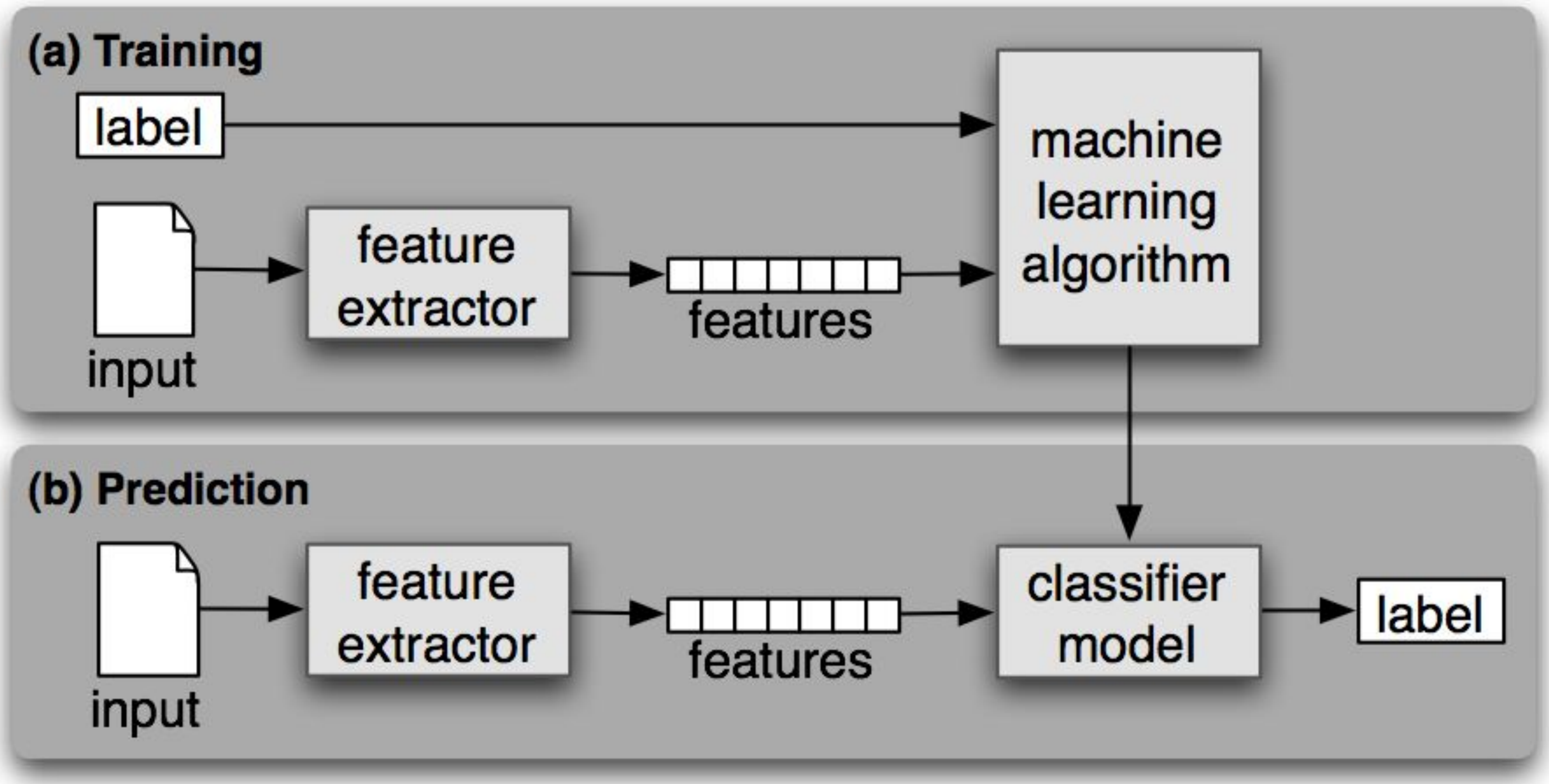
Goals

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?

