Replacing and Correcting Words

No text here

Introduction

In this chapter, we will go over various word replacement and correction techniques.
The recipes cover the gamut of linguistic compression, spelling correction, and text normalization.

Stemming words

Stemming is a technique to remove affixes from a word, ending up with the stem.
For example, the stem of cooking is cook, and a good stemming algorithm knows that the ing suffix can be removed.
Instead of storing all forms of a word, a search engine can store only the stems, greatly reducing the size of index while increasing retrieval accuracy.
It is designed to remove and replace well-known suffixes of English words, and its usage in NLTK will be covered in the next section.
NLTK comes with an implementation of the Porter stemming algorithm, which is very easy to use.
Simply instantiate the PorterStemmer class and call the stem() method with the word you want to stem: >>> from nltk.stem import PorterStemmer >>> stemmer = PorterStemmer() >>> stemmer.stem('cooking') 'cook' >>> stemmer.stem('cookery') 'cookeri' How it works...
The PorterStemmer class knows a number of regular word forms and suffixes and uses   this knowledge to transform your input word to a final stem through a series of steps.
The resulting stem is often a shorter word, or at least a common form of the word, which has   the same root meaning.
It is known to   be slightly more aggressive than the PorterStemmer functions: >>> from nltk.stem import LancasterStemmer >>> stemmer = LancasterStemmer() >>> stemmer.stem('cooking') 'cook' >>> stemmer.stem('cookery') 'cookery' The RegexpStemmer class You can also construct your own stemmer using the RegexpStemmer class.
It takes   a single regular expression (either compiled or as a string) and removes any prefix or   suffix that matches the expression: >>> from nltk.stem import RegexpStemmer >>> stemmer = RegexpStemmer('ing') >>> stemmer.stem('cooking') 'cook' >>> stemmer.stem('cookery') 'cookery' >>> stemmer.stem('ingleside') 'leside' 31 www.it-ebooks.info

Lemmatizing words with WordNet

So unlike stemming, you are always left with a valid word that means the same thing.
This will allow the WordNetLemmatizer class to access WordNet. You should also be familiar with the part-of-speech tags covered in the Looking up Synsets for a word in WordNet recipe of Chapter 1, Tokenizing Text and WordNet Basics.
32 www.it-ebooks.info Chapter 2 How to do it...
We will use the WordNetLemmatizer class to find lemmas: >>> from nltk.stem import WordNetLemmatizer >>> lemmatizer = WordNetLemmatizer() >>> lemmatizer.lemmatize('cooking') 'cooking' >>> lemmatizer.lemmatize('cooking', pos='v') 'cook' >>> lemmatizer.lemmatize('cookbooks') 'cookbook' How it works...
The WordNetLemmatizer class is a thin wrapper around the wordnet corpus and uses the morphy() function of the WordNetCorpusReader class to find a lemma.
Unlike with stemming, knowing the part of speech of the word is important.
As demonstrated previously, cooking does not return a different lemma unless you specify that the POS is a verb.
Here's an example that illustrates one of the major differences between stemming and lemmatization: >>> from nltk.stem import PorterStemmer >>> stemmer = PorterStemmer() >>> stemmer.stem('believes') 'believ' >>> lemmatizer.lemmatize('believes') 'belief' Instead of just chopping off the es like the PorterStemmer class, the WordNetLemmatizer class finds a valid root word.
By returning a lemma, you will always get a valid word.

Replacing words matching regular expressions

If stemming and lemmatization are a kind of linguistic compression, then word replacement can be thought of as error correction or text normalization.
In this recipe, we will replace words based on regular expressions, with a focus on expanding contractions.
Remember when we were tokenizing words in Chapter 1, Tokenizing Text and WordNet Basics, and it was clear that most tokenizers had trouble with contractions?
This recipe aims to fix this by replacing contractions with their expanded forms, for example, by replacing "can't" with "cannot" or "would've" with "would have".
Getting ready Understanding how this recipe works will require a basic knowledge of regular expressions and the re module.
This will be a list of tuple pairs, where the first element is the pattern to match with and the second element is the replacement.
Next, we will create a RegexpReplacer class that will compile the patterns and provide a replace() method to substitute all the found patterns with their replacements.
The following code can be found in the replacers.py module in the book's code bundle and is meant to be imported, not typed into the console: import re replacement_patterns = [(r'won\'t', 'will not'), (r'can\'t', 'cannot'), (r'i\'m', 'i am'), (r'ain\'t', 'is not'), (r'(\w+)\'ll', '\g<1> will'), (r'(\w+)n\'t', '\g<1> not'), (r'(\w+)\'ve', '\g<1> have'), (r'(\w+)\'s', '\g<1> is'), (r'(\w+)\'re', '\g<1> are'), (r'(\w+)\'d', '\g<1> would')] class RegexpReplacer(object): def __init__(self, patterns=replacement_patterns): self.patterns = [(re.compile(regex), repl) for (regex, repl) in patterns] def replace(self, text): s = text for (pattern, repl) in self.patterns: s = re.sub(pattern, repl, s) return s 35 www.it-ebooks.info Replacing and Correcting Words How it works...
In replacement patterns, we have defined tuples such as r'(\w+)\'ve' and '\g<1> have'.
By grouping the characters before 've in parenthesis, a match group is found and can be used in the substitution pattern with the \g<1> reference.
This replacement technique can work with any kind of regular expression, not just contractions.
The RegexpReplacer class can take any list of replacement patterns for whatever purpose.

Removing repeating characters

This recipe presents a method to remove these annoying repeating characters in order to end up with a proper English word.
This will allow us to match and remove repeating characters.
It will have a replace() method that takes a single word and returns a more correct version of that word, with the dubious repeating characters removed.
This code can be found in replacers.py in the book's code bundle and is meant to be imported: import re class RepeatReplacer(object): def __init__(self):   self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)')   self.repl = r'\1\2\3' def replace(self, word):   repl_word = self.repeat_regexp.sub(self.repl, word)   if repl_word != word: return self.replace(repl_word) else: return repl_word 37 www.it-ebooks.info Replacing and Correcting Words And now some example use cases: >>> from replacers import RepeatReplacer >>> replacer = RepeatReplacer() >>> replacer.replace('looooove') 'love' >>> replacer.replace('oooooh') 'oh' >>> replacer.replace('goose') 'gose' How it works...
The RepeatReplacer class starts by compiling a regular expression to match and define a replacement string with backreferences.
The repeat_regexp pattern matches three groups: 0 or more starting characters (\w*) f A single character (\w) that is followed by another instance of that character (\2) f 0 or more ending characters (\w*) f The replacement string is then used to keep all the matched groups, while discarding the backreference to the second group.
This continues until only one o remains, when repeat_regexp no longer matches the string and no more characters are removed.
If WordNet recognizes the word, then we can stop replacing characters.
Here is the WordNet-augmented version: import re from nltk.corpus import wordnet class RepeatReplacer(object): def __init__(self):   self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)') self.repl = r'\1\2\3' 38 www.it-ebooks.info

Spelling correction with Enchant

Replacing repeating characters is actually an extreme form of spelling correction.
Getting ready You will need to install Enchant and a dictionary for it to use.
Enchant is an offshoot of the AbiWord open source word processor, and more information on it can be found at http://www.abisource.com/projects/enchant/.
39 www.it-ebooks.info Replacing and Correcting Words How to do it...
We will create a new class called SpellingReplacer in replacers.py, and this time, the replace() method will check Enchant to see whether the word is valid.
Then, in the replace() method, it first checks whether the given word is present in the dictionary.
If it is, no spelling correction is necessary and the word is returned.
If the word is not found, it looks up a list of suggestions and returns the first suggestion, as long as its edit distance is less than or equal to max_dist.
The max_dist value then acts as a constraint on the Enchant suggest function to ensure that no unlikely replacement words are returned.
Here is an example showing all the suggestions for languege, a misspelling of language: 40 www.it-ebooks.info Chapter 2 >>> import enchant >>> d = enchant.Dict('en') >>> d.suggest('languege') ['language', 'languages', 'languor', "language's"] Except for the correct suggestion, language, all the other words have an edit distance of three or greater.
You can first check whether the dictionary exists using enchant.dict_exists(), which will return True if the named dictionary exists, or False otherwise.
The en_GB dictionary Always ensure that you use the correct dictionary for whichever language you are performing spelling correction on.
The word theater is the American English spelling whereas the British English spelling is theatre: >>> import enchant >>> dUS = enchant.Dict('en_US') >>> dUS.check('theater') True >>> dGB = enchant.Dict('en_GB') >>> dGB.check('theater') False 41 www.it-ebooks.info Replacing and Correcting Words >>> from replacers import SpellingReplacer >>> us_replacer = SpellingReplacer('en_US') >>> us_replacer.replace('theater') 'theater' >>> gb_replacer = SpellingReplacer('en_GB') >>> gb_replacer.replace('theater') 'theatre' Personal word lists Enchant also supports personal word lists.
You could then create a dictionary augmented with your personal word list as follows: >>> d = enchant.Dict('en_US') >>> d.check('nltk') False >>> d = enchant.DictWithPWL('en_US', 'mywords.txt') >>> d.check('nltk') True To use an augmented dictionary with our SpellingReplacer class, we can create a subclass in replacers.py that takes an existing spelling dictionary: class CustomSpellingReplacer(SpellingReplacer): def __init__(self, spell_dict, max_dist=2): self.spell_dict = spell_dict self.max_dist = max_dist This CustomSpellingReplacer class will not replace any words that you put into mywords.txt: >>> from replacers import CustomSpellingReplacer >>> d = enchant.DictWithPWL('en_US', 'mywords.txt') >>> replacer = CustomSpellingReplacer(d) >>> replacer.replace('nltk') 'nltk' See also The previous recipe covered an extreme form of spelling correction by replacing repeating characters.
You can also perform spelling correction by simple word replacement as discussed in the next recipe.

