Tokenizing Text and WordNet Basics

No text here

Introduction

Natural Language ToolKit (NLTK) is a comprehensive Python library for natural language processing and text analytics. NLTK is often used for rapid prototyping of text processing programs and can even be used in production applications. Demos of select NLTK functionality and production-ready APIs are available at http://text-processing.com.

Tokenizing text into sentences

We'll start with sentence tokenization, or splitting a paragraph into a list of sentences. If you've used earlier versions of NLTK (such as version 2.0), note that some of the APIs have changed in Version 3 and are not backwards compatible. Once you've installed NLTK, you'll also need to install the data following the instructions at http://nltk.org/data.html. 8 www.it-ebooks.info Chapter 1 How to do it... Once NLTK is installed and you have a Python console running, we can start by creating a paragraph of text: >>> para = "Hello World. First we need to import the sentence tokenization function, and then we can call it with the paragraph as an argument: >>> from nltk.tokenize import sent_tokenize >>> sent_tokenize(para) ['Hello World. ', "It's good to see you. ", 'Thanks for buying this book.'] So if you're going to be tokenizing a lot of sentences, it's more efficient to load the PunktSentenceTokenizer class once, and call its tokenize() method instead: >>> import nltk.data >>> tokenizer = nltk.data.load('tokenizers/punkt/PY3/english.pickle') >>> tokenizer.tokenize(para) ['Hello World. ', "It's good to see you. ", 'Thanks for buying this book.']

Tokenizing sentences into words

Basic word tokenization is very simple; use the word_tokenize() function: >>> from nltk.tokenize import word_tokenize >>> word_tokenize('Hello World.') ['Hello', 'World', '.'] It's equivalent to the following code: >>> from nltk.tokenize import TreebankWordTokenizer >>> tokenizer = TreebankWordTokenizer() >>> tokenizer.tokenize('Hello World.') ['Hello', 'World', '.'] The inheritance tree looks like what's shown in the following diagram: TokenizerI tokenize(s) PunktWordTokenizer TreebankWordTokenizer RegexpTokenizer WordPunctTokenizer WhitespaceTokenizer Separating contractions The TreebankWordTokenizer class uses conventions found in the Penn Treebank corpus. For example, consider the following code: >>> word_tokenize("can't") ['ca', "n't"] If you find this convention unacceptable, then read on for alternatives, and see the next recipe for tokenizing with regular expressions.

Tokenizing sentences using regular expressions

12 www.it-ebooks.info Chapter 1 How to do it... We'll create an instance of RegexpTokenizer, giving it a regular expression string to use for matching tokens: >>> from nltk.tokenize import RegexpTokenizer >>> tokenizer = RegexpTokenizer("[\w']+") >>> tokenizer.tokenize("Can't is a contraction.") ["Can't", 'is', 'a', 'contraction'] There's also a simple helper function you can use if you don't want to instantiate the class, as shown in the following code: >>> from nltk.tokenize import regexp_tokenize >>> regexp_tokenize("Can't is a contraction. ", "[\w']+") ["Can't", 'is', 'a', 'contraction'] Now we finally have something that can treat contractions as whole words, instead of splitting them into tokens. Simple whitespace tokenizer The following is a simple example of using RegexpTokenizer to tokenize on whitespace: >>> tokenizer = RegexpTokenizer('\s+', gaps=True) >>> tokenizer.tokenize("Can't is a contraction.")

Training a sentence tokenizer

Here's an example of training a sentence tokenizer on dialog text, using overheard.txt from the webtext corpus: >>> from nltk.tokenize import PunktSentenceTokenizer >>> from nltk.corpus import webtext >>> text = webtext.raw('overheard.txt') >>> sent_tokenizer = PunktSentenceTokenizer(text) 14 www.it-ebooks.info Chapter 1 Let's compare the results to the default sentence tokenizer, as follows: >>> sents1 = sent_tokenizer.tokenize(text) >>> sents1[0] 'White guy: So, do you have any plans for this evening?' >>> from nltk.tokenize import sent_tokenize >>> sents2 = sent_tokenize(text) >>> sents2[0] 'White guy: So, do you have any plans for this evening?' >>> sents1[678] 'Girl: But you already have a Big Mac...' >>> sents2[678] 'Girl: But you already have a Big Mac...\\nHobo: Oh, this is all theatrical.' This difference is a good demonstration of why it can be useful to train your own sentence tokenizer, especially when your text isn't in the typical paragraph-sentence structure.

