Northeastern University

Naive Bayes

Roi Yehoshua

Agenda

- Naive Bayes classifiers
- Laplace smoothing
- Text analysis and preprocessing
- The NLTK and spaCy libraries
- Document classification

The Naive Bayes Model

- Given a sample (**x**, *y*), compute the **class posterior probability** using Bayes' rule:

$$P(y = k|\mathbf{x}) = \frac{P(\mathbf{x}|y = k)P(y = k)}{P(\mathbf{x})}$$

- The **class prior probability** can be estimated from the frequency of class *k* in the data

$$P(y = k) = \frac{n_k}{n}$$

n_k: number of samples of class *k* in the training set

n: total number of training samples

- *P*(**x**) is a normalizing factor: $P(\mathbf{x}) = \sum_{k=1}^{K} P(\mathbf{x}|y = k)P(y = k)$

- How to estimate *P*(**x** | *y* = *k*) from the data?
  - Problem: Need to estimate an exponential number of probabilities!
  - e.g., if there are *m* binary features, we need to estimate $2^m K$ probabilities

Naive Bayes Classification

- **Naive Bayes assumption**: the features are conditionally independent given the class

$$P(\mathbf{x}|y=k) = P(x_1|y=k) \cdot P(x_2|y=k) \cdots P(x_m|y=k) = \prod_{j=1}^{m} P(x_j|y=k)$$

- Therefore we can write the posterior probabilities of the classes as:

$$P(y=k|\mathbf{x}) = \frac{P(y=k)\prod_{j=1}^{m} P(x_j|y=k)}{P(\mathbf{x})}$$

  - Now we only need to estimate *m* parameters instead of 2^m for each class

- If we only care about the labels (and not the probabilities):

$$\hat{y} = \underset{k}{\operatorname{argmax}}\, P(y=k)\prod_{j=1}^{m} P(x_j|y=k)$$

- The different NB classifiers differ by the assumption they make on $P(x_j|y)$

Bernoulli Naive Bayes

- Assumption: each feature is distributed according to a Bernoulli distribution
- Example: in text classification, each feature x_j may represent the occurrence or absence of the *j*th word from the vocabulary in the text
- The parameters of the model are estimated using relative frequency counting
  - For each feature *j* and class *k*:

$$P(x_j = 1|y = k) = \frac{n_{jk}}{n_k}$$

number of times feature *j* appears in samples of class *k*

number of samples in class *k*

Categorical Naive Bayes

- Assumption: each feature has a categorical distribution
  - Generalization of Bernoulli distribution to *m* possible categories
- The parameters of the model are:
  - For each feature *j*, class *k* and category *v*

$$P(x_j = v|y = k) = \frac{n_{jvk}}{n_k}$$

number of times feature *j* has the value *v* in samples of class *k*

number of samples in class *k*

Categorical Naive Bayes Example

- We'd like to decide whether to go out play tennis based on weather conditions

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Given a new sample:
x = (Outlook = sunny, Temperature = cool, Humidity = High, Wind = strong)
Predict whether to play tennis or not.

Categorical Naive Bayes Example

- We first estimate the class prior probabilities based on their frequency in the data:

$$P(\text{PlayTennis} = \text{Yes}) = 9/14 = 0.643$$
$$P(\text{PlayTennis} = \text{No}) = 5/14 = 0.357$$

- We now estimate the conditional probabilities of the features in the new sample:

$$P(\text{Outlook} = \text{Sunny}|\text{PlayTennis} = \text{Yes}) = 2/9 = 0.222$$
$$P(\text{Outlook} = \text{Sunny}|\text{PlayTennis} = \text{No}) = 3/5 = 0.6$$
$$P(\text{Temperature} = \text{Cool}|\text{PlayTennis} = \text{Yes}) = 3/9 = 0.333$$
$$P(\text{Temperature} = \text{Cool}|\text{PlayTennis} = \text{No}) = 1/5 = 0.2$$
$$P(\text{Humidity} = \text{High}|\text{PlayTennis} = \text{Yes}) = 3/9 = 0.333$$
$$P(\text{Humidity} = \text{High}|\text{PlayTennis} = \text{No}) = 4/5 = 0.8$$
$$P(\text{Wind} = \text{Strong}|\text{PlayTennis} = \text{Yes}) = 3/9 = 0.333$$
$$P(\text{Wind} = \text{Strong}|\text{PlayTennis} = \text{No}) = 3/5 = 0.6$$

Categorical Naive Bayes Example

- Therefore, the class posterior probabilities are:

$$P(\text{Yes}|\mathbf{x}) = \alpha P(\text{Yes})P(\text{O} = \text{Sunny}|\text{Yes})P(\text{T} = \text{Cool}|\text{Yes})P(\text{H} = \text{High}|\text{Yes})P(\text{W} = \text{Strong}|\text{Yes})$$
$$= \alpha 0.643 \cdot 0.222 \cdot 0.333 \cdot 0.333 \cdot 0.333 = 0.0053\alpha$$

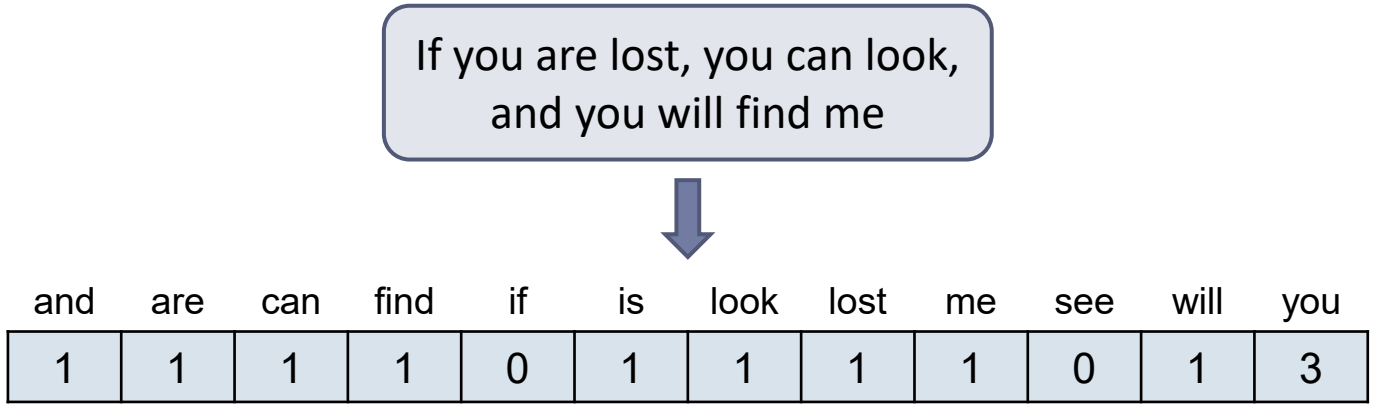
$$P(\text{No}|\mathbf{x}) = \alpha P(\text{No})P(\text{O} = \text{Sunny}|\text{No})P(\text{T} = \text{Cool}|\text{No})P(\text{H} = \text{High}|\text{No})P(\text{W} = \text{Strong}|\text{No})$$
$$= \alpha 0.357 \cdot 0.6 \cdot 0.2 \cdot 0.8 \cdot 0.6 = 0.0206\alpha$$

  - $\alpha = 1/P(\mathbf{x})$ is a constant term

- Since $P(\text{No}|\mathbf{x}) > P(\text{Yes}|\mathbf{x})$, the sample is classified as PlayTennis = No

Multinomial Naive Bayes

- Assumption:
  - There is only one categorical feature *x* that can take one of *m* categories
  - Each feature vector (x_1, …, x_m) is a histogram, where x_j counts the number of times *x* had the value *j* in that particular instance
- The multinomial distribution is a generalization of the binomial distribution to more than two possible outcomes in each trial
- Example: a bag-of-words model in text classification



Multinomial Naïve Bayes

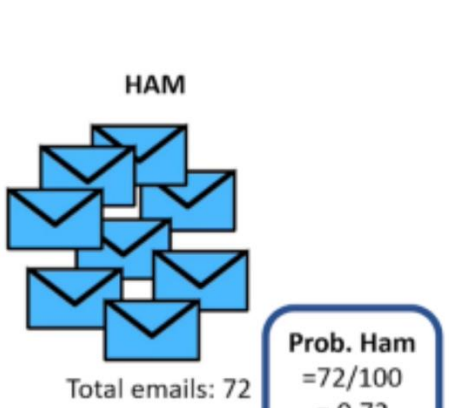
- The parameters of the model are:
  - For each class *k* and category *v*

$$P(x = v|y = k) = \frac{n_{vk}}{n_k}$$

number of times *x* got the value *v* (e.g., the word v appeared in the text) in samples of class *k*

total number of samples (e.g., words) in class *k*

Example: Spam Filter



Prob. Ham
=72/100
= 0.72

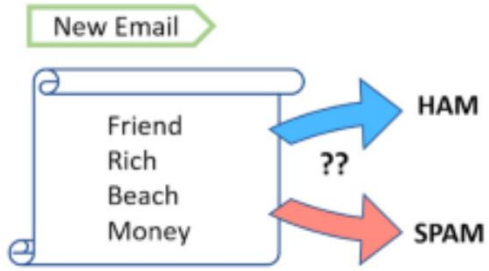
Prob. Spam
=28/100
= 0.28

HAM Total Words **263**

Word	Count	Prob of Word if mail is Ham
Friend	86	0.326996
Rich	3	0.011407
Money	9	0.034221
Beach	90	0.342205
Office	75	0.285171

SPAM Total Words **243**

Word	Count	Prob of Word if mail is Spam
Friend	59	0.242798
Rich	63	0.259259
Money	97	0.399177
Beach	24	0.098765
Office	**0**	**0**