Replacing synonyms

It is often useful to reduce the vocabulary of a text by replacing words with common synonyms. Vocabulary reduction can also increase the occurrence of significant collocations, which was covered in the Discovering word collocations recipe of Chapter 1, Tokenizing Text and WordNet Basics. Getting ready You will need a defined mapping of a word to its synonym.

We will start by hardcoding the synonyms as a Python dictionary, and then explore other options to store synonym maps.

We'll first create a WordReplacer class in replacers.py that takes a word replacement mapping: class WordReplacer(object): def __init__(self, word_map): self.word_map = word_map def replace(self, word): return self.word_map.get(word, word) Then, we can demonstrate its usage for simple word replacement: >>> from replacers import WordReplacer >>> replacer = WordReplacer({'bday': 'birthday'}) >>> replacer.replace('bday') 'birthday' >>> replacer.replace('happy') 'happy' How it works... The replace() method looks up the given word in its word_map dictionary and returns the replacement synonym if it exists.

43 www.it-ebooks.info Replacing and Correcting Words If you were only using the word_map dictionary, you wouldn't need the WordReplacer class and could instead call word_map.get() directly. However, WordReplacer can act as a base class for other classes that construct the word_map dictionary from various file formats.

CSV synonym replacement The CsvWordReplacer class extends WordReplacer in replacers.py in order to construct the word_map dictionary from a CSV file: import csv class CsvWordReplacer(WordReplacer): def __init__(self, fname): word_map = {} for line in csv.reader(open(fname)): word, syn = line word_map[word] = syn super(CsvWordReplacer, self).__init__(word_map) Your CSV file should consist of two columns, where the first column is the word and the second column is the synonym meant to replace it.

If this file is called synonyms.csv and the first line is bday, birthday, then you can perform the following: >>> from replacers import CsvWordReplacer >>> replacer = CsvWordReplacer('synonyms.csv') >>> replacer.replace('bday') 'birthday' >>> replacer.replace('happy') 'happy' 44 www.it-ebooks.info Chapter 2 YAML synonym replacement If you have PyYAML installed, you can create YamlWordReplacer in replacers.py as shown in the following: import yaml class YamlWordReplacer(WordReplacer): def __init__(self, fname): word_map = yaml.load(open(fname)) super(YamlWordReplacer, self).__init__(word_map) Download and installation instructions for PyYAML are located at http://pyyaml.org/wiki/PyYAML.

You can also type pip install pyyaml on the command prompt Your YAML file should be a simple mapping of word: synonym, such as bday: birthday.

If the file is named synonyms.yaml, then you can perform the following: >>> from replacers import YamlWordReplacer >>> replacer = YamlWordReplacer('synonyms.yaml') >>> replacer.replace('bday') 'birthday' >>> replacer.replace('happy') 'happy' See also You can use the WordReplacer class to perform any kind of word replacement, even spelling correction for more complicated words that can't be automatically corrected, as we did in the previous recipe.

Replacing negations with antonyms

This time, instead of creating custom word mappings, we can use WordNet to replace words with unambiguous antonyms.
Refer to the Looking up lemmas and synonyms in WordNet recipe in Chapter 1, Tokenizing Text and WordNet Basics, for more details on antonym lookups.
With antonym replacement, you can replace not uglify with beautify, resulting in the sentence let's beautify our code.
To do this, we will create an AntonymReplacer class in replacers.py as follows: from nltk.corpus import wordnet class AntonymReplacer(object): def replace(self, word, pos=None): antonyms = set() for syn in wordnet.synsets(word, pos=pos): for lemma in syn.lemmas(): for antonym in lemma.antonyms(): antonyms.add(antonym.name()) if len(antonyms) == 1: return antonyms.pop() else: return None def replace_negations(self, sent): i, l = 0, len(sent) words = [] while i < l: word = sent[i] if word == 'not' and i+1 < l: ant = self.replace(sent[i+1]) if ant: words.append(ant) i += 2 continue words.append(word) i += 1 return words 46 www.it-ebooks.info Chapter 2 Now, we can tokenize the original sentence into ["let's", 'not', 'uglify', 'our', 'code'] and pass this to the replace_negations() function.
The replace() method takes a single word and an optional part-of-speech tag, then looks up the Synsets for the word in WordNet. Going through all the Synsets and every lemma of each Synset, it creates a set of all antonyms found.
In replace_negations(), we look through a tokenized sentence for the word not.
All other words are appended as is, resulting in a tokenized sentence with unambiguous negations replaced by their antonyms.
As unambiguous antonyms aren't very common in WordNet, you might want to create a custom antonym mapping in the same way we did for synonyms.
This AntonymWordReplacer can be constructed by inheriting from both WordReplacer and AntonymReplacer: class AntonymWordReplacer(WordReplacer, AntonymReplacer): pass The order of inheritance is very important, as we want the initialization and replace function of WordReplacer combined with the replace_negations function from AntonymReplacer.
The result is a replacer that can perform the following: >>> from replacers import AntonymWordReplacer >>> replacer = AntonymWordReplacer({'evil': 'good'}) >>> replacer.replace_negations(['good', 'is', 'not', 'evil']) ['good', 'is', 'good'] 47 www.it-ebooks.info Replacing and Correcting Words Of course, you can also inherit from CsvWordReplacer or YamlWordReplacer instead of WordReplacer if you want to load the antonym word mappings from a file.
In Chapter 1, Tokenizing Text and WordNet Basics, WordNet usage is covered in detail in the Looking up Synsets for a word in WordNet and Looking up lemmas and synonyms in WordNet recipes.

Creating Custom Corpora

No text here

Introduction

In this chapter, we'll cover how to use corpus readers and create custom corpora.
If you want to train your own model, such as a part-of-speech tagger or text classifier, you will need to create a custom corpus to train on.
Model training is covered in the subsequent chapters.
This information is essential for future chapters when we'll need to access the corpora as training data.

Setting up a custom corpus

In order to avoid conflict with the official data package, we'll create a custom nltk_data directory in our home directory.
The following is some Python code to create this directory and verify that it is in the list of known paths specified by nltk.data.path: >>> import os, os.path >>> path = os.path.expanduser('~/nltk_data') >>> if not os.path.exists(path): ...
The path should be %UserProfile%\nltk_data on Windows, or ~/nltk_data on Unix, Linux, and Mac OS X.
50 www.it-ebooks.info Chapter 3 If the last line does not return True, try creating the nltk_data directory manually in your home directory, then verify that the absolute path is in nltk.data.path.
It's essential to ensure that this directory exists and is in nltk.data.path before continuing.
You can see a list of the directories by running python -c "import nltk.data; print(nltk.data.path)".
Create this corpora directory within the nltk_data directory, so that the path is ~/nltk_data/corpora.
Let's call it cookbook, giving us the full path, which is ~/nltk_data/corpora/cookbook.
So on Unix, Linux, and Mac OS X, you could run the following to create the directory: mkdir -p ~/nltk_data/corpora/cookbook Now, we can create a simple wordlist file and make sure it loads.
Put this file into ~/nltk_data/corpora/cookbook/.
Now we can use nltk.data.load(), as shown in the following code, to load the file: >>> import nltk.data >>> nltk.data.load('corpora/cookbook/mywords.txt', format='raw') b'nltk\n' We need to specify format='raw' since nltk.data.load() doesn't know how to interpret .txt files.
The nltk.data.load() function recognizes a number of formats, such as 'raw', 'pickle', and 'yaml'.
In the previous case, we have a .txt file, which is not a recognized extension, so we have to specify the 'raw' format.
But, if we used a file that ended in .yaml, then we would not need to specify the format.
Filenames passed into nltk.data.load() can be absolute or relative paths.
The file is found using nltk.data.find(path), which searches all known paths combined with the relative path.
When using relative paths, be sure to use choose unambiguous names for your files so as not to conflict with any existing NLTK data.