Filtering stopwords in a tokenized sentence

16 www.it-ebooks.info Chapter 1 How to do it... We're going to create a set of all English stopwords, then use it to filter stopwords from a sentence with the help of the following code: >>> from nltk.corpus import stopwords >>> english_stops = set(stopwords.words('english')) >>> words = ["Can't", 'is', 'a', 'contraction'] >>> [word for word in words if word not in english_stops] ["Can't", 'contraction'] How it works... As such, it has a words() method that can take a single argument for the file ID, which in this case is 'english', referring to a file containing a list of English stopwords. There's more... You can see the list of all English stopwords using stopwords.words('english') or by examining the word list file at nltk_data/corpora/stopwords/english. You can see the complete list of languages using the fileids method as follows: >>> stopwords.fileids() ['danish', 'dutch', 'english', 'finnish', 'french', 'german', 'hungarian', 'italian', 'norwegian', 'portuguese', 'russian', 'spanish', 'swedish', 'turkish'] Any of these fileids can be used as an argument to the words() method to get a list of stopwords for that language. For example: >>> stopwords.words('dutch') ['de', 'en', 'van', 'ik', 'te', 'dat', 'die', 'in', 'een', 'hij', 'het', 'niet', 'zijn', 'is', 'was', 'op', 'aan', 'met', 'als', 'voor', 'had', 'er', 'maar', 'om', 'hem', 'dan', 'zou', 'of', 'wat', 'mijn', 'men', 'dit', 'zo', 'door', 'over', 'ze', 'zich', 'bij', 'ook', 'tot', 'je', 'mij', 'uit', 'der', 'daar', 'haar', 'naar', 'heb', 'hoe', 'heeft', 'hebben', 'deze', 'u', 'want', 'nog', 'zal', 'me', 'zij', 'nu', 'ge', 'geen', 'omdat', 'iets', 'worden', 'toch', 'al', 'waren', 'veel', 'meer', 'doen', 'toen', 'moet', 'ben', 'zonder', 'kan', 'hun', 'dus', 'alles', 'onder', 'ja', 'eens', 'hier', 'wie', 'werd', 'altijd', 'doch', 'wordt', 'wezen', 'kunnen', 'ons', 'zelf', 'tegen', 'na', 'reeds', 'wil', 'kon', 'niets', 'uw', 'iemand', 'geweest', 'andere'] 17 www.it-ebooks.info

Looking up Synsets for a word in WordNet

How to do it... Now we're going to look up the Synset for cookbook, and explore some of the properties and methods of a Synset using the following code: >>> from nltk.corpus import wordnet >>> syn = wordnet.synsets('cookbook')[0] >>> syn.name() 'cookbook.n.01' >>> syn.definition() 'a book of recipes and cooking directions' How it works... You can look up any word in WordNet using wordnet.synsets(word) to get a list of Synsets. The name() method will give you a unique name for the Synset, which you can use to get the Synset directly: >>> wordnet.synset('cookbook.n.01') Synset('cookbook.n.01') The definition() method should be self-explanatory. Some Synsets also have an examples() method, which contains a list of phrases that use the word in context: >>> wordnet.synsets('cooking')[0].examples() ['cooking can be a great art', 'people are needed who have experience in cookery', 'he left the preparation of meals to his wife'] Working with hypernyms Synsets are organized in a structure similar to that of an inheritance tree. The Calculating WordNet Synset similarity recipe details the functions used to calculate the similarity based on the distance between two words in the hypernym tree: >>> syn.hypernyms() [Synset('reference_book.n.01')] >>> syn.hypernyms()[0].hyponyms() [Synset('annual.n.02'), Synset('atlas.n.02'), Synset('cookbook.n.01'), Synset('directory.n.01'), Synset('encyclopedia.n.01'), Synset('handbook.n.01'), Synset('instruction_book.n.01'), Synset('source_book.n.01'), Synset('wordbook.n.01')] >>> syn.root_hypernyms() [Synset('entity.n.01')] As you can see, reference_book is a hypernym of cookbook, but cookbook is only one of the many hyponyms of reference_book. You can trace the entire path from entity down to cookbook using the hypernym_paths() method, as follows: >>> syn.hypernym_paths() [[Synset('entity.n.01'), Synset('physical_entity.n.01'), Synset('object.n.01'), Synset('whole.n.02'), Synset('artifact.n.01'), Synset('creation.n.02'), Synset('product.n.02'), Synset('work.n.02'), Synset('publication.n.01'), Synset('book.n.01'), Synset('reference_ book.n.01'), Synset('cookbook.n.01')]]