Probability new email is Ham

$P(H) \times P(Friend|H) \times P(Rich|H) \times P(Beach|H) \times P(Money|H)$

$0.72 \times 0.34 \times 0.02 \times 0.38 \times 0.10 =$ **0.00094**

Probability new email is Spam

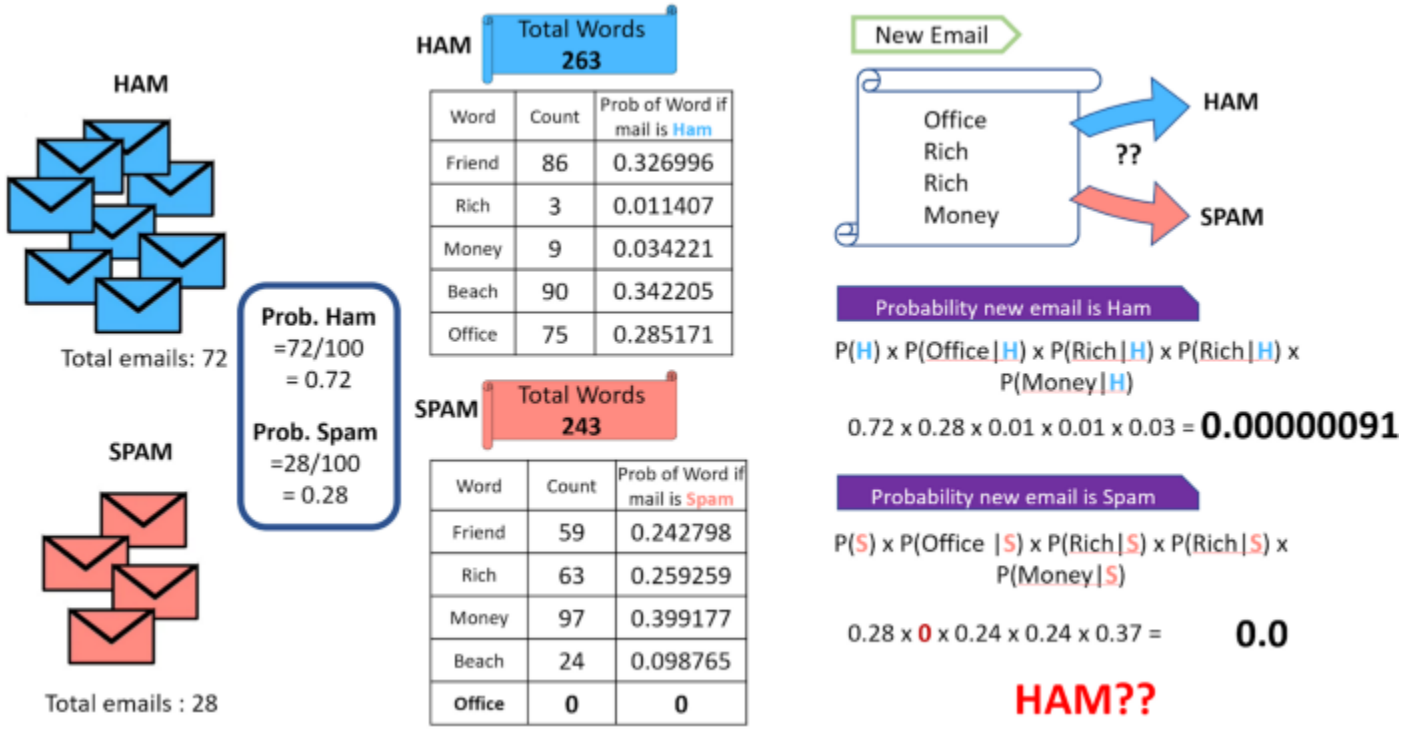
$P(S) \times P(Friend|S) \times P(Rich|S) \times P(Beach|S) \times P(Money|S)$

$0.28 \times 0.04 \times 0.67 \times 0.11 \times 0.13 =$ **0.00001**

HAM!!!

Source: https://www.atoti.io/how-to-solve-the-zero-frequency-problem-in-naive-bayes/

The Zero Frequency Problem



Laplace Smoothing

- Add a small sample correction in all the probability estimates, such that no probability is never set to exactly zero
- In categorical Naive Bayes:

$$P(x_j = v|y = k) = \frac{n_{jvk} + \alpha}{n_k + \alpha n_j}$$

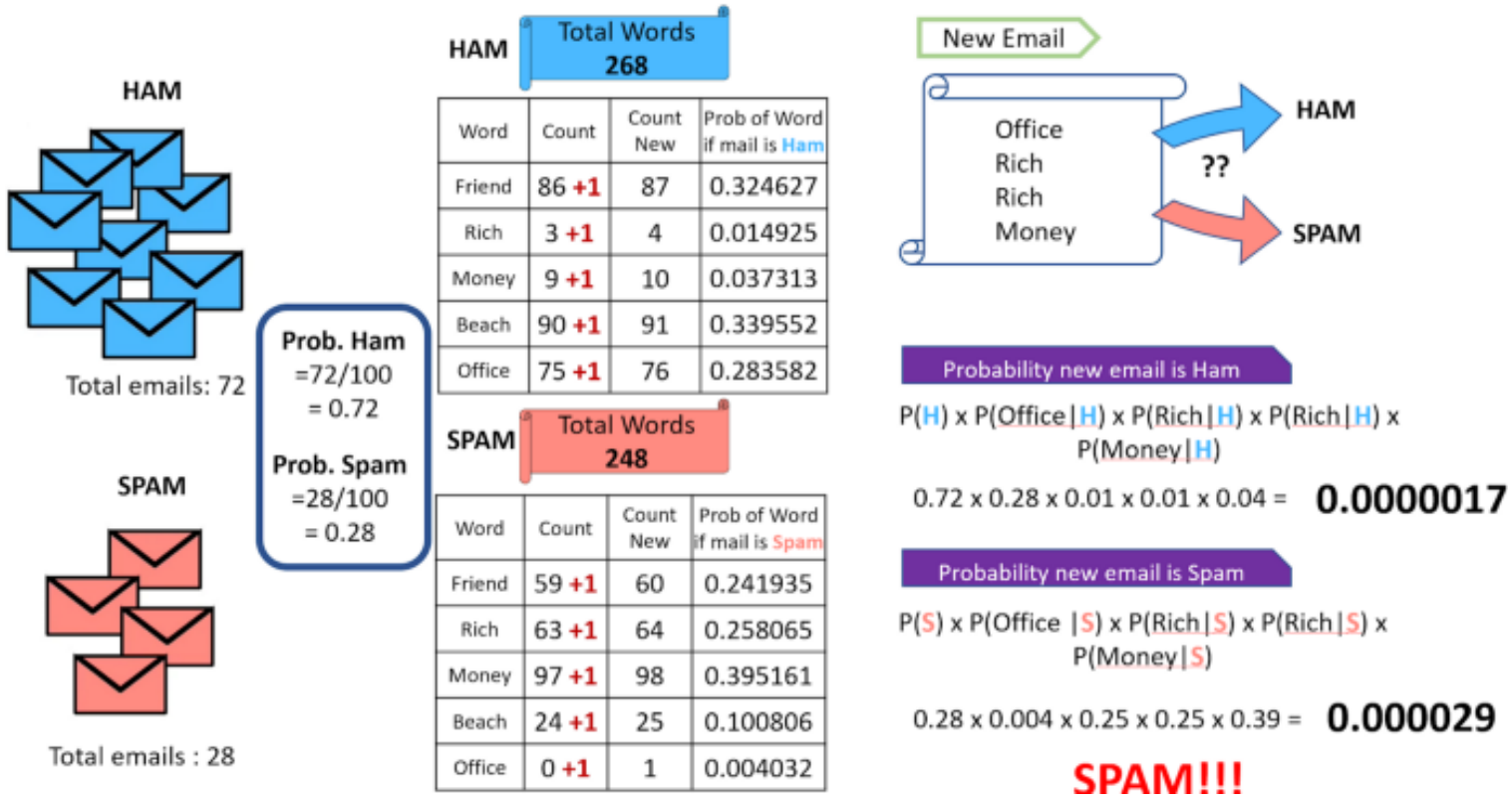
- In multinomial Naive Bayes:

$$P(x = v|y = k) = \frac{n_{vk} + \alpha}{n_k + \alpha n}$$

- α is a smoothing parameter
  - Setting $\alpha = 1$ is called **Laplace smoothing** (the most common one
  - $\alpha < 1$ is called Lidstone smoothing

Example: Laplace Smoothing

- Solving the zero frequency problem



Naïve Bayes in Scikit-Learn

- The module sklearn.naive_bayes contains implementation for all three models:

class `sklearn.naive_bayes.` **`BernoulliNB`**(*, *alpha=1.0*, *binarize=0.0*, *fit_prior=True*, *class_prior=None*) [source]

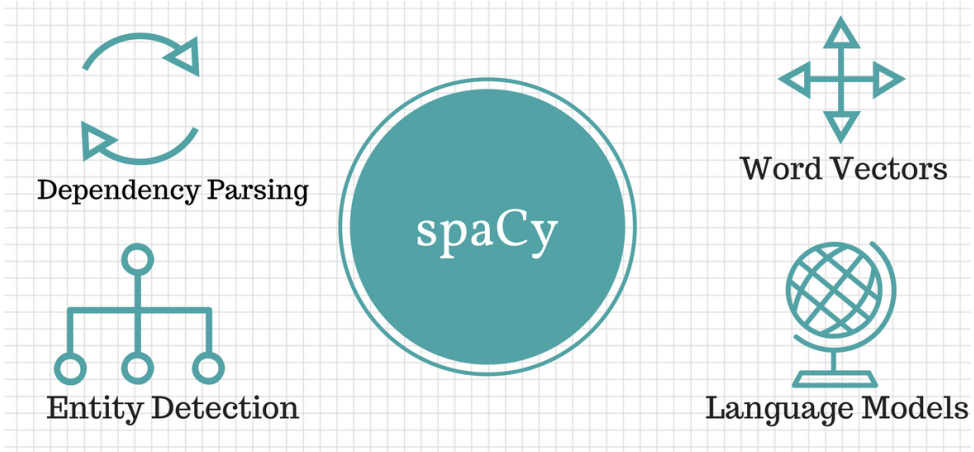
class `sklearn.naive_bayes.` **`CategoricalNB`**(*, *alpha=1.0*, *fit_prior=True*, *class_prior=None*, *min_categories=None*) [source]

class `sklearn.naive_bayes.` **`MultinomialNB`**(*, *alpha=1.0*, *fit_prior=True*, *class_prior=None*) [source]