Creating a wordlist corpus

In fact, you've already used it when we used the stopwords corpus in Chapter 1, Tokenizing Text and WordNet Basics, in the Filtering stopwords in a tokenized sentence and Discovering word collocations recipes.
Let's create a file named wordlist that looks like this: nltk corpus corpora wordnet 52 www.it-ebooks.info Chapter 3 How to do it...
Now we can instantiate a WordListCorpusReader class that will produce a list of words from our file. Otherwise, you must use a directory path such as nltk_data/ corpora/cookbook: >>> from nltk.corpus.reader import WordListCorpusReader >>> reader = WordListCorpusReader('.', ['wordlist']) >>> reader.words() ['nltk', 'corpus', 'corpora', 'wordnet'] >>> reader.fileids() ['wordlist'] How it works...
The CorpusReader class does all the work of identifying which files to read, while WordListCorpusReader reads the files and tokenizes each line   to produce a list of words.
The following is an inheritance diagram: CorpusReader ?leids() WordListCorpusReader words() When you call the words() function, it calls nltk.tokenize.line_tokenize() on the raw file data, which you can access using the raw() function as follows: >>> reader.raw() 'nltk\ncorpus\ncorpora\nwordnet\n' >>> from nltk.tokenize import line_tokenize >>> line_tokenize(reader.raw()) ['nltk', 'corpus', 'corpora', 'wordnet'] 53 www.it-ebooks.info Creating Custom Corpora There's more...
In the Filtering stopwords in a tokenized sentence recipe in Chapter 1, Tokenizing Text and WordNet Basics, we saw that it had one wordlist file for each language, and you could access the words for that language by calling stopwords.words(fileid).
If you want to create your own multifile wordlist corpus, this is a great example to follow.
There's one file with 850 basic words,   and another list with over 200,000 known English words, as shown in the following code: >>> from nltk.corpus import words >>> words.fileids() ['en', 'en-basic'] >>> len(words.words('en-basic')) 850 >>> len(words.words('en')) 234936 See also The Filtering stopwords in a tokenized sentence recipe in Chapter 1, Tokenizing Text and WordNet Basics, has more details on using the stopwords corpus.
In the following recipes, we'll cover more advanced corpus file formats and corpus reader classes.

Creating a part-of-speech tagged   word corpus

How to train and use a tagger is covered in detail in Chapter 4, Part-of-speech Tagging, but first we must know how to create and use a training corpus of part-of-speech tagged words.
For example, the treebank corpus uses different tags as compared to the brown corpus, even though both are English text.
If you were to put the previous excerpt into a file called brown.pos, you could then create a TaggedCorpusReader class using the following code: >>> from nltk.corpus.reader import TaggedCorpusReader >>> reader = TaggedCorpusReader('.', r'.*\.pos') >>> reader.words() ['The', 'expense', 'and', 'time', 'involved', 'are', ...] >>> reader.tagged_words() [('The', 'AT-TL'), ('expense', 'NN'), ('and', 'CC'), ...] >>> reader.sents() [['The', 'expense', 'and', 'time', 'involved', 'are', 'astronomical', '.']] >>> reader.tagged_sents() 55 www.it-ebooks.info Creating Custom Corpora [[('The', 'AT-TL'), ('expense', 'NN'), ('and', 'CC'), ('time', 'NN'), ('involved', 'VBN'), ('are', 'BER'), ('astronomical', 'JJ'), ('.', '.')]] >>> reader.paras() [[['The', 'expense', 'and', 'time', 'involved', 'are', 'astronomical', '.']]] >>> reader.tagged_paras() [[[('The', 'AT-TL'), ('expense', 'NN'), ('and', 'CC'), ('time', 'NN'), ('involved', 'VBN'), ('are', 'BER'), ('astronomical', 'JJ'), ('.', '.')]]] How it works...
We could have done the same thing as we did with the WordListCorpusReader class, and pass ['brown.pos'] as the second argument, but this way you can see how to include multiple files in a corpus without naming each one explicitly.
The TaggedCorpusReader class provides a number of methods for extracting text from a corpus. First, you can get a list of all words or a list of tagged tokens.
Finally, you can get a list of paragraphs, where each paragraph is a list of sentences and each sentence is a list of words or tagged tokens.
The following is an inheritance diagram listing all the major methods: CorpusReader ?leids() TaggedCorpusReader words() sents() paras() tagged_words() tagged_sents() tagged_paras() 56 www.it-ebooks.info Chapter 3 There's more...
The TaggedCorpusReader class tries to have good defaults, but you can customize them by passing in your own tokenizers at the time of initialization.
If you want to use a different tokenizer, you can pass that in as word_tokenizer, as shown in the following code: >>> from nltk.tokenize import SpaceTokenizer >>> reader = TaggedCorpusReader('.', r'.*\.pos', word_ tokenizer=SpaceTokenizer()) >>> reader.words() ['The', 'expense', 'and', 'time', 'involved', 'are', ...] Customizing the sentence tokenizer The default sentence tokenizer is an instance of nltk.tokenize.RegexpTokenize with '\n' to identify the gaps.
To customize this, you can pass in your own tokenizer as sent_tokenizer, as shown in the following code: >>> from nltk.tokenize import LineTokenizer >>> reader = TaggedCorpusReader('.', r'.*\.pos', sent_ tokenizer=LineTokenizer()) >>> reader.sents() [['The', 'expense', 'and', 'time', 'involved', 'are', 'astronomical', '.']] Customizing the paragraph block reader Paragraphs are assumed to be split by blank lines.
There are a number of other block reader functions in nltk.corpus.reader.util, whose purpose is to read blocks of text from a stream.
Customizing the tag separator If you don't want to use '/' as the word/tag separator, you can pass an alternative string to TaggedCorpusReader for sep.
The default is sep='/', but if you want to split words and tags with '|', such as 'word|tag', then you should pass in sep='|'.
57 www.it-ebooks.info Creating Custom Corpora Converting tags to a universal tagset NLTK 3.0 provides a method for converting known tagsets to a universal tagset.
For example, treebank tag mappings are in nltk_data/taggers/universal_tagset/en-ptb.map.

Creating a chunked phrase corpus

This is exactly what chunks are: subtrees within a sentence tree, and they will be covered in much more detail in Chapter 5, Extracting Chunks.
The following is a sample sentence tree with three Noun Phrase (NP) chunks shown as subtrees: S .. ? NP NP NP trimmed VBN have VBP about IN said VBD Earlier JJR staff-reduction NN moves NNS jobs NNS 300 CD the DT spoke man NN s This recipe will cover how to create a corpus with sentences that contain chunks.
Getting ready The following is an excerpt from the tagged treebank corpus.
It has part-of-speech tags, as in the previous recipe, but it also has square brackets for denoting chunks.
The following sentence is the same sentence as in the previous tree diagram, but in text form: [Earlier/JJR staff-reduction/NN moves/NNS] have/VBP trimmed/VBN about/ IN [300/CD jobs/NNS] ,/, [the/DT spokesman/NN] said/VBD ./.
In this format, every chunk is a noun phrase.
Words that are not within brackets are part of the sentence tree, but are not part of any noun phrase subtree.
Put the previous excerpt into a file called treebank.chunk, and then do the following: >>> from nltk.corpus.reader import ChunkedCorpusReader >>> reader = ChunkedCorpusReader('.', r'.*\.chunk') >>> reader.chunked_words() [Tree('NP', [('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS')]), ('have', 'VBP'), ...] >>> reader.chunked_sents() [Tree('S', [Tree('NP', [('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS')]), ('have', 'VBP'), ('trimmed', 'VBN'), ('about', 'IN'), Tree('NP', [('300', 'CD'), ('jobs', 'NNS')]), (',', ','), Tree('NP', [('the', 'DT'), ('spokesman', 'NN')]), ('said', 'VBD'), ('.', '.')])] >>> reader.chunked_paras() [[Tree('S', [Tree('NP', [('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS')]), ('have', 'VBP'), ('trimmed', 'VBN'), ('about', 'IN'), Tree('NP', [('300', 'CD'), ('jobs', 'NNS')]), (',', ','), Tree('NP', [('the', 'DT'), ('spokesman', 'NN')]), ('said', 'VBD'), ('.', '.')])]] The ChunkedCorpusReader class provides the same methods as the TaggedCorpusReader for getting tagged tokens, along with three new methods for getting chunks.
In chunked_sents(), you get a list of sentence trees, with each noun phrase as a subtree of the sentence.
In chunked_words(), you get a list of noun phrase trees alongside tagged tokens of words that were not in a chunk.
The following is an inheritance diagram listing the major methods: CorpusReader ?leids() ChunkedCorpusReader words() sents() paras() tagged_words() tagged_sents() tagged_paras() chunked_words() chunked_sents() chunked_paras() 60 www.it-ebooks.info Chapter 3 You can draw a tree by calling the draw() method.
Using the corpus reader defined earlier, you could do reader.chunked_sents()[0].draw() to get the same sentence tree diagram shown at the beginning of this recipe.
The default is nltk.chunk.util.tagstr2tree(), which parses a sentence string containing bracketed chunks into a sentence tree, with each chunk as a noun phrase subtree.
An alternative format for denoting chunks is called IOB tags.
IOB tags are similar to part-of-speech tags, but provide a way to denote the inside, outside, and beginning of a chunk.
Each word is on its own line with a part-of-speech tag followed by an IOB tag: Mr. NNP B-NP Meador NNP I-NP had VBD B-VP been VBN I-VP executive JJ B-NP vice NN I-NP president NN I-NP of IN B-PP Balcor NNP B-NP .
B-VP and I-VP denote the beginning and inside of a verb phrase.