- How can we identify particular features of language data that are salient for classifying it?
- How can we construct models of language that can be used to perform language processing tasks automatically?
- What can we learn about language from these models?

1 Supervised Classification

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


(a) Training
label
machine learning algorithm
input
feature extractor
features
(b) Prediction
input
feature extractor
features
classifier model
label

Gender Identification

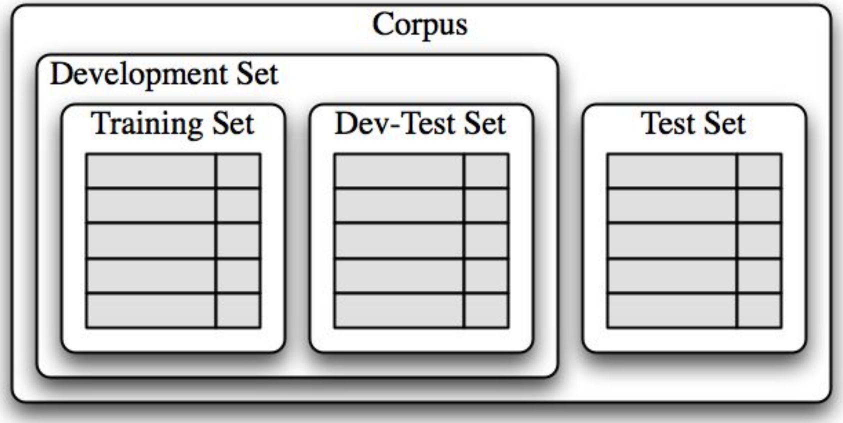
- Names ending in a, e and i are likely to be female, while names ending in k, o, r, s and t are likely to be male
- Decide what **features** of the input are relevant, and **how to encode** those features
- feature extractor function builds a dictionary containing relevant information about a given name:

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

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

>>> 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
```

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
- if you provide too many features, then the algorithm will have a higher chance of relying on idiosyncrasies of your training data (**overfitting**) and can be especially problematic when working with small training sets
- To refine the feature set, first, we select a development set, being subdivided into the training set and the dev-test set
- The training set is used to train the model, and the dev-test set is used to perform error analysis.

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



- we train a model using the training set, and then run it on the dev-test set
- Using the dev-test set, we can generate a list of the errors that the classifier makes when predicting name genders
- Looking through this list of errors makes it clear that some suffixes that are more than one letter can be indicative of name genders
    - names ending in yn are mostly 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

```
>>> 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))
0.75

>>> 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=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
```

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

Document Classification

- Documents labeled with categories
- the Movie Reviews Corpus categorizes each review as positive or negative
- For document topic identification, we define a feature extractor checks whether each of these words is present in a given document
- To check how reliable the resulting classifier is, we compute its accuracy on the test set and use show_most_informative_features() to find out which features the classifier found to be most informative

```
>>> 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)

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')))
{'contains(waste)': False, 'contains(lot)': False, ...}

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
```

Part-of-Speech Tagging

- To train a classifier to work out which suffixes are most informative
- First, find out what the most common suffixes are
- Define a feature extractor function which checks a given word for these suffixes
- Classifier makes its decisions based only on information about which of the common suffixes a given word has

```
>>> 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', ...]

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

- Train a new "decision tree" classifier
- Decision tree models are often fairly easy to interpret — we can even instruct NLTK to print them out as pseudocode:

```
>>> 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'

>>> 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
```