- The argument alpha specifies the Laplace smoothing parameter
  - Use 0 for no smoothing

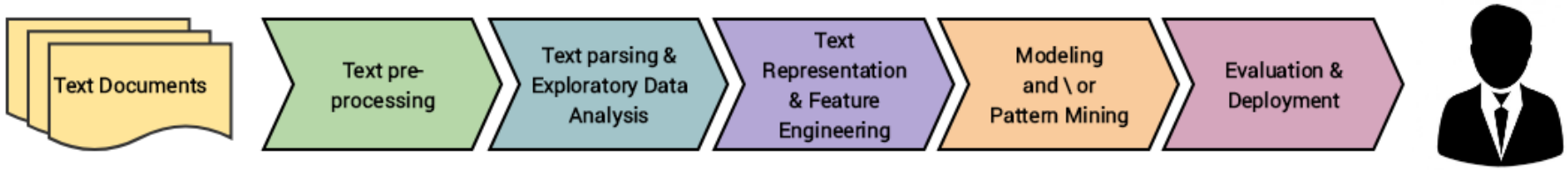
Python Libraries for Text Analysis

- Many of the basic text analysis operations can be done in Scikit-Learn (e.g., tokenization, removing stop words, TF-IDF vectorization)
- For more advanced operations (e.g., lemmatization, parsing, word embeddings) you can use the following Python libraries:
  - NLTK (Natural Language Toolkit) https://www.nltk.org/
  - spaCy https://spacy.io/
- Both can be installed via pip



Python Libraries for NLP

- Provide a set of tools for processing text and solving common NLP tasks
  - e.g., tokenization, parsing, part-of-speech tagging, NER, etc.
- Popular libraries
  - NLTK (Natural Language Toolkit)
  - spaCy
  - HuggingFace Transformers
- A typical NLP application workflow



NLTK (Natural Language Toolkit)

- A popular open-source Python library for NLP tasks
  - Including tokenization, stemming, tagging, parsing and semantic reasoning
- Provides 50 text corpora and lexical resources such as WordNet
- Home page: https://www.nltk.org/
- Can be easily installed via pip

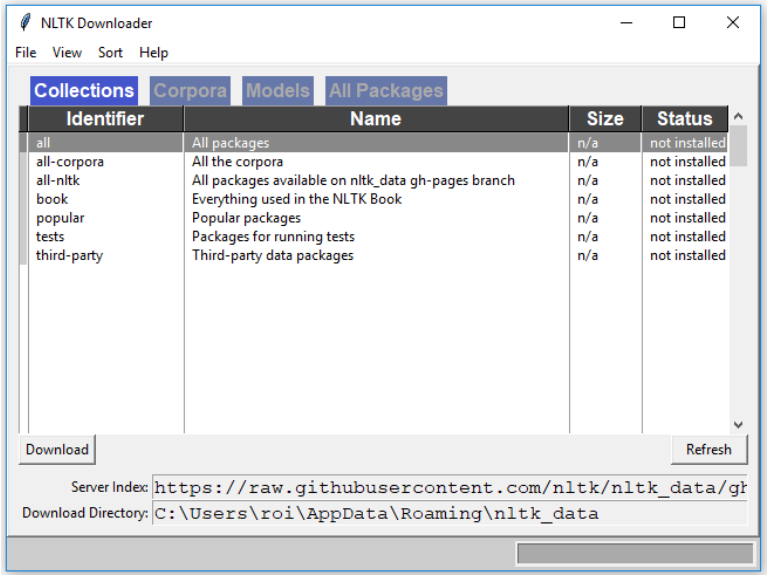
```
pip install nltk
```

Installing NLTK Data

- To download the NLTK data, type in a Jupyter Notebook (or Python interpreter):

```
import nltk
nltk.download()
```

- A new window should open, showing the NLTK downloader
  - Change the directory to C:\nltk_data (Windows) or /usr/local/share/nltk_data (Mac)



NLTK Corpora

- The corpora can be read using **nltk.corpus**
- The **raw()** method of the corpus returns the raw text of the entire corpus

```
from nltk.corpus import gutenberg

gutenberg.raw()
```

```
'[Emma by Jane Austen 1816]\n\nVOLUME I\n\nCHAPTER I\n\n\nEmma Woodhouse, handsome, clever, and
rich, with a comfortable home\nand happy disposition, seemed to unite some of the best blessings
\nof existence; and had lived nearly twenty-one years in the world\nwith very little to distress
or vex her.\n\nShe was the youngest of the two daughters of a most affectionate,\nindulgent fath
er; and had, in consequence of her sister\'s marriage,\nbeen mistress of his house from a very e
```

- The **word()** method returns the text as a list of words

```
gutenberg.words()
```

```
['[', 'Emma', 'by', 'Jane', 'Austen', '1816', ']', ...]
```

```
len(gutenberg.words())
```

```
2621613
```

NLTK Corpora

- The **fileids()** method of the corpus returns a list of the corpus files

```
print(gutenberg.fileids())
```

```
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt', 'blake-poems.tx
t', 'bryant-stories.txt', 'burgess-busterbrown.txt', 'carroll-alice.txt', 'chesterton-ball.txt',
'chesterton-brown.txt', 'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.tx
t', 'milton-paradise.txt', 'shakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbet
h.txt', 'whitman-leaves.txt']
```

- You can use the **raw()** method to get the text of a specific file

```
gutenberg.raw('carroll-alice.txt')
```

```
'[Alice\'s Adventures in Wonderland by Lewis Carroll 1865]\n\nCHAPTER I. Down the Rabbit-Hole\n
\nAlice was beginning to get very tired of sitting by her sister on the\nbank, and of having not
hing to do: once or twice she had peeped into the\nbook her sister was reading, but it had no pi
ctures or conversations in\nit, \'and what is the use of a book,\' thought Alice \'without pictu
res or\nconversation?\'\n\nSo she was considering in her own mind (as well as she could, for the
```

Word Lists and Lexicons

- The NLTK data package also includes a number of lexicons and word lists
- For example, to get all the words in the English language:

```
from nltk.corpus import words

words.words('en')
```

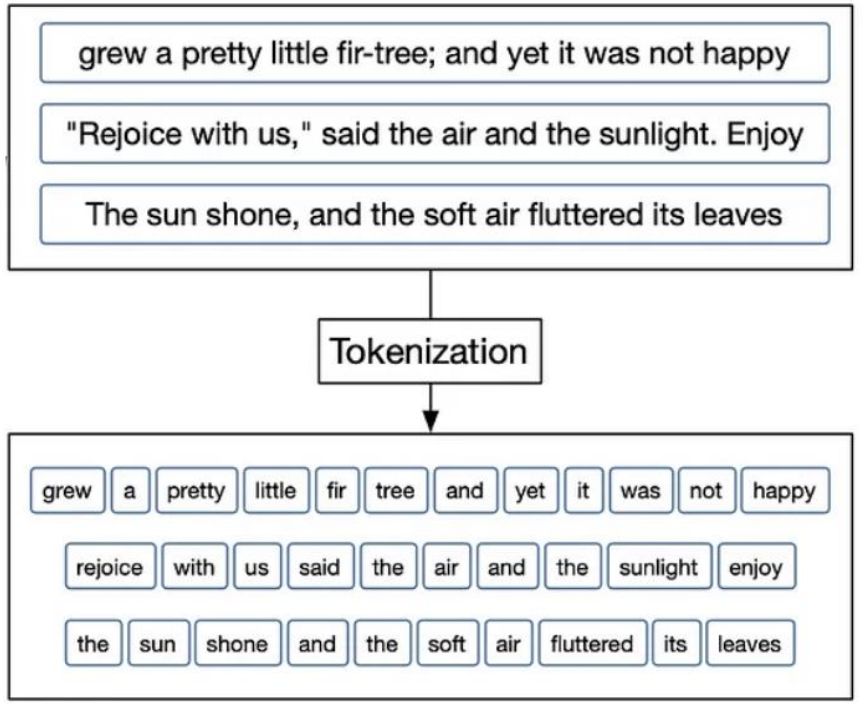
```
['A',
 'a',
 'aa',
 'aal',
 'aalii',
 'aam',
 'Aani',
 'aardvark',
 'aardwolf',
 'Aaron',
 'Aaronic',
```

Text Pre-Processing

- Before using text for NLP tasks we need to clean and normalize it
- This usually includes the following steps:
  - Text cleaning
    - Remove non-alphanumeric characters (e.g., punctuation marks or math symbols)
    - Remove non-relevant text (e.g., HTML tags in web data)
  - Lowercasing
    - Convert all characters to lowercase to ensure uniformity (e.g., “The”, “THE”, “the” -> “the”)
  - Tokenization: divide the text into tokens
  - Removing stopwords (commonly used words, such as “a”, “the”)
  - Stemming and lemmatization
    - Reduce words to their root or base form

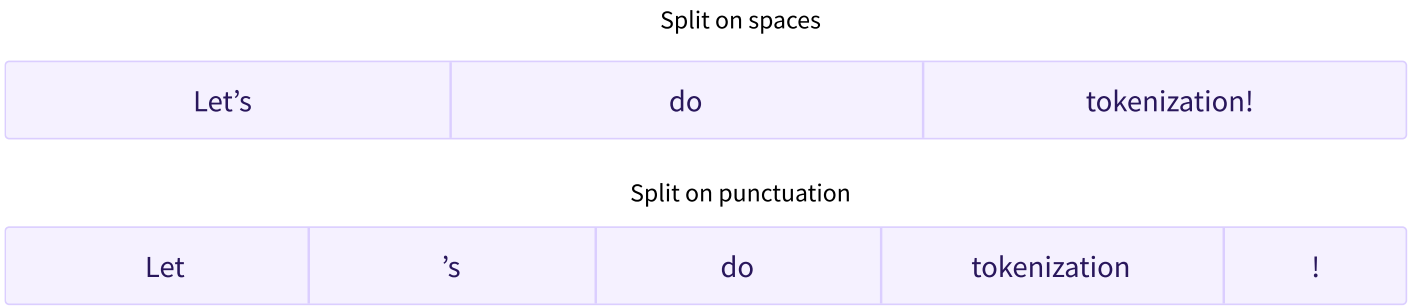
Tokenization

- Splitting a text into a sequence of tokens
- Three main approaches
  - Word-level tokenization
  - Character-level tokenization
  - Subword-level tokenization