Creating a categorized text corpus

The brown corpus, for example, has a number of different categories, as shown in the following code: >>> from nltk.corpus import brown >>> brown.categories() ['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance', 'science_fiction'] In this recipe, we'll learn how to create our own categorized text corpus. Getting ready The easiest way to categorize a corpus is to have one file for each category.

These two superclasses require three arguments: the root directory, the fileids arguments,   and a category specification: >>> from nltk.corpus.reader import CategorizedPlaintextCorpusReader >>> reader = CategorizedPlaintextCorpusReader('.', r'movie_.*\.txt', cat_pattern=r'movie_(\w+)\.txt') >>> reader.categories() ['neg', 'pos'] >>> reader.fileids(categories=['neg']) ['movie_neg.txt'] >>> reader.fileids(categories=['pos']) ['movie_pos.txt'] 64 www.it-ebooks.info Chapter 3 How it works...

The first two arguments to CategorizedPlaintextCorpusReader are the root directory and fileids, which are passed on to the PlaintextCorpusReader class to read in the files.

In our case, the category is the part of the fileid argument after movie_ and before .txt.

The cat_pattern keyword is passed to CategorizedCorpusReader, which overrides the common corpus reader functions such as fileids(), words(), sents(), and paras() to accept a categories keyword argument.

The CategorizedCorpusReader class also provides the categories() function, which returns a list of all the known categories in the corpus.

The CategorizedPlaintextCorpusReader class is an example of using multiple inheritance to join methods from multiple superclasses, as shown in the following diagram: CorpusReader ?leids() CategorizedCorpusReader categories() ?leids() PlaintextCorpusReader words() sents() paras() CategorizedPlaintextCorpusReader There's more...

Instead of cat_pattern, you could pass in a cat_map, which is a dictionary mapping   a fileid argument to a list of category labels, as shown in the following code: >>> reader = CategorizedPlaintextCorpusReader('.', r'movie_.*\.txt', cat_map={'movie_pos.txt': ['pos'], 'movie_neg.txt': ['neg']}) >>> reader.categories() ['neg', 'pos'] 65 www.it-ebooks.info

Creating a categorized chunk corpus reader

NLTK provides a CategorizedPlaintextCorpusReader and CategorizedTaggedCorpusReader class, but there's no categorized corpus reader for chunked corpora.
66 www.it-ebooks.info Chapter 3 Getting ready Refer to the earlier recipe, Creating a chunked phrase corpus, for an explanation of ChunkedCorpusReader, and refer to the previous recipe for details on CategorizedPlaintextCorpusReader and CategorizedTaggedCorpusReader, both of which inherit from CategorizedCorpusReader.
We'll create a class called CategorizedChunkedCorpusReader that inherits from both CategorizedCorpusReader and ChunkedCorpusReader.
The following code is found in catchunked.py: from nltk.corpus.reader import CategorizedCorpusReader, ChunkedCorpusReader class CategorizedChunkedCorpusReader(CategorizedCorpusReader, ChunkedCorpusReader): def __init__(self, *args, **kwargs): CategorizedCorpusReader.__init__(self, kwargs) ChunkedCorpusReader.__init__(self, *args, **kwargs) def _resolve(self, fileids, categories): if fileids is not None and categories is not None: raise ValueError('Specify fileids or categories, not both') if categories is not None: return self.fileids(categories) else: return fileids All of the following methods call the corresponding function in ChunkedCorpusReader with the value returned from _resolve().
We'll start with the plain text methods: def raw(self, fileids=None, categories=None): return ChunkedCorpusReader.raw(self, self._resolve(fileids, categories)) def words(self, fileids=None, categories=None): return ChunkedCorpusReader.words(self, self._resolve(fileids, categories)) def sents(self, fileids=None, categories=None): return ChunkedCorpusReader.sents(self, self._resolve(fileids, categories)) 67 www.it-ebooks.info Creating Custom Corpora def paras(self, fileids=None, categories=None): return ChunkedCorpusReader.paras(self, self._resolve(fileids, categories)) Next is the code for the tagged text methods: def tagged_words(self, fileids=None, categories=None): return ChunkedCorpusReader.tagged_words(self, self._resolve(fileids, categories)) def tagged_sents(self, fileids=None, categories=None): return ChunkedCorpusReader.tagged_sents(self, self._resolve(fileids, categories)) def tagged_paras(self, fileids=None, categories=None): return ChunkedCorpusReader.tagged_paras(self, self._resolve(fileids, categories)) And finally, we have code for the chunked methods, which is what we've really been after: def chunked_words(self, fileids=None, categories=None): return ChunkedCorpusReader.chunked_words(self, self._resolve(fileids, categories)) def chunked_sents(self, fileids=None, categories=None): return ChunkedCorpusReader.chunked_sents(self, self._resolve(fileids, categories)) def chunked_paras(self, fileids=None, categories=None): return ChunkedCorpusReader.chunked_paras(self, self._resolve(fileids, categories)) All these methods together give us a complete CategorizedChunkedCorpusReader class.
The CategorizedChunkedCorpusReader class overrides all the ChunkedCorpusReader methods to take a categories argument for locating fileids.
This _resolve() function makes use of CategorizedCorpusReader.fileids() to return fileids for a given list of categories.
68 www.it-ebooks.info Chapter 3 The inheritance diagram looks like this: CorpusReader CategorizedCorpusReader categories() fileids() fileids() ChunkedCorpusReader words() sents() paras() tagged_words() tagged_sents() tagged_paras() chunked_words() chunked_sents() chunked_paras() CategorizedChunkedCorpusReader The following is example code for using the

Lazy corpus loading

To speed up module import time when a corpus reader is defined, NLTK provides a LazyCorpusLoader class that can transform itself into your actual corpus reader as soon as you need it.
The LazyCorpusLoader class requires two arguments: the name of the corpus and the corpus reader class, plus any other arguments needed to initialize the corpus reader class.
The name argument specifies the root directory name of the corpus, which must be within a corpora subdirectory of one of the paths in nltk.data.path.
For example, if you have a custom corpora named cookbook in your local nltk_data directory, its path would be ~/nltk_data/corpora/cookbook.
You'd then pass 'cookbook' to LazyCorpusLoader as the name, and LazyCorpusLoader will   look in ~/nltk_data/corpora for a directory named 'cookbook'.
73 www.it-ebooks.info Creating Custom Corpora The second argument to LazyCorpusLoader is reader_cls, which should be the name of a subclass of CorpusReader, such as WordListCorpusReader.
You will also need to pass in any other arguments required by the reader_cls argument for initialization.
The LazyCorpusLoader class stores all the arguments given, but otherwise does nothing until you try to access an attribute or method.
Calls nltk.data.find('corpora/%s' % name) to find the corpus data   root directory.
2. Instantiates the corpus reader class with the root directory and any other arguments.