Looking up lemmas and synonyms in WordNet

In the following code, we'll find that there are two lemmas for the cookbook Synset using the lemmas() method: >>> from nltk.corpus import wordnet >>> syn = wordnet.synsets('cookbook')[0] >>> lemmas = syn.lemmas() >>> len(lemmas) 2 >>> lemmas[0].name() 'cookbook' >>> lemmas[1].name() 'cookery_book' >>> lemmas[0].synset() == lemmas[1].synset() True How it works... As you can see, cookery_book and cookbook are two distinct lemmas in the same Synset. So if you wanted to get all synonyms for a Synset, you could do the following: >>> [lemma.name() for lemma in syn.lemmas()] ['cookbook', 'cookery_book'] All possible synonyms As mentioned earlier, many words have multiple Synsets because the word can have different meanings depending on the context. But, let's say you didn't care about the context, and wanted to get all the possible synonyms for a word: >>> synonyms = [] >>> for syn in wordnet.synsets('book'): ... for lemma in syn.lemmas(): ... synonyms.append(lemma.name()) >>> len(synonyms) 38 21 www.it-ebooks.info Tokenizing Text and WordNet Basics As you can see, there appears to be 38 possible synonyms for the word 'book'. If, instead, we take the set of synonyms, there are fewer unique words, as shown in the following code: >>> len(set(synonyms)) 25 Antonyms Some lemmas also have antonyms. The word good, for example, has 27 Synsets, five of which have lemmas with antonyms, as shown in the following code: >>> gn2 = wordnet.synset('good.n.02') >>> gn2.definition() 'moral excellence or admirableness' >>> evil = gn2.lemmas()[0].antonyms()[0] >>> evil.name 'evil' >>> evil.synset().definition() 'the quality of being morally wrong in principle or practice' >>> ga1 = wordnet.synset('good.a.01') >>> ga1.definition() 'having desirable or positive qualities especially those suitable for a thing specified' >>> bad = ga1.lemmas()[0].antonyms()[0] >>> bad.name() 'bad' >>> bad.synset().definition() 'having undesirable or negative qualities' The antonyms() method returns a list of lemmas.