Word-Level Tokenization

- Split the text into words



- Pros: Aligned with human’s language cognition
- Drawbacks:
  - Generates a large word vocabulary with many low-frequency words
    - e.g., there are over 500,000 words in the English language
  - Different forms of the same word will have different tokens, e.g., “dog” and “dogs”
  - Out-of-vocabulary issue: words not in the training corpus will not be represented
    - Need a special token to represent these words, e.g., [UNK]
  - Can yield different segmentation results in different languages (e.g., Chinese words)

Character-Level Tokenization

- Split the text into characters

- Benefits
  - Vocabulary is much smaller
  - There are much fewer out-of-vocabulary (unknown) tokens
- Drawbacks
  - The tokenization will produce a huge amount of tokens
  - The tokens are less meaningful

Subword-Level Tokenization

- Idea: rare words should be decomposed into meaningful subwords

Let's </w>	do</w>	token	ization</w>	!</w>

- Benefits
  - Good coverage with small vocabularies
  - Close to no unknown tokens
- Different approaches used in LLMs
  - Byte-Pair Encoding (BPE), used in GPT
  - WordPiece tokenization, used in BERT
  - Unigram tokenization, used in multilingual models

Tokenization in NLTK

- The function word_tokenize() performs word-level tokenization
  - Using NLTK's recommended tokenizer

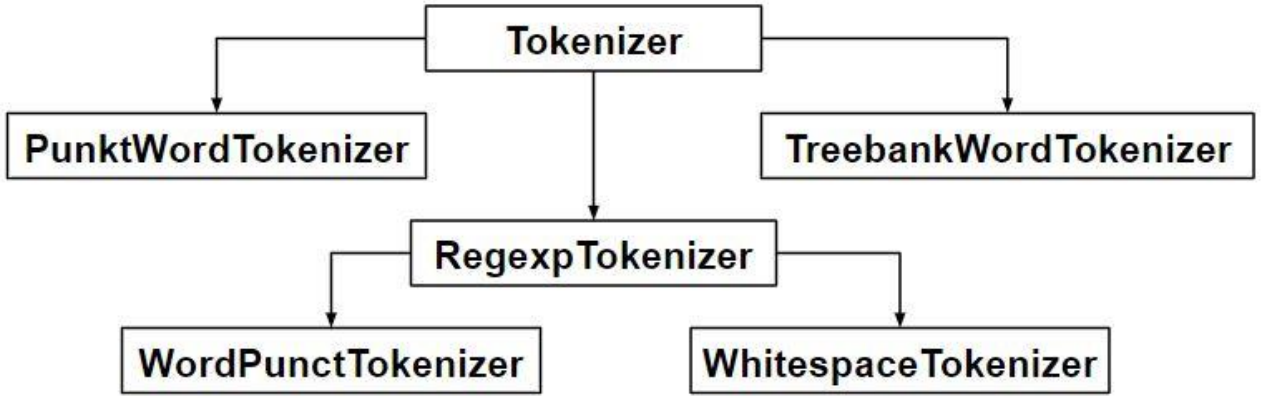
```
import nltk
```

```
text = "Apple is looking at buying a U.K. startup for $1 billion"

tokens = nltk.word_tokenize(text)
print(tokens)
```

```
['Apple', 'is', 'looking', 'at', 'buying', 'a', 'U.K.', 'startup', 'for', '$', '1', 'billion']
```

Tokenization in NLTK

- NLTK provides various tokenizer objects in the nltk.tokenize package



- For example, using a RegexpTokenizer to get only alphanumeric tokens:

```
from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r'\w+')
tokens = tokenizer.tokenize(text)
print(tokens)
```

```
['Apple', 'is', 'looking', 'at', 'buying', 'a', 'U', 'K', 'startup', 'for', '1', 'billion']
```

N-Grams

- **nltk.util** provides functions for computing *n*-grams (sequences of contiguous tokens)
  - bigrams(), trigrams(), ngrams()

```
import nltk
from nltk.util import ngrams
```

```
text = "Apple is looking at buying a U.K. startup for $1 billion"

tokens = nltk.word_tokenize(text)
trigrams = ngrams(tokens, n=3)
list(trigrams)
```

```
[('Apple', 'is', 'looking'),
 ('is', 'looking', 'at'),
 ('looking', 'at', 'buying'),
 ('at', 'buying', 'a'),
 ('buying', 'a', 'U.K.'),
 ('a', 'U.K.', 'startup'),
 ('U.K.', 'startup', 'for'),
 ('startup', 'for', '$'),
 ('for', '$', '1'),
 ('$', '1', 'billion')]
```

Stop Words

- Stop words are commonly used words in a language, such as “a”, “the”, “is”, “on”
- They are often filtered out since they carry little semantic value
- We can use the stopwords list in NLTK to filter out these words:

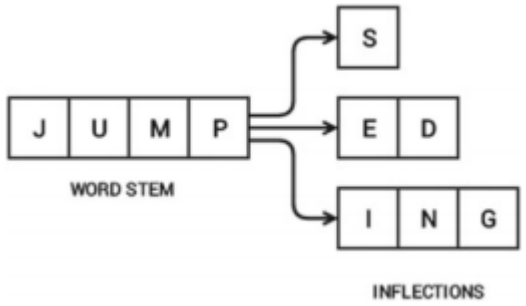
```
import nltk
from nltk.corpus import stopwords
```

```
text = "Apple is looking at buying a U.K. startup for $1 billion"

tokens = nltk.word_tokenize(text)
stop_words = stopwords.words('english')
filtered_tokens = [token for token in tokens if token not in stop_words]
filtered_tokens
```

```
['Apple', 'looking', 'buying', 'U.K.', 'startup', '$', '1', 'billion']
```

Stemming

- **Word stem** is the base form (or root form) of a word
- New words are created by attaching affixes to a stem in a process known as **inflection**
  - The inflection of verbs is called **conjugation**



- **Stemming** removes morphological affixes from words, leaving only the word stem

The boy's cars have different colors	→	the boy car have differ color

Stemming

- NLTK includes a few stemmers, the most common one is PorterStemmer

```
import nltk
from nltk.stem import PorterStemmer
```

```
ps = PorterStemmer()
sentence = "The boy is eating apples in different colors"

words = nltk.word_tokenize(sentence)
stemmed_words = [ps.stem(word) for word in words]
stemmed_words
```

```
['the', 'boy', 'is', 'eat', 'appl', 'in', 'differ', 'color']
```

Lemmatization

- Lemmatization reduces a word to its base or dictionary form called **lemma**
- Unlike stemming, lemmatization considers the morphological context of the word
- The output of lemmatization is always a proper word
  - e.g., stemming may convert “better” to “bett”, while lemmatization converts it to “good”
- The class WordNetLemmatizer perform lemmatizations using the WordNet lexicon

```
from nltk.stem import WordNetLemmatizer

wnl = WordNetLemmatizer()
sentence = "The boy is eating apples in different colors"

words = nltk.word_tokenize(sentence)
lemmas = [wnl.lemmatize(word) for word in words]
lemmas
```

```
['The', 'boy', 'is', 'eating', 'apple', 'in', 'different', 'color']
```

Part-of-Speech (POS) Tagging

- Identify the grammatical part of each word
  - e.g., nouns, verbs, adjectives, adverbs, pronouns, prepositions, conjunctions, etc.
- You can use the function **nltk.pos_tag()** to tag a given list of tokens:

```
text = "Apple is looking at buying a U.K. startup for $1 billion"

tokens = nltk.word_tokenize(text)
tagged = nltk.pos_tag(tokens)
tagged
```

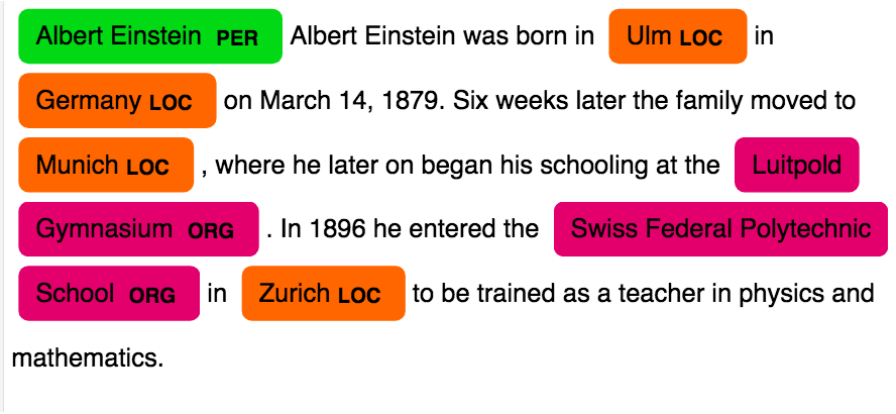
```
[('Apple', 'NNP'),
 ('is', 'VBZ'),
 ('looking', 'VBG'),
 ('at', 'IN'),
 ('buying', 'VBG'),
 ('a', 'DT'),
 ('U.K.', 'NNP'),
 ('startup', 'NN'),
 ('for', 'IN'),
 ('$', '$'),
 ('1', 'CD'),
 ('billion', 'CD')]
```

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

Tag	Description
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Whdeterminer
WP	Whpronoun
WP$	Possessive whpronoun
WRB	Whadverb

Named Entity Recognition (NER)