Creating a custom corpus view

A corpus view is a class wrapper around a corpus file that reads in blocks of tokens as needed.
But, if you have a custom file format that needs special handling, this recipe will show you how to create and use a custom corpus view.
The main corpus view class is StreamBackedCorpusView, which opens a single file as a stream, and maintains an internal cache of blocks it has read.
Blocks of tokens are read in with a block reader function.
In the Creating a part-of-speech tagged word corpus recipe, we discussed the default para_block_reader function of the TaggedCorpusReader class, which reads lines from a file until it finds a blank line, then returns those lines as a single paragraph token.
The actual block reader function is nltk.corpus.reader.util.read_ blankline_block.
The TaggedCorpusReader class passes this block reader function into a TaggedCorpusView class whenever it needs to read blocks from a file.
The TaggedCorpusView class is a subclass of StreamBackedCorpusView that knows to split paragraphs of word/tag into (word, tag) tuples.
We'll start with the simple case of a plain text file with a heading that should be ignored by the corpus reader.
To ignore this heading, we need to subclass the PlaintextCorpusReader class so we can override its CorpusView class variable with our own StreamBackedCorpusView subclass.
The following is the code found in corpus.py: from nltk.corpus.reader import PlaintextCorpusReader from nltk.corpus.reader.util import StreamBackedCorpusView class IgnoreHeadingCorpusView(StreamBackedCorpusView): def __init__(self, *args, **kwargs): StreamBackedCorpusView.__init__(self, *args, **kwargs) # open self._stream self._open() # skip the heading block self.read_block(self._stream) # reset the start position to the current position in the stream self._filepos = [self._stream.tell()] class IgnoreHeadingCorpusReader(PlaintextCorpusReader): CorpusView = IgnoreHeadingCorpusView To demonstrate that this works as expected, here is code showing that the default PlaintextCorpusReader class finds four paragraphs, while our IgnoreHeadingCorpusReader class only has three paragraphs: >>> from nltk.corpus.reader import PlaintextCorpusReader >>> plain = PlaintextCorpusReader('.', ['heading_text.txt']) >>> len(plain.paras()) 4 >>> from corpus import IgnoreHeadingCorpusReader >>> reader = IgnoreHeadingCorpusReader('.', ['heading_text.txt']) >>> len(reader.paras()) 3 76 www.it-ebooks.info Chapter 3 How it works...
Most corpus readers do not have a CorpusView class variable because they require very specific corpus views.
This function is defined by StreamBackedCorpusView, and sets the internal instance variable self._stream to the opened file.
3. Resets the start file position to the current position of self._stream.
The following is a diagram illustrating the relationships between the classes: AbstractLazySequence __len__() iterate_from() CorpusReader StreamBackedCorpusView read_block() PlaintextCorpusReader CorpusView IgnoreHeadingCorpusReader IgnoreHeadingCorpusView CorpusView 77 www.it-ebooks.info Creating Custom Corpora There's more...
Corpus views can get a lot fancier and more complicated, but the core concept is the same: read blocks from a stream to return a list of tokens.
There are a number of block readers provided in nltk.corpus.reader.util, but you can always create your own.
Define it as a separate function and pass it into StreamBackedCorpusView as block_reader.

Creating a MongoDB-backed corpus reader

That is in part due to the design of the CorpusReader base class, and also the assumption that most corpus data will be in text files.
However, sometimes you'll have a bunch of data stored in a database that you want to access and use just like a text file corpus.
In this recipe, we'll cover the case where you have documents in MongoDB, and you want to use a particular field of each document as your block of text.
The following code assumes that your database is on localhost port 27017, which is the MongoDB default configuration, and that you'll be using the test database with a collection named corpus that contains documents with a text field.
Since the CorpusReader class assumes you have a file-based corpus, we can't directly subclass it.
The StreamBackedCorpusView class is a subclass of nltk.util.AbstractLazySequence, so we'll subclass AbstractLazySequence to create a MongoDB view, and then create a new class that will use the view to provide functionality similar to the PlaintextCorpusReader class.
The following is the code, which is found in mongoreader.py: import pymongo from nltk.data import LazyLoader from nltk.tokenize import TreebankWordTokenizer from nltk.util import AbstractLazySequence, LazyMap, LazyConcatenation class MongoDBLazySequence(AbstractLazySequence): def __init__(self, host='localhost', port=27017, db='test', collection='corpus', field='text'): self.conn = pymongo.MongoClient(host, port) self.collection = self.conn[db][collection] self.field = field def __len__(self): return self.collection.count() def iterate_from(self, start): f = lambda d: d.get(self.field, '') return iter(LazyMap(f, self.collection.find(fields= [self.field], skip=start))) class MongoDBCorpusReader(object): def __init__(self, word_tokenizer=TreebankWordTokenizer(), sent_tokenizer=LazyLoader('tokenizers/punkt/PY3 /english.pickle'),**kwargs): self._seq = MongoDBLazySequence(**kwargs) self._word_tokenize = word_tokenizer.tokenize self._sent_tokenize = sent_tokenizer.tokenize def text(self): return self._seq 80 www.it-ebooks.info Chapter 3 def words(self): return LazyConcatenation(LazyMap(self._word_tokenize, self.text())) def sents(self): return LazyConcatenation(LazyMap(self._sent_tokenize, self.text())) How it works...
Subclasses must implement the __len__() and iterate_from(start) methods, while it provides the rest of the list and iterator emulation methods.
By creating the MongoDBLazySequence subclass as our view, we can iterate over documents in the MongoDB collection on demand, without keeping all the documents in memory.
The LazyMap class is a lazy version of Python's built-in map() function, and is used in iterate_from() to transform the document into the specific field that we're interested in.
The MongoDBCorpusReader class creates an internal instance of MongoDBLazySequence for iteration, then defines the word and sentence tokenization methods.
The text() method simply returns the instance of MongoDBLazySequence, which results in a lazily evaluated list of each text field.
The words() method uses LazyMap and LazyConcatenation to return a lazily evaluated list of all words, while the sents() method does the same for sentences.
For example, if you had a db named website, with a collection named comments, whose documents had a field called comment, you could create a MongoDBCorpusReader class as follows: >>> reader = MongoDBCorpusReader(db='website', collection='comments', field='comment') You can also pass in custom instances for word_tokenizer and sent_tokenizer, as long as the objects implement the nltk.tokenize.TokenizerI interface by providing a tokenize(text) method.

Corpus editing with file locking

However, modifying a corpus file while other processes are using it, such as through a corpus reader, can lead to dangerous undefined behavior.
This library provides cross-platform file locking, and so will work on Windows, Unix/Linux, Mac OS X, and more.
Here are two file editing functions: append_line() and remove_line().
An exclusive lock means that these functions will wait until no other process is reading from or writing to the file.
These functions can be found in corpus.py, as follows: import lockfile, tempfile, shutil def append_line(fname, line): with lockfile.FileLock(fname):   fp = open(fname, 'a+')   fp.write(line) fp.write('\n')   fp.close() def remove_line(fname, line): 82 www.it-ebooks.info Chapter 3   with lockfile.FileLock(fname):   tmp = tempfile.TemporaryFile()   fp = open(fname, 'rw+')   # write all lines from orig file, except if matches given line   for l in fp: if l.strip() != line: tmp.write(l)   # reset file pointers so entire files are copied   fp.seek(0)   tmp.seek(0)   # copy tmp into fp, then truncate to remove trailing line(s)   shutil.copyfileobj(tmp, fp)   fp.truncate()   fp.close() tmp.close() The lock acquiring and releasing happens transparently when you do with lockfile.
You can use these functions as follows: >>> from corpus import append_line, remove_line >>> append_line('test.txt', 'foo') >>> remove_line('test.txt', 'foo') In append_line(), a lock is acquired, the file is opened in append mode, the text is written along with an end-of-line character, and then the file is closed, releasing the lock.
A lock acquired by lockfile only protects the file from other processes that also use lockfile.
In other words, just because your Python process has a lock with lockfile doesn't mean a non-Python process can't modify the file.
For this reason, it's best to only use lockfile with files that will not be edited by an non-Python processes, or Python processes that do not use lockfile.
83 www.it-ebooks.info Creating Custom Corpora The remove_line() function is a bit more complicated.
The remove_line() function does not work on Mac OS X, but does work on Linux.
For remove_line() to work, it must be able to open a file in both read and write modes, and Mac OS X does not allow this.
These functions are best suited for a wordlist corpus, or some other corpus type with presumably unique lines, that may be edited by multiple people at about the same time,   such as through a web interface.