Calculating WordNet Synset similarity

This seems intuitively very similar to a cookbook, so let's see what WordNet similarity has to say about it with the help of the following code: >>> from nltk.corpus import wordnet >>> cb = wordnet.synset('cookbook.n.01') >>> ib = wordnet.synset('instruction_book.n.01') >>> cb.wup_similarity(ib) 0.9166666666666666 So they are over 91% similar! One of the core metrics used to calculate similarity is the shortest path distance between the two Synsets and their common hypernym: >>> ref = cb.hypernyms()[0] >>> cb.shortest_path_distance(ref) 1 >>> ib.shortest_path_distance(ref) 1 >>> cb.shortest_path_distance(ib) 2 So cookbook and instruction_book must be very similar, because they are only one step away from the same reference_book hypernym, and, therefore, only two steps away from each other. >>> dog = wordnet.synsets('dog')[0] >>> dog.wup_similarity(cb) 0.38095238095238093 Wow, dog and cookbook are apparently 38% similar! This is because they share common hypernyms further up the tree: >>> sorted(dog.common_hypernyms(cb)) [Synset('entity.n.01'), Synset('object.n.01'), Synset('physical_ entity.n.01'), Synset('whole.n.02')] Comparing verbs The previous comparisons were all between nouns, but the same can be done for verbs as well: >>> cook = wordnet.synset('cook.v.01') >>> bake = wordnet.0('bake.v.02') >>> cook.wup_similarity(bake) 00.6666666666666666 The previous Synsets were obviously handpicked for demonstration, and the reason is that the hypernym tree for verbs has a lot more breadth and a lot less depth. Path and Leacock Chordorow (LCH) similarity Two other similarity comparisons are the path similarity and the LCH similarity, as shown in the following code: >>> cb.path_similarity(ib) 0.3333333333333333 >>> cb.path_similarity(dog) 0.07142857142857142 >>> cb.lch_similarity(ib) 2.538973871058276 >>> cb.lch_similarity(dog) 0.9985288301111273 24 www.it-ebooks.info

Discovering word collocations

These bigrams are found using association measurement functions in the nltk.metrics package, as follows: >>> from nltk.corpus import webtext >>> from nltk.collocations import BigramCollocationFinder >>> from nltk.metrics import BigramAssocMeasures >>> words = [w.lower() for w in webtext.words('grail.txt')] >>> bcf = BigramCollocationFinder.from_words(words) >>> bcf.nbest(BigramAssocMeasures.likelihood_ratio, 4) [("'"", 's'), ('arthur', ':'), ('#', '1'), ("'"", 't')] 25 www.it-ebooks.info Tokenizing Text and WordNet Basics Well, that's not very useful! Let's refine it a bit by adding a word filter to remove punctuation and stopwords: >>> from nltk.corpus import stopwords >>> stopset = set(stopwords.words('english')) >>> filter_stops = lambda w: len(w) < 3 or w in stopset >>> bcf.apply_word_filter(filter_stops) >>> bcf.nbest(BigramAssocMeasures.likelihood_ratio, 4) [('black', 'knight'), ('clop', 'clop'), ('head', 'knight'), ('mumble', 'mumble')] Much better, we can clearly see four of the most common bigrams in Monty Python and the Holy Grail. This time, we'll look for trigrams in Australian singles advertisements with the help of the following code: >>> from nltk.collocations import TrigramCollocationFinder >>> from nltk.metrics import TrigramAssocMeasures >>> words = [w.lower() for w in webtext.words('singles.txt')] >>> tcf = TrigramCollocationFinder.from_words(words) >>> tcf.apply_word_filter(filter_stops) >>> tcf.apply_freq_filter(3) >>> tcf.nbest(TrigramAssocMeasures.likelihood_ratio, 4) [('long', 'term', 'relationship')] Now, we don't know whether people are looking for a long-term relationship or not, but clearly it's an important topic. Scoring ngrams In addition to the nbest() method, there are two other ways to get ngrams (a generic term used for describing bigrams and trigrams) from a collocation finder: above_score(score_fn, min_score): This can be used to get all ngrams with f scores that are at least min_score. score_ngrams(score_fn): This will return a list with tuple pairs of (ngram, score).

Replacing and Correcting Words

No text here

Introduction

The recipes cover the gamut of linguistic compression, spelling correction, and text normalization. All of these methods can be very useful for preprocessing text before search indexing, document classification, and text analysis.

Stemming words

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 following is an inheritance diagram that explains this: StemmerI stem() LancasterStemmer PorterStemmer RegexpStemmer SnowballStemmer The LancasterStemmer class The functions of the LancasterStemmer class are just like the functions of the PorterStemmer class, but can produce slightly different results. 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

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. 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.