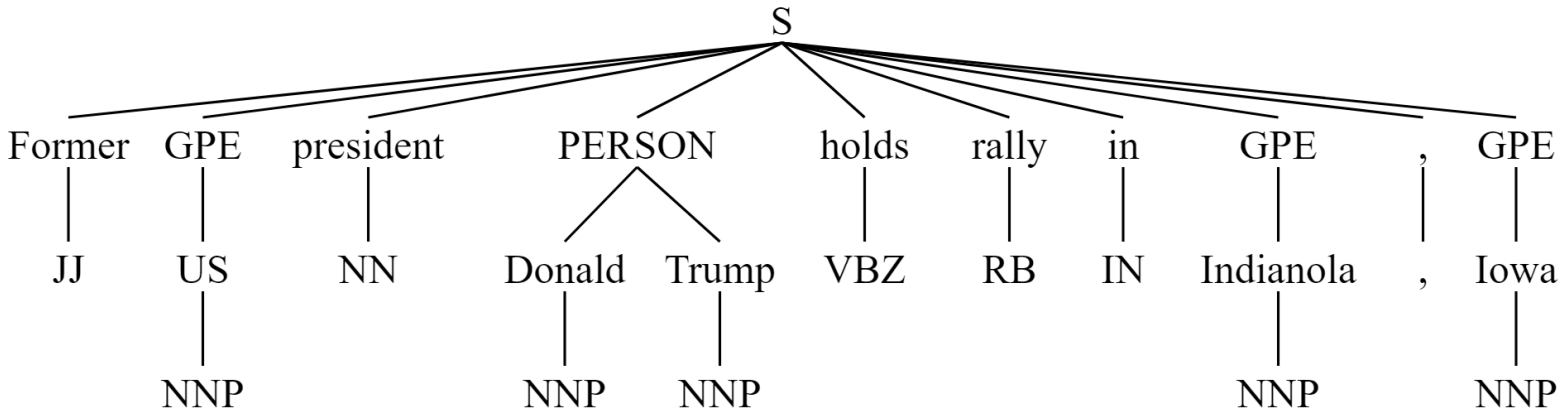
- NER locates and classifies named entities in the text into predefined categories
- The main categories are:
  - **Person**: Names of individuals
  - **Organization**: Names of companies, institutions, agencies, and other groups
  - **Location**: Names of geographical entities, like cities, countries, rivers, etc.
- Other categories include dates and times, monetary values, events, languages, etc.
- It is a crucial step before many NLP tasks, e.g., question answering and translation



Named Entity Recognition (NER)

- The method **nltk.chunk.ne_chunk()** performs an NER on a list of tagged tokens:

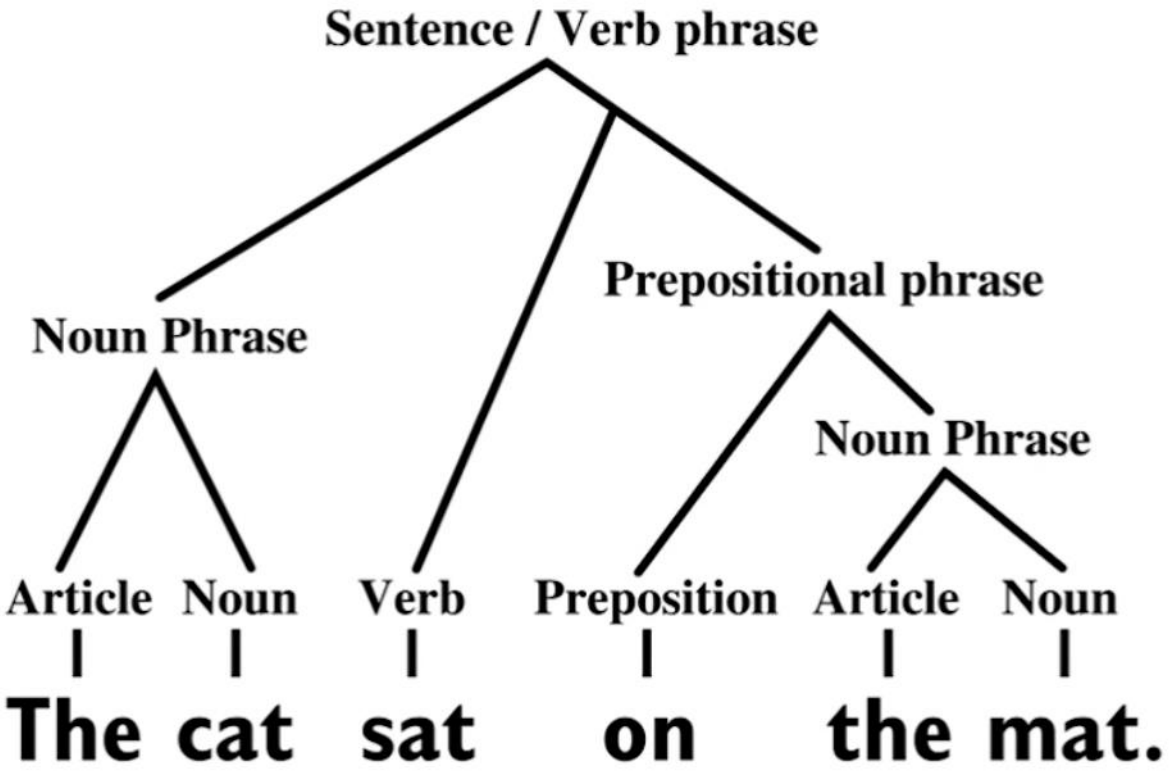
```
text = "Former US president Donald Trump holds rally in Indianola, Iowa"
tokens = nltk.word_tokenize(text)
tagged = nltk.pos_tag(tokens)
entities = nltk.chunk.ne_chunk(tagged)
entities
```

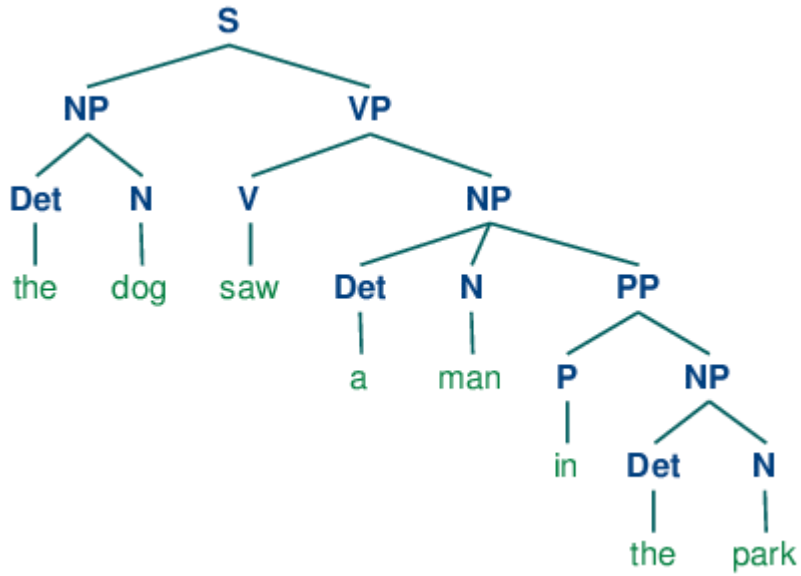


- Requires the svgling package for displaying the syntax tree (pip install svgling)

Syntax Trees

- Syntax tree is a tree representation of syntactic structure of sentences



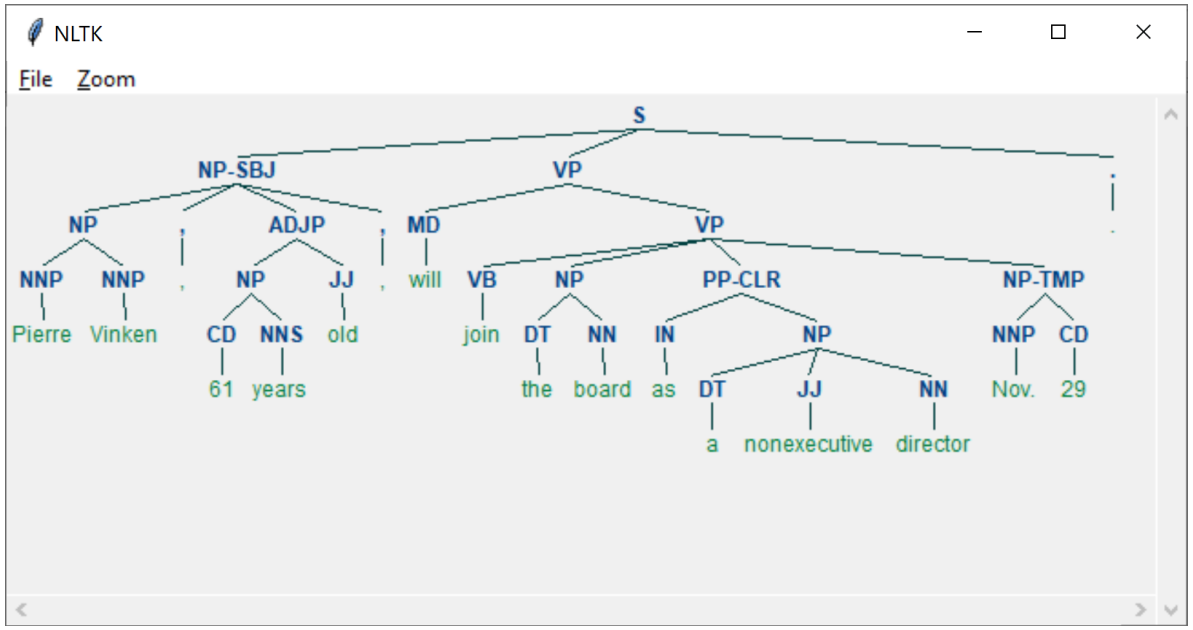


Syntax Trees

- Display a syntax tree in NLTK:

```
from nltk.corpus import treebank
```

```
treebank.parsed_sents()[0].draw()
```