Exploiting Context

- Contextual features often provide powerful clues about the correct tag — for example, when tagging the word "fly," knowing that the previous word is "a" will allow us to determine that it is functioning as a noun, not a verb
- In order to accommodate features that depend on a word's context, we pass in a complete (untagged) sentence, along with the index of the target word

```
def pos_features(sentence, i):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]
    return features
>>> pos_features(brown.sents()[0], 8)
{'suffix(3)': 'ion', 'prev-word': 'an', 'suffix(2)': 'on', 'suffix(1)': 'n'}

>>> tagged_sents = brown.tagged_sents(categories='news')
>>> featuresets = []
>>> for tagged_sent in tagged_sents:
...     untagged_sent = nltk.tag.untag(tagged_sent)
...     for i, (word, tag) in enumerate(tagged_sent):
...         featuresets.append( (pos_features(untagged_sent, i),
tag) )

>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)

>>> nltk.classify.accuracy(classifier, test_set)
0.78915962207856782
```

Sequence Classification

- One sequence classification strategy, known as consecutive classification or greedy sequence classification finds the most likely class label for the first input, then to use that answer to help find the best label for the next input. The process can be repeated until all of the inputs have been labeled

```
>>> 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))
0.79796012981
```

```
def pos_features(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    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)
```

2. Further Examples of Supervised Classification

Sentence Segmentation

- Sentence segmentation can be viewed as a classification task for punctuation: whenever we encounter a symbol that can end a sentence, such as a period or a question mark, we decide whether it terminates the preceding sentence

```
>>> sents = nltk.corpus.treebank_raw.sents()
>>> tokens = []
>>> boundaries = set()
>>> offset = 0
>>> for sent in sents:
...     tokens.extend(sent)
...     offset += len(sent)
...     boundaries.add(offset-1)

>>> def punct_features(tokens, i):
...     return {'next-word-capitalized': tokens[i+1][0].isupper(),
...             'prev-word': tokens[i-1].lower(),
...             'punct': tokens[i],
...             'prev-word-is-one-char': len(tokens[i-1]) == 1}

>>> featuresets = [(punct_features(tokens, i), (i in
boundaries))
...               for i in range(1, len(tokens)-1)
...               if tokens[i] in '.?!']

>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
0.936026936026936
```

Identifying Dialogue Act Types

- Dialogue utterances as a type of action performed by the speaker
    - performative statements such as "I forgive you" or "I bet you can't climb that hill."
    - greetings, questions, answers, assertions, and clarifications
- Recognizing the dialogue acts can be an important first step in understanding the conversation
- NPS Chat Corpus consists of over 10,000 posts from instant messaging sessions, labeled with one of 15 dialogue act types, such as "Statement," "Emotion," "ynQuestion", and "Continuer."

```
>>> posts = nltk.corpus.nps_chat.xml_posts()[:10000]

>>> def dialogue_act_features(post):
...     features = {}
...     for word in nltk.word_tokenize(post):
...         features['contains({})'.format(word.lower())] =
True
...     return features

>>> featuresets = [(dialogue_act_features(post.text),
post.get('class'))
...                for post in posts]
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:],
featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print(nltk.classify.accuracy(classifier, test_set))
0.67
```

Recognizing Textual Entailment

- Recognizing textual entailment (RTE) is the task of determining whether a given piece of text T entails another text called the "hypothesis"
- our features count the degree of word overlap, and the degree to which there are words in the hypothesis but not in the text (captured by the method hyp_extra())
- features indicate that all important words in the hypothesis are contained in the text, and thus there is some evidence for labeling this as True

```
def rte_features(rtepair):
    extractor = nltk.RTEFeatureExtractor(rtepair)
    features = {}
    features['word_overlap'] = len(extractor.overlap('word'))
    features['word_hyp_extra'] = len(extractor.hyp_extra('word'))
    features['ne_overlap'] = len(extractor.overlap('ne'))
    features['ne_hyp_extra'] = len(extractor.hyp_extra('ne'))
    return features

>>> rtepair = nltk.corpus.rte.pairs(['rte3_dev.xml'])[33]
>>> extractor = nltk.RTEFeatureExtractor(rtepair)
>>> print(extractor.text_words)
{'Russia', 'Organisation', 'Shanghai', 'Asia', 'four', 'at',
'operation', 'SCO', ...}
>>> print(extractor.hyp_words)
{'member', 'SCO', 'China'}
>>> print(extractor.overlap('word'))
set()
>>> print(extractor.overlap('ne'))
{'SCO', 'China'}
>>> print(extractor.hyp_extra('word'))
{'member'}
```