Part-of-speech Tagging

No text here

Introduction

No text here

Default tagging

Default tagging provides a baseline for part-of-speech tagging.
Getting ready We're going to use the treebank corpus for most of this chapter because it's a common standard and is quick to load and test.
The DefaultTagger class takes a single argument, the tag you want to apply.
DefaultTagger is most useful when you choose the most common part-of-speech tag.
86 www.it-ebooks.info Chapter 4 >>> from nltk.tag import DefaultTagger >>> tagger = DefaultTagger('NN') >>> tagger.tag(['Hello', 'World']) [('Hello', 'NN'), ('World', 'NN')] Every tagger has a tag() method that takes a list of tokens, where each token is a single word.
Here's a diagram showing the inheritance tree: TaggerI tag() evaluate() SequentialBackoffTagger choose_tag() DefaultTagger The choose_tag() method of DefaultTagger is very simple: it returns the tag we gave it at the time of initialization.
87 www.it-ebooks.info Part-of-speech Tagging There's more...
You can find a complete list of possible tags for the treebank corpus at http://www.ling.upenn.
These tags are also documented in Appendix, Penn Treebank Part-of-speech Tags.
Evaluating accuracy To know how accurate a tagger is, you can use the evaluate() method, which takes a list of tagged tokens as a gold standard to evaluate the tagger.
Using our default tagger created earlier, we can evaluate it against a subset of the treebank corpus tagged sentences.
>>> from nltk.corpus import treebank >>> test_sents = treebank.tagged_sents()[3000:] >>> tagger.evaluate(test_sents) 0.14331966328512843 So, by just choosing NN for every tag, we can achieve 14 % accuracy testing on one-fourth of the treebank corpus.
Of course, accuracy will be different if you choose a different default tag.
Tagging sentences TaggerI also implements a tag_sents() method that can be used to tag a list of sentences, instead of a single sentence.
Here's an example of tagging two simple sentences: >>> tagger.tag_sents([['Hello', 'world', '.'], ['How', 'are', 'you', '?']]) [[('Hello', 'NN'), ('world', 'NN'), ('.', 'NN')], [('How', 'NN'), ('are', 'NN'), ('you', 'NN'), ('?', 'NN')]] The result is a list of two tagged sentences, and of course, every tag is NN because we're using the DefaultTagger class.
The tag_sents() method can be quiet useful if you have many sentences you wish to tag all at once.
Calling this function with a tagged sentence will return a list of words without the tags.
>>> from nltk.tag import untag >>> untag([('Hello', 'NN'), ('World', 'NN')]) ['Hello', 'World'] 88 www.it-ebooks.info

Training a unigram part-of-speech tagger

Therefore, a unigram tagger only uses a single word as its context for determining the part-of-speech tag.
UnigramTagger can be trained by giving it a list of tagged sentences at initialization.
89 www.it-ebooks.info Part-of-speech Tagging How it works...
UnigramTagger builds a context model from the list of tagged sentences.
Because UnigramTagger inherits from ContextTagger, instead of providing a choose_tag() method, it must implement a context() method, which takes the same three arguments as choose_tag().
Here's an inheritance diagram showing each class, starting at SequentialBackoffTagger:
SequentialBackoffTagger choose_tag() ContextTagger context() NgramTagger UnigramTagger Let's see how accurate the UnigramTagger class is on the test sentences (see the previous recipe for how test_sents is created).
>>> tagger.evaluate(test_sents) 0.8588819339520829 It has almost 86 % accuracy for a tagger that only uses single word lookup to determine the part-of-speech tag.
Given the list of tagged sentences, it calculates the frequency that a tag has occurred for each context.
Overriding the context model All taggers that inherit from ContextTagger can take a pre-built model instead of training their own.
This model is simply a Python dict mapping a context key to a tag.
Here's an example where we pass a very simple model to the UnigramTagger class instead of a training set.
>>> tagger = UnigramTagger(model={'Pierre': 'NN'}) >>> tagger.tag(treebank.sents()[0]) [('Pierre', 'NN'), ('Vinken', None), (',', None), ('61', None), ('years', None), ('old', None), (',', None), ('will', None), ('join', None), ('the', None), ('board', None), ('as', None), ('a', None), ('nonexecutive', None), ('director', None), ('Nov.', None), ('29', None), ('.', None)] Since the model only contained the context key Pierre, only the first word got a tag.
So, unless you know exactly what you are doing, let the tagger train its own model instead of passing in your own.
One good case for passing a self-created model to the UnigramTagger class is for when you have a dictionary of words and tags, and you know that every word should always map to its tag.
Then, you can put this UnigramTagger as your first backoff tagger (covered in the next recipe) to look up tags for unambiguous words.
Minimum frequency cutoff The ContextTagger class uses frequency of occurrence to decide which tag is most likely for a given context.
By default, it will do this even if the context word and tag occurs only once.
If you'd like to set a minimum frequency threshold, then you can pass a cutoff value to the UnigramTagger class.
>>> tagger = UnigramTagger(train_sents, cutoff=3) >>> tagger.evaluate(test_sents) 0.7757392618173969 In this case, using cutoff=3 has decreased accuracy, but there may be times when a cutoff is a good idea.

Combining taggers with backoff tagging

It allows you to chain taggers together so that if one tagger doesn't know how to tag a word, it can pass the word on to the next backoff tagger.
So, we'll use the DefaultTagger class from the Default tagging recipe in this chapter as the backoff to the UnigramTagger class covered in the previous recipe, Training a unigram part-of-speech tagger.
>>> tagger1 = DefaultTagger('NN') >>> tagger2 = UnigramTagger(train_sents, backoff=tagger1) >>> tagger2.evaluate(test_sents) 0.8758471832505935 By using a default tag of NN whenever the UnigramTagger is unable to tag a word, we've increased the accuracy by almost 2%!
When a SequentialBackoffTagger class is initialized, it creates an internal list of backoff taggers with itself as the first element.
Here's some code to illustrate this: >>> tagger1._taggers == [tagger1] True >>> tagger2._taggers == [tagger2, tagger1] True 92 www.it-ebooks.info Chapter 4 The _taggers list is the internal list of backoff taggers that the SequentialBackoffTagger class uses when the tag() method is called.
It goes through its list of taggers, calling choose_tag() on each one.
This means that if the primary tagger can tag the word, then that's the tag that will be returned.
Of course, None will never be returned if your final backoff tagger is a DefaultTagger.
There's a few taggers that we'll cover in the later recipes that cannot be used as part of a backoff tagging chain, such as the BrillTagger class.
However, these taggers generally take another tagger to use as a baseline, and a SequentialBackoffTagger class is often a good choice for that baseline.
See also In the next recipe, we'll combine more taggers with backoff tagging.
Also, see the previous two recipes for details on the DefaultTagger and UnigramTagger classes.

Training and combining ngram taggers

In addition to UnigramTagger, there are two more NgramTagger subclasses: BigramTagger and TrigramTagger.
An ngram is a subsequence of n items, so the BigramTagger subclass looks at two items (the previous tagged word and the current word), and the TrigramTagger subclass looks at three items.
These two taggers are good at handling words whose part-of-speech tag is context-dependent.
The idea with the NgramTagger subclasses is that by looking at the previous words and part-of-speech tags, we can better guess the part-of-speech tag for the current word.
In the case of NgramTagger subclasses, the context is some number of previous tagged words.
Since a UnigramTagger class doesn't care about the previous context, it is able to have higher baseline accuracy by simply guessing the most common tag for each word.
>>> from nltk.tag import BigramTagger, TrigramTagger >>> bitagger = BigramTagger(train_sents) >>> bitagger.evaluate(test_sents) 0.11310166199007123 >>> tritagger = TrigramTagger(train_sents) >>> tritagger.evaluate(test_sents) 0.0688107058061731 94 www.it-ebooks.info Chapter 4 Where BigramTagger and TrigramTagger can make a contribution is when we combine them with backoff tagging.
This time, instead of creating each tagger individually, we'll create a function that will take train_sents, a list of SequentialBackoffTagger classes, and an optional final backoff tagger, then train each tagger with the previous tagger as a backoff.
Here's the code from tag_util.py: def backoff_tagger(train_sents, tagger_classes, backoff=None): for cls in tagger_classes: backoff = cls(train_sents, backoff=backoff) return backoff And to use it, we can do the following: >>> from tag_util import backoff_tagger >>> backoff = DefaultTagger('NN') >>> tagger = backoff_tagger(train_sents, [UnigramTagger, BigramTagger, TrigramTagger], backoff=backoff) >>> tagger.evaluate(test_sents) 0.8806820634578028 So, we've gained almost 1% accuracy by including the BigramTagger and TrigramTagger subclasses in the backoff chain.
The backoff_tagger function creates an instance of each tagger class in the list, giving it train_sents and the previous tagger as a backoff.
The order of the list of tagger classes is quite important: the first class in the list (UnigramTagger) will be trained first and given the initial backoff tagger (the DefaultTagger).
The final tagger returned will be an instance of the last tagger class in the list (TrigramTagger).
95 www.it-ebooks.info Part-of-speech Tagging There's more...
The backoff_tagger function doesn't just work with NgramTagger classes, it can also be used for constructing a chain containing any subclasses of SequentialBackoffTagger.
BigramTagger and TrigramTagger, because they are subclasses of NgramTagger and ContextTagger, can also take a model and cutoff argument, just like the UnigramTagger.
Quadgram tagger The NgramTagger class can be used by itself to create a tagger that uses more than three ngrams for its context key.
>>> from nltk.tag import NgramTagger >>> quadtagger = NgramTagger(4, train_sents) >>> quadtagger.evaluate(test_sents) 0.058234405352903085 It's even worse than the TrigramTagger!
Here's an alternative implementation of a QuadgramTagger class that we can include in a list to backoff_tagger.
from nltk.tag import NgramTagger class QuadgramTagger(NgramTagger): def __init__(self, *args, **kwargs): NgramTagger.__init__(self, 4, *args, **kwargs) This is essentially how BigramTagger and TrigramTagger are implemented: simple subclasses of NgramTagger that pass in the number of ngrams to look at in the history argument of the context() method.
>>> from taggers import QuadgramTagger >>> quadtagger = backoff_tagger(train_sents,