Computing BLEU Score

- The nltk.translate.bleu_score module provides functions to calculate the BLEU score for a given set of reference translations and a candidate translation

```
from nltk.translate.bleu_score import sentence_bleu
```

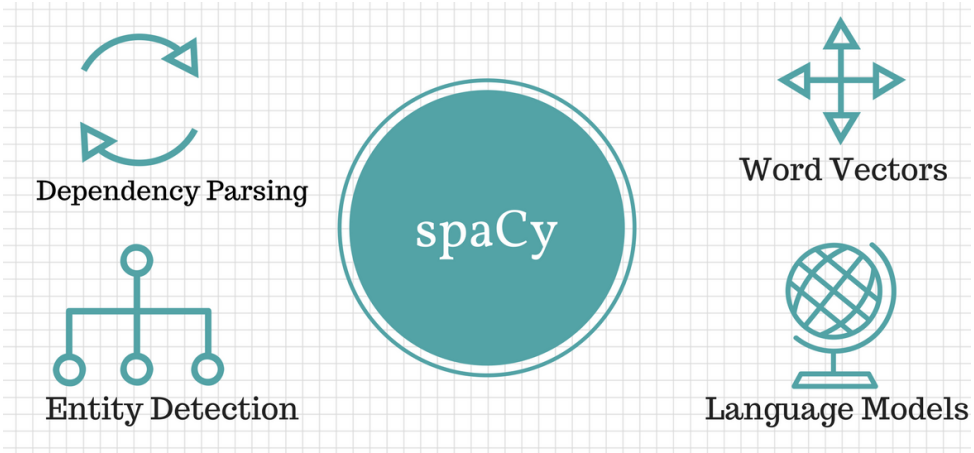
```
# Machine Translation (MT)
candidate = "The black cat sat on the mat".split()

# Reference Translation (Ref)
reference = "The black cat sat on a mat".split()

# Compute BLEU score
score = sentence_bleu([reference], candidate)
print('BLEU score:', score)
```

```
BLEU score: 0.6434588841607617
```

The SpaCy Library

- Provides tools for common NLP tasks
  - e.g., tokenization, part-of-speech tagging, NER, word vectors
- Has a customizable pipeline
- Supports multiple languages
- Provides integration with deep learning models
- Home page: https://spacy.io/
- Can be easily installed via pip

```
pip install spacy
```



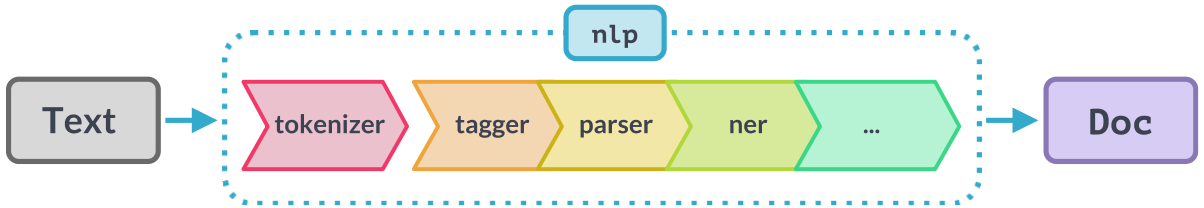
SpaCy Architecture

- **Language** is the core class for processing text
  - Contains the shared vocabulary, tokenization rules, and a processing pipeline
  - Created by loading a SpaCy model using the **spacy.load()** function
- A **Doc** object is created by processing a string of text with a language model
  - Each individual token in the Doc is represented as a **Token** object

Source: https://spacy.io/api

Processing Pipeline

- A **pipeline** consists of several components that are applied in sequence to Doc
  - such as tokenizer, tagger, parser, named entity recognizer
  - Can contain custom components such as statistical models
- Each of these components adds additional information to the Doc object



Loading SpaCy Models

- After installing SpaCy, you need to download an NLP model
  - A model includes tokenizer, tagger, parser and NER
- SpaCy offers various models for different languages
- For example, to download a small English model, use the following:

```
python -m spacy download en_core_web_sm
```

- To load the model in the script, use the following code:

```
import spacy
```

```
# Load the English NLP model
nlp = spacy.load('en_core_web_sm')
```

Text Processing

- To process some text, simply pass it to the loaded SpaCy model

```
# Process a text
doc = nlp("Apple is looking at buying a U.K. startup for $1 billion")
```

- This will create a Doc object that has been processed through the NLP pipeline
  - It is constructed by a Tokenizer and modified in place by the components of the pipeline
- The Doc object owns the sequence of tokens and all their annotations
- We can iterate over the Doc object to access each token:

```
# Tokenization
print([token.text for token in doc])
```

```
['Apple', 'is', 'looking', 'at', 'buying', 'a', 'U.K.', 'startup', 'for', '$', '1', 'billion']
```

POS Tagging and Lemmatization

- SpaCy's tokenization process provides various attributes for each token
- For example, you can access the lemma, part of speech, or whether the token is an alphabetic word:

```
for token in doc:
    print(token.text, token.lemma_, token.pos_, token.is_alpha)
```

```
Apple Apple PROPN True
is be AUX True
looking look VERB True
at at ADP True
buying buy VERB True
a a DET True
U.K. U.K. PROPN False
startup startup NOUN True
for for ADP True
$ $ SYM False
1 1 NUM False
billion billion NUM True
```

NER

- Use doc.ents to get the entities and their categories:

```
# Named entity recognition
for entity in doc.ents:
    print(entity.text, entity.label_)
```

```
Apple ORG
U.K. GPE
$1 billion MONEY
```

Example: Document Classification

- In the following example we'll use a Naïve Bayes classifier to classify text documents into different categories
- The "Twenty Newsgroups" dataset contains around 20,000 newsgroups posts, partitioned (nearly) evenly across 20 topics
- It has become popular for experiments in text applications of ML techniques, such as text classification and text clustering
- You can download the data set using the function fetch_20newsgroups():

```
from sklearn.datasets import fetch_20newsgroups

twenty_train = fetch_20newsgroups(subset='train')
twenty_test = fetch_20newsgroups(subset='test')
```

  - The training and test sets need to be downloaded separately
  - It may take a few minutes to download the data the first time you try to load it:

Example: Document Classification

- Let's take a look at the list of category names:

```
twenty_train.target_names
```

```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

Example: Document Classification

- The text files themselves are loaded in the **data** attribute
- The category index of each document is stored in the **target** attribute

```
print(twenty_train.data[0])
```

```
From: lerxst@wam.umd.edu (where's my thing)
Subject: WHAT car is this!?
Nntp-Posting-Host: rac3.wam.umd.edu
Organization: University of Maryland, College Park
Lines: 15

 I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In additi
on,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.

Thanks,
- IL
   ---- brought to you by your neighborhood Lerxst ----
```

```
twenty_train.target_names[twenty_train.target[0]]
```

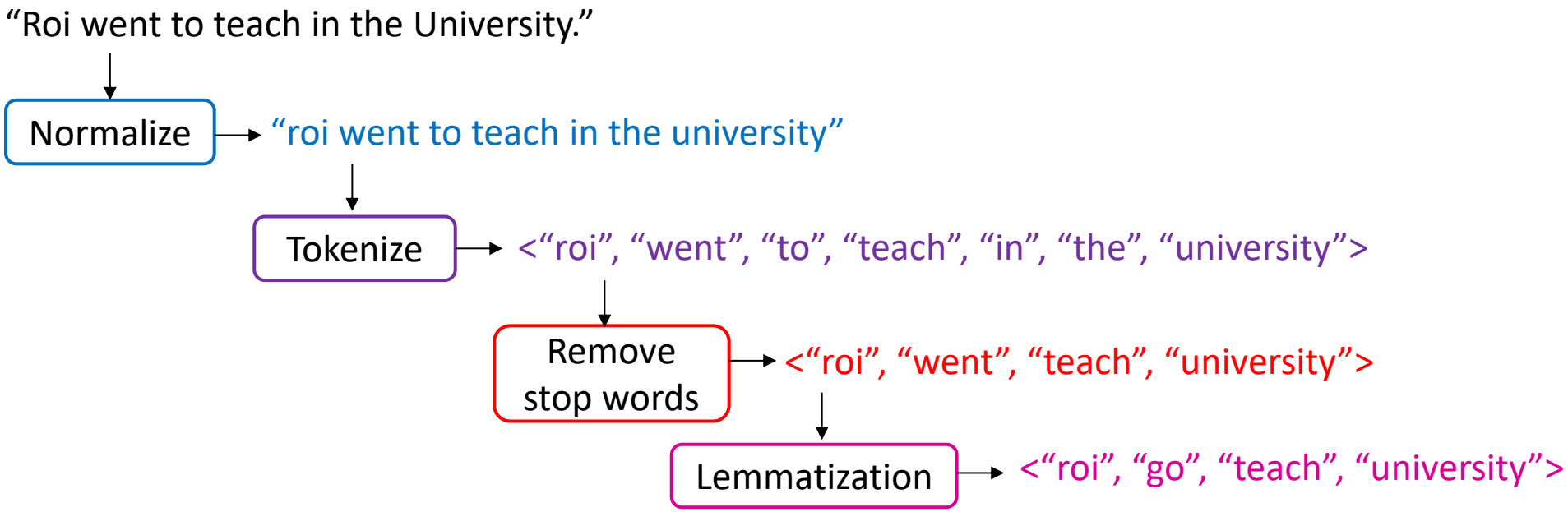
```
'rec.autos'
```

Text Analysis

- In order to perform ML on text documents, we first need to turn the text content into numerical feature vectors (or “vectorize” the text)
- This process typically involves two steps:
  - Pre-process, clean, and normalize the text
  - Transform the text documents into numerical representations

Text Preprocessing

- Common pre-processing techniques include:
  - Normalization (e.g., lower casing and removal of special characters)
  - Tokenization
  - Removal of stop words
  - Stemming / Lemmatization



Representing Text as Numbers

- Common strategies for converting text into numerical vectors:
  - Bag of words model
  - TF-IDF vectors
  - Word embeddings

Bag of Words (BoW) Model

- Each document is represented by a word counts vector
- Assign a fixed integer id to each word occurring in any document of the training set
  - e.g., by building a dictionary from words to integer indices
- For each document *i*, count the number of occurrences of each word *w* and store it in *X*[*i*, *j*] as the value of feature *j* where *j* is the index of word *w* in the dictionary
- *X* is called a **document-term matrix**
- This matrix is very sparse: most values of *X* are 0

	littl	hous	prairi	mari	lamb	silenc	twinkl	star
"Little House on the Prairie"	1	1	1	0	0	0	0	0
"Mary had a Little Lamb"	1	0	0	1	1	0	0	0
"The Silence of the Lambs"	0	0	0	0	1	1	0	0
"Twinkle Twinkle Little Star"	1	0	0	0	0	0	2	1

Document-Term Matrix

CountVectorizer

class `sklearn.feature_extraction.text.` **`CountVectorizer`**(*, *input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *analyzer='word'*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.int64'>*) [source]

- Converts a collection of text documents to a matrix of token counts
- Also includes text preprocessing, tokenizing and filtering of stopwords

Argument	Description
lowercase	Convert all characters to lowercase before tokenizing
analyzer	Whether the feature should be made of word or character n-grams
stop_words	Can be ‘english’ or a list of stop words
token_pattern	Regular expression denoting what constitutes a “token”
ngram_range	The lower and upper boundary of the range of n-values for different n-grams to be extracted
max_df	Ignore terms that have a document frequency strictly higher than the given threshold
min_df	Ignore terms that have a document frequency strictly lower than the given threshold

CountVectorizer

- Example using a sample corpus with 4 short documents:

```
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]
```

```
from sklearn.feature_extraction.text import CountVectorizer

count_vect = CountVectorizer()
X = count_vect.fit_transform(corpus)
```

```
X.shape
```

```
(4, 9)
```

CountVectorizer

- Once fitted, the vectorizer has built a dictionary that maps words to feature indices:

```
count_vect.vocabulary_
```

```
{'this': 8,
 'is': 3,
 'the': 6,
 'first': 2,
 'document': 1,
 'second': 5,
 'and': 0,
 'third': 7,
 'one': 4}
```

- get_feature_names() returns array mapping from feature indices to words:

```
count_vect.get_feature_names()
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

CountVectorizer

- CountVectorizer returns the document-term matrix as a **SciPy sparse matrix**
- Some Scikit estimator classes cannot handle sparse matrices (e.g., CategoricalNB)
- You can convert it to a standard NumPy array by calling its **toarray()** method

```
vocab = count_vect.get_feature_names()
pd.DataFrame(X.toarray(), columns=vocab)
```

	and	document	first	is	one	second	the	third	this
0	0	1	1	1	0	0	1	0	1
1	0	2	0	1	0	1	1	0	1
2	1	0	0	1	1	0	1	1	1
3	0	1	1	1	0	0	1	0	1

TF-IDF Model

- Issues with the bag of words model:
  - Longer documents have higher average word count values than shorter documents, even though they might discuss the same topics
  - Some of the words appear in many documents in the corpus and are therefore less informative than those that occur in a small subset of the corpus
- TF-IDF stands for “Term Frequency times Inverse Document Frequency”
- TF-IDF scales down the impact of words that occur frequently in the corpus
- It combines two metrics:
  - tf – term frequency
  - idf – inverse document frequency

TF-IDF Model

- Given a corpus of documents *D*
- For a term *t* in document *d* we compute:

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{|\{t' \in d\}|}$$

frequency of term t in document d

number of terms in document d

$$\mathrm{idf}(t) = \log \frac{|D|}{|\{d \in D | t \in d\}|}$$

number of documents that contain the term t

- Then tf-idf is calculated as:

$$\mathrm{tfidf}(t,d) = \mathrm{tf}(t,d) \cdot \mathrm{idf}(t)$$

Example of TF-IDF

- Given the following term count tables of two documents:

Document 1

Term	Term Count
this	1
is	1
a	2
sample	1

Document 2

Term	Term Count
this	1
is	1
another	2
example	3

- The tf-idf of the word “example” is:

$$\text{tf}(\text{"example"}, d_1) = \frac{0}{5} = 0$$

$$\text{idf}(\text{"example"}) = \log\left(\frac{2}{1}\right) = 0.301$$

$$\text{tfidf}(\text{"example"}, d_1) = 0 \cdot 0.301 = 0$$

$$\text{tf}(\text{"example"}, d_2) = \frac{3}{7} \approx 0.429$$

$$\text{tfidf}(\text{"example"}, d_2) = 0.429 \cdot 0.301 = 0.129$$

TfidfVectorizer

class `sklearn.feature_extraction.text.` **`TfidfVectorizer`**(*, *input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *analyzer='word'*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.float64'>*, *norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*) [source]

- Converts a collection of text documents to a matrix of TF-IDF features
- Equivalent to CountVectorizer followed by TfidfTransformer
- It has similar parameters to CountVectorizer

TfidfVectorizer

- Example using the previous sample corpus:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vect = TfidfVectorizer()
X = tfidf_vect.fit_transform(corpus)
```

```
vocab = tfidf_vect.get_feature_names()
pd.DataFrame(X.toarray(), columns=vocab)
```

	and	document	first	is	one	second	the	third	this
0	0.000000	0.469791	0.580286	0.384085	0.000000	0.000000	0.384085	0.000000	0.384085
1	0.000000	0.687624	0.000000	0.281089	0.000000	0.538648	0.281089	0.000000	0.281089
2	0.511849	0.000000	0.000000	0.267104	0.511849	0.000000	0.267104	0.511849	0.267104
3	0.000000	0.469791	0.580286	0.384085	0.000000	0.000000	0.384085	0.000000	0.384085

Example: Text Classification

- Let's define a pipeline that combines TF-IDF vectorizer and a Naïve Bayes classifier
- The variant of NB most suitable for word counts / TF-IDF is multinomial Naïve Bayes

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB

clf = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('clf', MultinomialNB())
])
```

- We can now train the model with a single command:

```
clf.fit(twenty_train.data, twenty_train.target)
```

```
Pipeline(steps=[('tfidf', TfidfVectorizer()), ('clf', MultinomialNB())])
```

Example: Text Classification

- Evaluation of the performance on the training and test sets:

```
clf.score(twenty_train.data, twenty_train.target)
```

```
0.9326498143892522
```

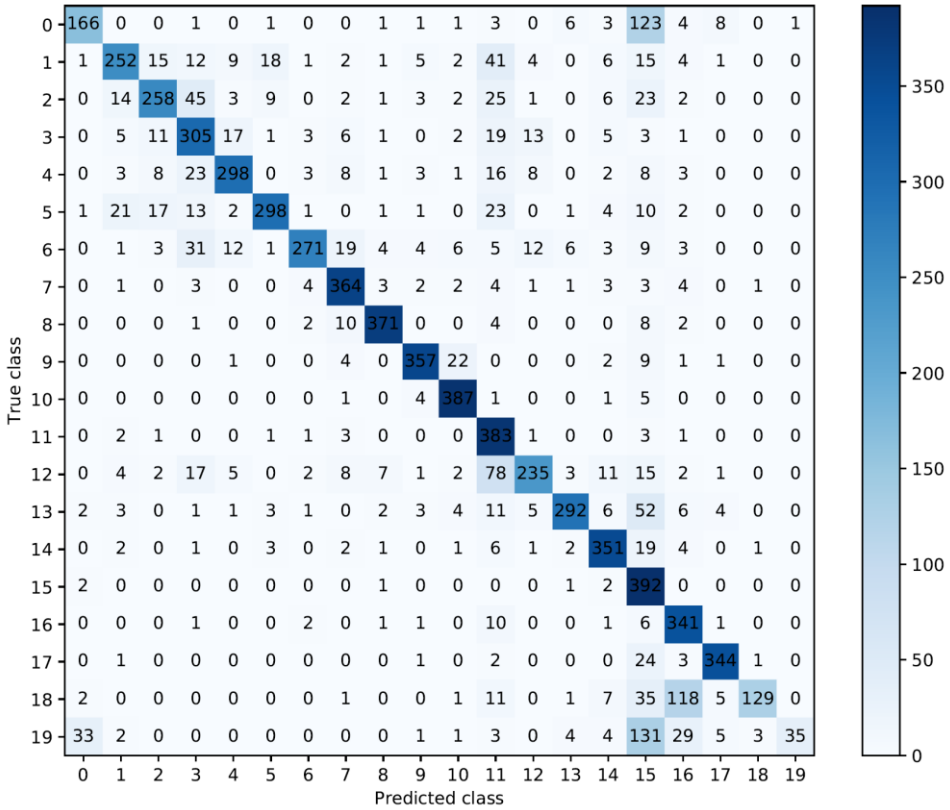
```
clf.score(twenty_test.data, twenty_test.target)
```

```
0.7738980350504514
```

Example: Text Classification

- Confusion matrix between the true and predicted labels on the test set:

```
from sklearn.metrics import confusion_matrix

test_pred = clf.predict(twenty_test.data)
conf_mat = confusion_matrix(twenty_test.target, test_pred)
plt.figure(figsize=(10, 8))
plot_confusion_matrix(conf_mat)
plt.savefig('figures/nb_conf_matrix.pdf')
```



- Posts from the newsgroups on atheism (#0) and Christianity (#15) are often confused

Example: Text Classification

- We can now use the model to predict the category of new documents:

```
new_docs = ['Fix the screen resolution', 'God is love', 'A car for sale']
new_pred = clf.predict(new_docs)

for doc, category in zip(new_docs, new_pred):
    print(doc, '=>', twenty_train.target_names[category])
```

```
Fix the screen resolution => comp.sys.mac.hardware
God is love => soc.religion.christian
A car for sale => misc.forsale
```