3 Evaluation

- The Test Set
- Accuracy, Precision and Recall, F-score
- Confusion Matrices
- Cross-Validation

4-6 Classifiers

- **Decision trees** are automatically constructed tree-structured flowcharts that are used to assign labels to input values based on their features. Although they're easy to interpret, they are not very good at handling cases where feature values interact in determining the proper label
- In **naive Bayes classifiers**, each feature independently contributes to the decision of which label should be used. This allows feature values to interact, but can be problematic when two or more features are highly correlated with one another
- **Maximum Entropy classifiers** use a basic model that is similar to the model used by naive Bayes; however, they employ iterative optimization to find the set of feature weights that maximizes the probability of the training set

7 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
- 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.
- 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.

Summary

- Modeling the linguistic data found in corpora can help us to understand linguistic patterns, and can be used to make predictions about new language data
- Supervised classifiers use labeled training corpora to build models that predict the label of an input based on specific features of that input.
- Supervised classifiers can perform a wide variety of NLP tasks, including document classification, part-of-speech tagging, sentence segmentation, dialogue act type identification, and determining entailment relations, and many other tasks.
- When training a supervised classifier, you should split your corpus into three datasets: a training set for building the classifier model; a dev-test set for helping select and tune the model's features; and a test set for evaluating the final model's performance. it is important that you use fresh data for the test set. Otherwise, your evaluation results may be unrealistically optimistic.
- Most of the models that are automatically constructed from a corpus are descriptive — they let us know which features are relevant to a given patterns or construction, but they don't give any information about causal relationships between those features and patterns.

Bayes's Theorem

P(A│B)를 알고 있을때 P(B│A)를 구할 수 있다

P(A)는 사건 A가 일어날 확률
P(B)는 사건 B가 일어날 확률
P(A │ B)는 사건 A가 일어났을 때 사건 B가 일어날 확률
P(B │ A)는 사건 B가 일어났을 때 사건 A가 일어날 확률

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

예를 들어, 한 학교의 학생이 남학생인 확률이 P(A)라고 하고, 학생이 키가 170이 넘는 확률을 P(B)라고 했을때, 남학생 중에서, 키가 170이 넘는 확률은 B의 조건부 확률이 되며 P(B|A)로 표현 한다.

앞의 남학생인 확률 P(A)와 키가 170이상인 확률 P(B)를 알고, 남학생중에서 키가 170인 확률 P(B|A)를 알면, 키가 170인 학생중에, 남학생인 확률 P(A|B)를 알 수 있다

Naive Bayesian Classification

매개 변수, x,y가 있을때, 분류 1에 속할 확률이 p1(x,y)이고, 분류 2에 속할 확률이 p2(x,y)일때,

p1(x,y) > p2(x,y) 이면, 이 값은 분류 1에 속함
p1(x,y) < p2(x,y) 이면, 이 값은 분류 2에 속함

즉, 분류하는 대상의 **각 분류별 확률을 측정하여, 그 확률이 큰 쪽으로 분류.**

예를 들어, 이메일에 대해서 분류가 스팸과 스팸이 아닌 분류가 있을때, 이메일에 들어가 있는 단어들 w1,…,wn 매개 변수 (“쇼핑”,”비아그라”,”보험”,….)에 대해서, 해당 이메일이 스팸일 확률과 스팸이 아닌 확률을 측정하여, 확률이 높은 쪽으로 판단

Naive Bayesian Classification 예

영화	단어	분류
1	fun,couple,love,love	Comedy
2	fast,furious,shoot	Action
3	Couple,fly,fast,fun,fun	Comedy
4	Furious,shoot,shoot,fun	Action
5	Fly,fast,shoot,love	Action

어떤 문서에 “fun,furious,fast” 단어만 있을 때, 이 문서는 Comedy인가? Action인가 ?