Creating a model of likely word tags

As previously mentioned in the Training a unigram part-of-speech tagger recipe, using a custom model with a UnigramTagger class should only be done if you know exactly what you're doing.
In this recipe, we're going to create a model for the most common words, most of which always have the same tag no matter what.
To find the most common words, we can use nltk.probability.FreqDist to count word frequencies in the treebank corpus.
Then, we can create a ConditionalFreqDist class for tagged words, where we count the frequency of every tag for every word.
Using these counts, we can construct a model of the 200 most frequent words as keys, with the most frequent tag for each word as a value.
from nltk.probability import FreqDist, ConditionalFreqDist def word_tag_model(words, tagged_words, limit=200): fd = FreqDist(words) cfd = ConditionalFreqDist(tagged_words) most_freq = (word for word, count in fd.most_common(limit)) return dict((word, cfd[word].max()) for word in most_freq) And to use it with a UnigramTagger class, we can do the following: >>> from tag_util import word_tag_model >>> from nltk.corpus import treebank >>> model = word_tag_model(treebank.words(), treebank.tagged_words()) >>> tagger = UnigramTagger(model=model) >>> tagger.evaluate(test_sents) 0.559680552557738 An accuracy of almost 56% is ok, but nowhere near as good as the trained UnigramTagger.
>>> default_tagger = DefaultTagger('NN') >>> likely_tagger = UnigramTagger(model=model, backoff=default_tagger) >>> tagger = backoff_tagger(train_sents, [UnigramTagger, BigramTagger, TrigramTagger], backoff=likely_tagger) >>> tagger.evaluate(test_sents) 0.8806820634578028 97 www.it-ebooks.info Part-of-speech Tagging The final accuracy is exactly the same as without the likely_tagger.
This is because the frequency calculations we did to create the model are almost exactly the same as what happens when we train a UnigramTagger class.
We give the list of words to a FreqDist class, which counts the frequency of each word.
Then, we get the top 200 words from the FreqDist class by calling fd.most_common(), which obviously returns a list of the most common words and counts.
But in NLTK3, FreqDist inherits from collections.Counter, and the keys() method does not use any predictable ordering.
And by putting the likely_tagger at the front of the chain, we can actually improve accuracy a little bit: >>> tagger = backoff_tagger(train_sents, [UnigramTagger, BigramTagger, TrigramTagger], backoff=default_tagger) >>> likely_tagger = UnigramTagger(model=model, backoff=tagger) >>> likely_tagger.evaluate(test_sents) 0.8824088063889488 Putting custom model taggers at the front of the backoff chain gives you complete control over how specific words are tagged, while letting the trained taggers handle everything else.

Tagging with regular expressions

You can use regular expression matching to tag words.
Or you could match on known word patterns, such as the suffix "ing".
The RegexpTagger class expects a list of two tuples, where the first element in the tuple is a regular expression and the second element is the tag.
The patterns shown in the following code can be found in tag_util.py: patterns = [(r'^\d+$', 'CD'), (r'.*ing$', 'VBG'), # gerunds, i.e. wondering (r'.*ment$', 'NN'), # i.e. wonderment (r'.*ful$', 'JJ') # i.e. wonderful] Once you've constructed this list of patterns, you can pass it into RegexpTagger.
For example, it could be positioned just before a DefaultTagger class, to tag words that the ngram tagger(s) missed.

Affix tagging

The AffixTagger class is another ContextTagger subclass, but this time the context is either the prefix or the suffix of a word.
This means the AffixTagger class is able to learn tags based on fixed-length substrings of the beginning or ending of a word.
The default arguments for an AffixTagger class specify three-character suffixes, and that words must be at least five characters long.
>>> prefix_tagger = AffixTagger(train_sents, affix_length=3) >>> prefix_tagger.evaluate(test_sents) 0.23587308439456076 100 www.it-ebooks.info Chapter 4 To learn on two-character suffixes, the code will look like this: >>> suffix_tagger = AffixTagger(train_sents, affix_length=-2) >>> suffix_tagger.evaluate(test_sents) 0.31940427368875457 How it works...
A positive value for affix_length means that the AffixTagger class will learn word prefixes, essentially word[:affix_length].
If affix_length is negative, then suffixes are learned using word[affix_length:].
You can combine multiple affix taggers in a backoff chain if you want to learn on multiple character length affixes.

Training a Brill tagger

The BrillTagger class is a transformation-based tagger.
Instead, the BrillTagger class uses a series of rules to correct the results of an initial tagger.
These rules are scored based on how many errors they correct minus the number of new errors they produce.
Here's a function from tag_util.py that trains a BrillTagger class using BrillTaggerTrainer.
from nltk.tag import brill, brill_trainer def train_brill_tagger(initial_tagger, train_sents, **kwargs):   templates = [brill.Template(brill.Pos([-1])),   brill.Template(brill.Pos([1])),   brill.Template(brill.Pos([-2])),   brill.Template(brill.Pos([2])),   brill.Template(brill.Pos([-2, -1])),   brill.Template(brill.Pos([1, 2])),   brill.Template(brill.Pos([-3, -2, -1])),   brill.Template(brill.Pos([1, 2, 3])),   brill.Template(brill.Pos([-1]), brill.Pos([1])),   brill.Template(brill.Word([-1])),   brill.Template(brill.Word([1])), 102 www.it-ebooks.info Chapter 4   brill.Template(brill.Word([-2])),   brill.Template(brill.Word([2])),   brill.Template(brill.Word([-2, -1])),   brill.Template(brill.Word([1, 2])),   brill.Template(brill.Word([-3, -2, -1])),   brill.Template(brill.Word([1, 2, 3])),   brill.Template(brill.Word([-1]), brill.Word([1])),   ]   trainer = brill_trainer.BrillTaggerTrainer(initial_tagger, templates, deterministic=True)   return trainer.train(train_sents, **kwargs) To use it, we can create our initial_tagger from a backoff chain of NgramTagger classes, then pass that into the train_brill_tagger() function to get a BrillTagger back.
The BrillTaggerTrainer class takes an initial_tagger argument and a list of templates.
The brill.Template class is such an implementation, and is actually imported from nltk.tbl.template.
The brill.Pos and brill.Word classes are subclasses of nltk.tbl.template.Feature, and they describe what kind of features to use in the template, in this case, one or more part-of-speech tags or words.
Template(brill.Pos([-1])) means that a rule can be generated using the previous part-of-speech tag.
The brill.Template(brill.Pos([1])) statement means that you can look at the next part-of-speech tag to generate a rule.
103 www.it-ebooks.info Part-of-speech Tagging The thinking behind a transformation-based tagger is this: given the correct training sentences, the output of the initial tagger, and the templates specifying features, try to generate transformation rules that correct the initial tagger's output to be more in-line with the training sentences.
You can control the number of rules generated using the max_rules keyword argument   to the BrillTaggerTrainer.train() method.
The default value is 200.
The default value   is 2, though 3 can be a good choice as well.
The score is a measure of how well a rule   corrects errors compared to how many new errors it introduces.
Tracing You can watch the BrillTaggerTrainer class do its work by passing trace=True into the constructor, for example, trainer = brill.BrillTaggerTrainer(initial_tagger, templates, deterministic=True, trace=True).
This will give you the following output: TBL train (fast) (seqs: 3000; tokens: 77511; tpls: 18; min score: 2; min acc: None)   Finding initial useful rules...