Handling Continuous Attributes

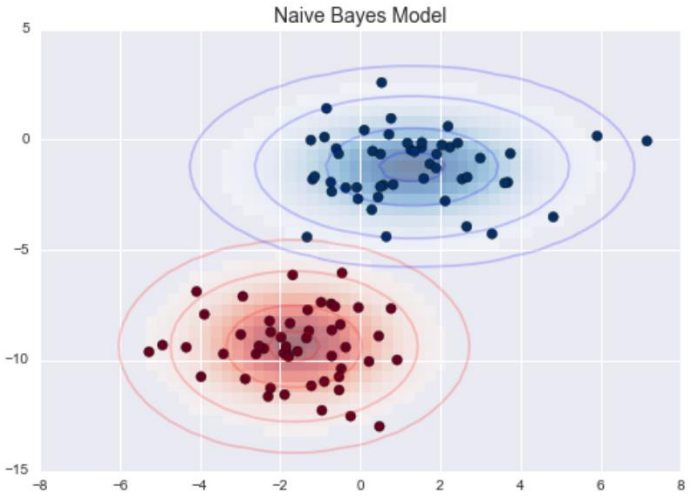
- There are two ways to handle continuous attributes in Naïve Bayes models:
  - Discretize each continuous attribute
  - Assume the continuous attribute has a certain form of probability distribution and estimate the parameters of the distribution from the training set
    - Typically a normal distribution is assumed

Gaussian Naïve Bayes

- The distribution of the features given the class is assumed to be Gaussian:

$$p(x_j|y=k) = \frac{1}{\sqrt{2\pi}\sigma_{jk}} \exp\left(-\frac{(x_j-\mu_{jk})^2}{2\sigma_{jk}^2}\right)$$

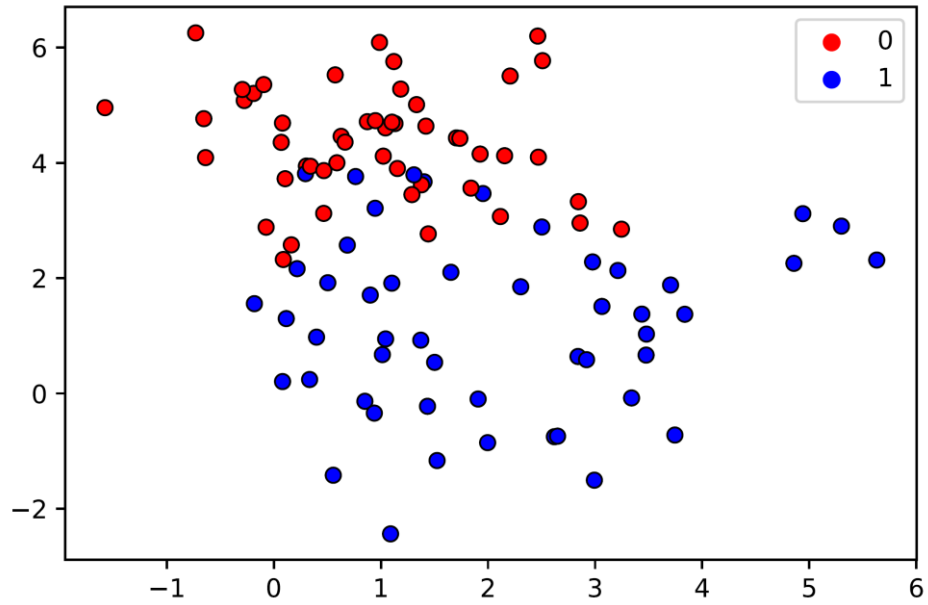
- The MLE estimates of the parameters μ_{jk}, σ_{jk} are the mean and standard deviation of feature x_j over all the data points that belong to class y in the training set



Gaussian Naïve Bayes

- For example, imagine that you have the following data:

```
from sklearn.datasets import make_blobs

X, y = make_blobs(100, 2, centers=2, cluster_std=[1, 1.5], random_state=0)
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y, palette=['r', 'b'],
                edgecolor='black')
```



Gaussian Naïve Bayes

- We first split the data into training and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=0)
```

- We now use GaussianNB to fit a Gaussian Naïve Bayes model to the data:

```
from sklearn.naive_bayes import GaussianNB

clf = GaussianNB()
clf.fit(X_train, y_train)
```

```
GaussianNB()
```

- The scores on the training and test sets are:

```
clf.score(X_train, y_train)
```

```
0.8875
```

```
clf.score(X_test, y_test)
```

```
0.9
```

Gaussian Naïve Bayes

- We can now use the fitted model to predict the class label for new samples:

```
clf.predict([[1, 2]])
```

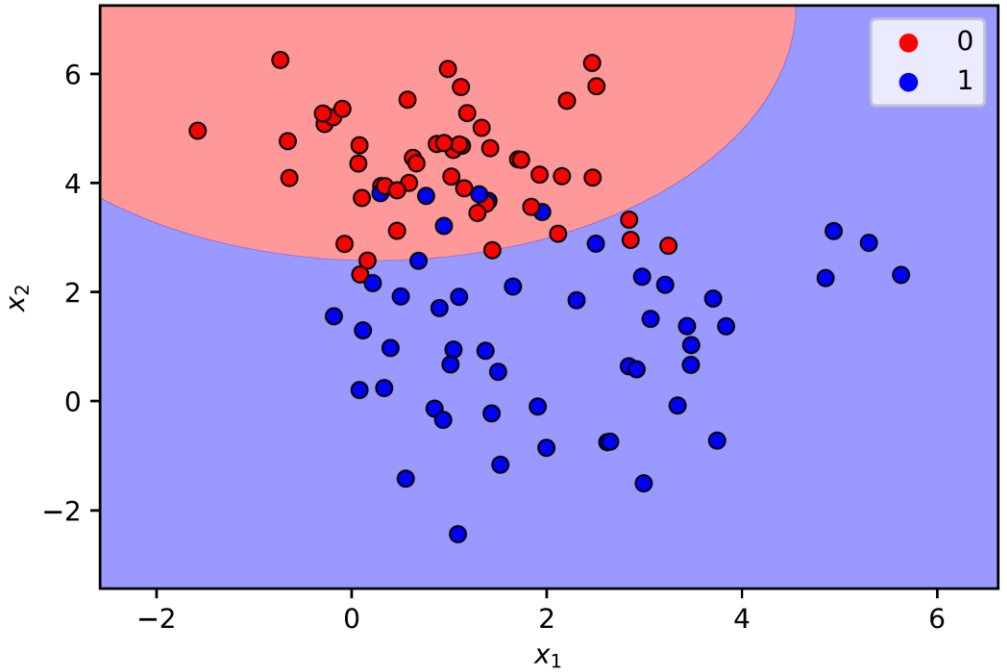
```
array([1])
```

```
clf.predict_proba([[1, 2]])
```

```
array([[0.17392267, 0.82607733]])
```

Gaussian Naïve Bayes

- We can also plot the decision boundary between the two classes:



- In general, the decision boundary will have a quadratic form
  - Unless the features have the same covariance matrix, in which case the quadratic part cancels out and the decision boundary is linear

Heterogonous Data Sets

- The following data set contains both categorical and continuous attributes
- In this case we can compute the class-conditional probability for each categorical attribute, along with the sample mean and variance for the continuous attributes

Tid	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	**No**
2	No	Married	100K	**No**
3	No	Single	70K	**No**
4	Yes	Married	120K	**No**
5	No	Divorced	95K	**Yes**
6	No	Married	60K	**No**
7	Yes	Divorced	220K	**No**
8	No	Single	85K	**Yes**
9	No	Married	75K	**No**
10	No	Single	90K	**Yes**

P(Home Owner=Yes|No) = 3/7
P(Home Owner=No|No) = 4/7
P(Home Owner=Yes|Yes) = 0
P(Home Owner=No|Yes) = 1
P(Marital Status=Single|No) = 2/7
P(Marital Status=Divorced|No) = 1/7
P(Marital Status=Married|No) = 4/7
P(Marital Status=Single|Yes) = 2/3
P(Marital Status=Divorced|Yes) = 1/3
P(Marital Status=Married|Yes) = 0

For Annual Income:
If class=No: sample mean=110
sample variance=2975
If class=Yes: sample mean=90
sample variance=25

Heterogonous Data Sets

- Predict the class label of a test sample
  x = (Home Owner = No, Marital Status = Married, Income = $120K)
- Using the information in the previous figure, the class-conditional probabilities are:

$$P(\mathbf{x}|\text{No}) = P(\text{Home Owner} = \text{No}|\text{No})P(\text{Status} = \text{Married}|\text{No})P(\text{Annual Income} = \$120\text{K}|\text{No})$$
$$= 4/7 \cdot 4/7 \cdot 0.0072 = 0.0024$$
$$P(\mathbf{x}|\text{Yes}) = P(\text{Home Owner} = \text{No}|\text{Yes})P(\text{Status} = \text{Married}|\text{Yes})P(\text{Annual Income} = \$120\text{K}|\text{Yes})$$
$$= 1 \cdot 0 \cdot 1.2 \cdot 10^{-9} = 0$$

- The prior probabilities of the classes are *P*(Yes) = 0.3 and *P*(No) = 0.7
- Therefore, the posterior probabilities are:

$$P(\text{No}|\mathbf{x}) = \alpha \cdot 0.7 \cdot 0.0024 = 0.0016\alpha$$
$$P(\text{Yes}|\mathbf{x}) = \alpha \cdot 0.3 \cdot 0 = 0$$

- Since *P*(No|**x**) > *P*(Yes|**x**), the sample is classified as No

Naïve Bayes Summary

Pros

- Very fast both in training and prediction
- Provide class probability estimates
- Easily interpretable
- Robust to noise
  - Noise points are averaged out when estimating the conditional probabilities
- Can handle missing values
  - Missing values are ignored when computing the conditional probability estimates
- Robust to irrelevant attributes
- Works well with high-dimensional data
- Very few hyperparameters

Cons

- Relies on the Naïve Bayes assumption
  - Correlated attributes can degrade the performance of the classifier
- Cannot handle continuous attributes without assumptions on the distribution
- Generally doesn’t perform as well as more complex models
- Can be used only for classification