영화가 Comedy일 확률: P(Comedy|Words) = P(Words|Comedy) X P(Comedy) / P(Words) -> A
영화가 Action 확률: P(Action|Words) = P(Words|Action) X P(Action) / P(Words) -> B
A > B라면 Comedy로 분류하고, A < B라면 Action으로 분류
A, B 확률을 구할때 분모에 P(Words) 인데 **A, B의 대소만 비교 하기 때문에 무시**
A = P(Words | Comedy) X P(Comedy)
B = P(Words | Action) X P(Action)

http://pigbrain.github.io/machinelearning/2015/07/09/NaiveBayesian_on_MachineLearning

영화	단어	분류
1	fun,couple,love,love	Comedy
2	fast,furious,shoot	Action
3	Couple,fly,fast,fun,fun	Comedy
4	Furious,shoot,shoot,fun	Action
5	Fly,fast,shoot,love	Action

- 각 단어의 빈도 수
  Count (fast,comedy) = 1
  Count(furious,comedy) = 0
  Count(fun,comedy) = 3
  Count(fast,action)= 2
  Count(furious,action)=2
  Count(fun,action) = 1
- P(Words|Comedy)는 P(fast,furious,fun|Comedy)로 표현 가능
  P(fast | Comedy) * P(furious | Comedy) * P(fun | Comedy)로 표현 가능
  Comedy 영화에 나오는 총 단어의 개수 : 9
  P(fast ｜ Comedy) * P(furious ｜ Comedy) * P(fun ｜ Comedy) = (1/9) * (0/9) * (3/9)
  전체 영화 5편중에서 2편이 Comedy이기때문에, P(Comedy) = 2/5
- A = P(Comedy|Words) = ((1/9) * (0/9) * (3/9)) * 2/5 = 0
- B = P(Action|Words) = ((2/11) * (2/11)*(1/11)) * 3/5 = 0.0018
- A < B 이므로 Action으로 분류

Laplace Smoothing을 이용한 예외 처리

나이브베이시안은 Training Data에 없는 새로운 단어가 나왔을 때 확률이 0이 되는 문제
위 현상을 방지하기 위해 도수에 +1을 해줌으로써 확률이 0이 되는 것을 방지

$$\hat{P}(x|c)=\frac{count(x,c)+1}{\Sigma_{x\in V}(count(x,c)+1)}=\frac{count(x,c)+1}{(\Sigma_{x\in V}count(x,c))+|V|}$$

|V|는 전체 단어의 수가 아니라 유일한 단어의 수-7. Laplace Smoothing을 적용하여 다시 계산한 각각의

$$P(comedy|d)=P(fast|comedy)\bullet P(furious|comedy)\bullet P(fun|comedy)\bullet P(comedy)$$
$$=\frac{1+1}{9+7}\bullet\frac{0+1}{9+7}\bullet\frac{3+1}{9+7}\bullet\frac{2}{5}$$
$$=\frac{2}{16}\bullet\frac{1}{16}\bullet\frac{4}{16}\bullet\frac{2}{5}$$
$$=0.00078$$

$$P(action|d)=P(fast|action)\bullet P(furious|action)\bullet P(fun|action)\bullet P(action)$$
$$=\frac{2+1}{11+7}\bullet\frac{2+1}{11+7}\bullet\frac{1+1}{11+7}\bullet\frac{3}{5}$$
$$=\frac{3}{18}\bullet\frac{3}{18}\bullet\frac{2}{18}\bullet\frac{3}{5}$$
$$=0.0018$$

action 확률이 높기 때문에 action으로 분류

http://unlimitedpower.tistory.com/entry/NLP-Naive-Bayesian-Classification나이브-베이즈-분류?category=407624