Training the TnT tagger

>>> from nltk.tag import tnt >>> tnt_tagger = tnt.TnT() >>> tnt_tagger.train(train_sents) >>> tnt_tagger.evaluate(test_sents) 0.8756313403842003 It's quite a good tagger all by itself, only slightly less accurate than the BrillTagger class from the previous recipe.
The TnT tagger maintains a number of internal FreqDist and ConditionalFreqDist instances based on the training data.
Then, during tagging, the frequencies are used to calculate the probabilities of possible tags for each word.
So, instead of constructing a backoff chain of NgramTagger subclasses, the TnT tagger uses all the ngram models together to choose the best tag.
It also tries to guess the tags for the whole sentence at once by choosing the most likely model for the entire sentence, based on the probabilities of each possible tag.
105 www.it-ebooks.info Part-of-speech Tagging Training is fairly quick, but tagging is significantly slower than the other taggers we've covered.
You can pass in a tagger for unknown words as unk.
If this tagger is already trained, then you must also pass in Trained=True.
Otherwise, it will call unk.train(data) with the same data you pass into the train() method.
Since none of the previous taggers have a public train() method, I recommend always passing Trained=True if you also pass an unk tagger.
>>> from nltk.tag import DefaultTagger >>> unk = DefaultTagger('NN') >>> tnt_tagger = tnt.TnT(unk=unk, Trained=True) >>> tnt_tagger.train(train_sents) >>> tnt_tagger.evaluate(test_sents) 0.892467083962875 So, we got an almost 2% increase in accuracy!
You must use a tagger that can tag a single word without having seen that word before.
This is because the unknown tagger's tag() method is only called with a single word sentence.
Passing in a UnigramTagger class that's been trained on the same data is pretty much useless, as it will have seen the exact same words and, therefore, have the same unknown word blind spots.
Controlling the beam search Another parameter you can modify for TnT is N, which controls the number of possible solutions the tagger maintains while trying to guess the tags for a sentence.
Increasing it will greatly increase the amount of memory used during tagging, without necessarily increasing the accuracy.
>>> tnt_tagger = tnt.TnT(N=100) >>> tnt_tagger.train(train_sents) >>> tnt_tagger.evaluate(test_sents) 0.8756313403842003 So, the accuracy is exactly the same, but we use significantly less memory to achieve it.

Using WordNet for tagging

If you remember from the Looking up Synsets for a word in WordNet recipe in Chapter 1, Tokenizing Text and WordNet Basics, WordNet Synsets specify a part-of-speech tag.
It's a   very restricted set of possible tags, and many words have multiple Synsets with different part-of-speech tags, but this information can be useful for tagging unknown words.
Getting ready First, we need to decide how to map WordNet part-of-speech tags to the Penn Treebank part-of-speech tags we've been using.
See the Looking up Synsets for a word in WordNet recipe in Chapter 1, Tokenizing Text and WordNet Basics, for more details.
WordNet tag Treebank tag n NN a JJ s JJ r RB v VB 107 www.it-ebooks.info Part-of-speech Tagging How to do it...
Now we can create a class that will look up words in WordNet, and then choose the most common tag from the Synsets it finds.
The WordNetTagger class defined in the following code can be found in taggers.py: from nltk.tag import SequentialBackoffTagger from nltk.corpus import wordnet from nltk.probability import FreqDist class WordNetTagger(SequentialBackoffTagger):   '''   >>> wt = WordNetTagger()   >>> wt.tag(['food', 'is', 'great'])   [('food', 'NN'), ('is', 'VB'), ('great', 'JJ')]   '''   def __init__(self, *args, **kwargs):   SequentialBackoffTagger.__init__(self, *args, **kwargs)   self.wordnet_tag_map = {   'n': 'NN',   's': 'JJ',   'a': 'JJ',   'r': 'RB',   'v': 'VB'   }   def choose_tag(self, tokens, index, history):   word = tokens[index]   fd = FreqDist()   for synset in wordnet.synsets(word):   fd[synset.pos()] += 1   return self.wordnet_tag_map.get(fd.max()) 108 www.it-ebooks.info Chapter 4 Another way the FreqDist API has changed between NLTK2 and NLTK3 is that the inc() method has been removed.
The WordNetTagger class simply counts the number of each part-of-speech tag found in the Synsets for a word.
We only have enough information to produce four different kinds of tags, while there are 36 possible tags in treebank.
There are many words that can have different part-of-speech tags depending on their context.
>>> from tag_util import backoff_tagger >>> from nltk.tag import UnigramTagger, BigramTagger, TrigramTagger >>> tagger = backoff_tagger(train_sents, [UnigramTagger, BigramTagger, TrigramTagger], backoff=wn_tagger) >>> tagger.evaluate(test_sents) 0.8848262464925534 See also The Looking up Synsets for a word in WordNet recipe in Chapter 1, Tokenizing Text and WordNet Basics, details how to use the wordnet corpus and what kinds of part-of-speech tags it knows about.

Tagging proper names

Using the included names corpus, we can create a simple tagger for tagging names as proper nouns. The NamesTagger class is a subclass of SequentialBackoffTagger as it's probably only useful near the end of a backoff chain.
If it isn't, we return None, so the next tagger in the chain can tag the word.
The following code can be found in taggers.py: from nltk.tag import SequentialBackoffTagger from nltk.corpus import names class NamesTagger(SequentialBackoffTagger): def __init__(self, *args, **kwargs): SequentialBackoffTagger.__init__(self, *args, **kwargs) self.name_set = set([n.lower() for n in names.words()]) def choose_tag(self, tokens, index, history): word = tokens[index] if word.lower() in self.name_set: return 'NNP' else: return None How it works...
>>> from taggers import NamesTagger >>> nt = NamesTagger() >>> nt.tag(['Jacob']) [('Jacob', 'NNP')]
It's probably best to use the NamesTagger class right before a DefaultTagger class, so it's at the end of a backoff chain.
But it could probably go anywhere in the chain since it's unlikely to mis-tag a word.

Classifier-based tagging

The ClassifierBasedPOSTagger class uses classification to do part-of-speech tagging.
The ClassifierBasedPOSTagger class is a subclass of ClassifierBasedTagger that implements a feature detector that combines many of the techniques of the previous taggers into a single feature set.
The feature detector finds multiple length suffixes, does some regular expression matching, and looks at the unigram, bigram, and trigram history to produce a fairly complete set of features for each word.
The feature sets it produces are used to train the internal classifier, and are used for classifying words into part-of-speech tags.
>>> from nltk.tag.sequential import ClassifierBasedPOSTagger >>> tagger = ClassifierBasedPOSTagger(train=train_sents) >>> tagger.evaluate(test_sents) 0.9309734513274336
Notice a slight modification to initialization: train_sents must be passed in as the train keyword argument.
The ClassifierBasedPOSTagger class inherits from ClassifierBasedTagger and only implements a feature_detector() method.
Once this classifier is trained, it is used to classify word features produced by the feature_detector() method.
111 www.it-ebooks.info Part-of-speech Tagging The ClassifierBasedTagger class is often the most accurate tagger, but it's also one of the slowest taggers.
For example, to use a MaxentClassifier, you'd do the following: >>> from nltk.classify import MaxentClassifier >>> me_tagger = ClassifierBasedPOSTagger(train=train_sents, classifier_builder=MaxentClassifier.train) >>> me_tagger.evaluate(test_sents) 0.9258363911072739
The MaxentClassifier class takes even longer to train than NaiveBayesClassifier.
112 www.it-ebooks.info Chapter 4 Detecting features with a custom feature detector If you want to do your own feature detection, there are two ways to do it: 1.
Subclass ClassifierBasedTagger and implement a feature_detector() method.
2. Pass a function as the feature_detector keyword argument into ClassifierBasedTagger at initialization.
A very simple example would be a unigram feature detector (found in tag_util.py).
def unigram_feature_detector(tokens, index, history): return {'word': tokens[index]} Then, using the second method, you'd pass this into ClassifierBasedTagger as feature_detector.
>>> from nltk.tag.sequential import ClassifierBasedTagger >>> from tag_util import unigram_feature_detector >>> tagger = ClassifierBasedTagger(train=train_sents, feature_ detector=unigram_feature_detector) >>> tagger.evaluate(test_sents) 0.8733865745737104 Setting a cutoff probability Because a classifier will always return the best result it can, passing in a backoff tagger is useless unless you also pass in a cutoff_prob argument to specify the probability threshold for classification.
Here's an example using the DefaultTagger class as the backoff, and setting cutoff_prob to 0.3: >>> default = DefaultTagger('NN') >>> tagger = ClassifierBasedPOSTagger(train=train_sents, backoff=default, cutoff_prob=0.3) >>> tagger.evaluate(test_sents) 0.9311029570472696 So, we get a slight increase in accuracy if the ClassifierBasedPOSTagger class uses the DefaultTagger class whenever its tag probability is less than 30%.