Log를 이용한 언더 플로우 방지

P(words|comedy)나 P(words|action)은 각 단어의 확률의 곱인데, 항목이 많은 경우 소숫점 아래로 계속 내려가서, 구분이 어려울 정도까지 값이 작게 나올 수 있다. 이를 해결 하기 위해서 로그 (log)를 사용.

log(a*b) = log (a) + log(b)와 같기 때문에,
P(comedy|words) = P(words|comedy)*P(comedy) 양쪽 공식에 모두 Log를
Log(P(comedy|words)) = Log(P(words|comedy)*P(comedy))

Log(P(words|comedy)*P(comedy))
= Log(P(fun|comedy)*P(furios|comedy)*…*P(Comedy))
= log(P(fun|comedy))+log(P(furios|comedy)+…+log(P(Comedy))

http://bcho.tistory.com/1010

Spam filtering

메일 제목으로 [광고, 중요] 카테고리 분류

E.g.,

영화평의 긍정 부정
https://ratsgo.github.io/machine%20learning/2017/05/18/naive/

```
from bayes import BayesianFilter
bf = BayesianFilter()
# 텍스트 학습
bf.fit("파격 세일 - 오늘까지만 30% 할인", "광고")
bf.fit("쿠폰 선물 & 무료 배송", "광고")
bf.fit("현데계 백화점 세일", "광고")
bf.fit("봄과 함께 찾아온 따뜻한 신제품 소식", "광고")
bf.fit("인기 제품 기간 한정 세일", "광고")
bf.fit("오늘 일정 확인", "중요")
bf.fit("프로젝트 진행 상황 보고","중요")
bf.fit("계약 잘 부탁드립니다","중요")
bf.fit("회의 일정이 등록되었습니다.","중요")
bf.fit("오늘 일정이 없습니다.","중요")
# 예측
pre, scorelist = bf.predict("재고 정리 할인, 무료 배송")
print("결과 =", pre)
print(scorelist)
```

Spam filtering: 전처리

1. 형태소 분석하여 조사, 어미, 구두점 제외
2. 단어와 카테고리 빈도 분석

```
import math, sys
from konlpy.tag import Twitter
class BayesianFilter:
    def __init__(self):
        self.words = set() # 출현한 단어 기록
        self.word_dict = {} # 카테고리마다의 출현 횟수 기록
        self.category_dict = {} # 카테고리 출현 횟수 기록
    # 형태소 분석하기 --- (※1)
    def split(self, text):
        results = []
        twitter = Twitter()
        # 단어의 기본형 사용
        malist = twitter.pos(text, norm=True, stem=True)
        for word in malist:
            # 어미/조사/구두점 등은 대상에서 제외
            if not word[1] in ["Josa", "Eomi", "Punctuation"]:
                results.append(word[0])
        return results
    # 단어와 카테고리의 출현 횟수 세기 --- (※2)
    def inc_word(self, word, category):
        # 단어를 카테고리에 추가하기
        if not category in self.word_dict:
            self.word_dict[category] = {}
        if not word in self.word_dict[category]:
            self.word_dict[category][word] = 0
        self.word_dict[category][word] += 1
        self.words.add(word)
    def inc_category(self, category):
        # 카테고리 계산하기
        if not category in self.category_dict:
            self.category_dict[category] = 0
        self.category_dict[category] += 1
```

Spam filtering

3. 텍스트 학습: 형태소 분할, 카테고리와 단어 연결
4. 단어 리스트를 주면 단어가 카테고리에 속할 점수의 합 계산. 입력 문서의 단어의 개수가 많아지다보면 확률값이 너무 작아지고 낮은 확률 값들의 곱으로 인해 상당히 낮은 숫자가 나타날 수 있어 비교가 잘 안되기도 하므로, log를 사용하여 확률 값들의 곱에서 합으로 변형하여 사용
5. 주어진 텍스트의 카테고리 점수를 계산하고 가장 높은 카테고리 리턴
6. 사전에 없는 단어가 나오면 확률이 0이 되므로 1을 더함

```
# 카테고리 내부의 단어 출현 횟수 구하기
    def get_word_count(self, word, category):
        if word in self.word_dict[category]:
            return self.word_dict[category][word]
        else:
            return 0
    # 카테고리 계산
    def category_prob(self, category):
        sum_categories = sum(self.category_dict.values())
        category_v = self.category_dict[category]
```

```
# 텍스트 학습하기 --- (※3)
def fit(self, text, category):
    word_list = self.split(text)
    for word in word_list:
        self.inc_word(word, category)
    self.inc_category(category)

# 단어 리스트에 점수 매기기--- (※4)
def score(self, words, category):
    score = math.log(self.category_prob(category))
    for word in words:
        score += math.log(self.word_prob(word, category))
    return score

# 예측하기 --- (※5)
def predict(self, text):
    best_category = None
    max_score = -sys.maxsize
    words = self.split(text)
    score_list = []
    for category in self.category_dict.keys():
        score = self.score(words, category)
        score_list.append((category, score))
        if score > max_score:
            max_score = score
            best_category = category
    return best_category, score_list

# 카테고리 내부의 단어 출현 비율 계산 --- (※6)
def word_prob(self, word, category):
    n = self.get_word_count(word, category) + 1 # ---(※6a)
    d = sum(self.word_dict[category].values()) + len(self.words)
    return n / d
```

Lvenshtein distance (edit distance) 문장 유사도 계산

- ‘가나다라'와 ‘가마바라'간의 거리는 ‘가나다라'를 ‘가마바라' 로 만들때 필요한 문자열 조작의 수: 2 (‘나'->’마', ‘다’->’바')

```
print(calc_distance("가나다라","가마바라"))
# 실행 예
samples = ["신촌역","신천군","신천역","신발","마곡역"]
base = samples[0]
r = sorted(samples, key = lambda n: calc_distance(base, n))
for n in r:
    print(calc_distance(base, n), n)
```

```
def calc_distance(a, b):
    if a == b: return 0
    a_len = len(a)
    b_len = len(b)
    if a == "": return b_len
    if b == "": return a_len
    # 2차원 표 (a_len+1, b_len+1) 준비하기 --- (※1)
    matrix = [[] for i in range(a_len+1)]
    for i in range(a_len+1): # 0으로 초기화
        matrix[i] = [0 for j in range(b_len+1)]
    # 0일 때 초깃값을 설정
    for i in range(a_len+1):
        matrix[i][0] = i
    for j in range(b_len+1):
        matrix[0][j] = j
    # 표 채우기 --- (※2)
    for i in range(1, a_len+1):
        ac = a[i-1]
        for j in range(1, b_len+1):
            bc = b[j-1]
            cost = 0 if (ac == bc) else 1
            matrix[i][j] = min([
                matrix[i-1][j] + 1,     # 문자 삽입
                matrix[i][j-1] + 1,     # 문자 제거
                matrix[i-1][j-1] + cost # 문자 변경
            ])
    return matrix[a_len][b_len]
```

N-gram 유사도

- n-gram의 같은 단어가 두 문장에 존재하는 비율
- 표절 등을 확인

```
a = "오늘 강남에서 맛있는 스파게티를 먹었다."
b = "강남에서 먹었던 오늘의 스파게티는 맛있었다."
# 2-gram
r2, word2 = diff_ngram(a, b, 2)
print("2-gram:", r2, word2)
# 3-gram
r3, word3  = diff_ngram(a, b, 3)
print("3-gram:", r3, word3)
```

```
def ngram(s, num):
    res = []
    slen = len(s) - num + 1
    for i in range(slen):
        ss = s[i:i+num]
        res.append(ss)
    return res
def diff_ngram(sa, sb, num):
    a = ngram(sa, num)
    b = ngram(sb, num)
    r = []
    cnt = 0
    for i in a:
        for j in b:
            if i == j:
                cnt += 1
                r.append(i)
    return cnt / len(a), r
```