Replacing words matching regular expressions

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... Here is a simple usage example: >>> from replacers import RegexpReplacer >>> replacer = RegexpReplacer() >>> replacer.replace("can't is a contraction") 'cannot is a contraction' >>> replacer.replace("I should've done that thing I didn't do") 'I should have done that thing I did not do' The RegexpReplacer.replace() function works by replacing every instance of a replacement pattern with its corresponding substitution pattern. 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. Replacement before tokenization Let's try using the RegexpReplacer class as a preliminary step before tokenization: >>> from nltk.tokenize import word_tokenize >>> from replacers import RegexpReplacer >>> replacer = RegexpReplacer() >>> word_tokenize("can't is a contraction") ['ca', "n't", 'is', 'a', 'contraction'] >>> word_tokenize(replacer.replace("can't is a contraction")) ['can', 'not', 'is', 'a', 'contraction'] Much better!

Removing 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 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. So, the word looooove gets split into (looo)(o)o(ve) and then recombined as loooove, discarding the last o. To correct this issue, we can augment the replace() function with a WordNet lookup. 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

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. If not, we will look up the suggested alternatives and return the best match using nltk.metrics.edit_distance(): import enchant from nltk.metrics import edit_distance class SpellingReplacer(object): def __init__(self, dict_name='en', max_dist=2): self.spell_dict = enchant.Dict(dict_name) self.max_dist = max_dist def replace(self, word): if self.spell_dict.check(word): return word suggestions = self.spell_dict.suggest(word) if suggestions and edit_distance(word, suggestions[0]) <= self.max_dist: return suggestions[0] else: return word The preceding class can be used to correct English spellings, as follows: >>> from replacers import SpellingReplacer >>> replacer = SpellingReplacer() >>> replacer.replace('cookbok') 'cookbook' How it works... 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 try this yourself with the following code: >>> from nltk.metrics import edit_distance >>> edit_distance('language', 'languege') 1 >>> edit_distance('language', 'languo') 3 There's more... You can use language dictionaries other than en, such as en_GB, assuming the dictionary has already been installed. To check which other languages are available, use enchant.list_languages(): >>> enchant.list_languages() ['en', 'en_CA', 'en_GB', 'en_US'] If you try to use a dictionary that doesn't exist, you will get enchant.DictNotFoundError. 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.

Replacing synonyms

How to do it... 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... 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. 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. 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

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. Here are some examples: >>> from replacers import AntonymReplacer >>> replacer = AntonymReplacer() >>> replacer.replace('good') >>> replacer.replace('uglify') 'beautify' >>> sent = ["let's", 'not', 'uglify', 'our', 'code'] >>> replacer.replace_negations(sent) ["let's", 'beautify', 'our', 'code'] How it works... 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.

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. This information is essential for future chapters when we'll need to access the corpora as training data. You've already accessed the WordNet corpus in Chapter 1, Tokenizing Text and WordNet Basics.

Setting up a custom corpus

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): ... os.mkdir(path) >>> os.path.exists(path) True >>> import nltk.data >>> path in nltk.data.path True If the last line, path in nltk.data.path, is True, then you should now have a nltk_data directory in your home directory. 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. 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. 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 file is found using nltk.data.find(path), which searches all known paths combined with the relative path.

Creating a wordlist corpus

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 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). It contains two files: female.txt and male.txt, each containing a list of a few thousand common first names organized by gender as follows: >>> from nltk.corpus import names >>> names.fileids() ['female.txt', 'male.txt'] >>> len(names.words('female.txt')) 5001 >>> len(names.words('male.txt')) 2943 English words corpus NLTK also comes with a large list of English words. 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.

Creating a part-of-speech tagged word corpus

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'), ('. 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 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... All the functions we just demonstrated depend on tokenizers to split the text. 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', '.']] Then you pass in tagset='universal' to a method like tagged_words(), as shown in the following code: >>> reader = TaggedCorpusReader('. ', r'. *\.pos', tagset='en-brown') >>> reader.tagged_words(tagset='universal') [('The', 'DET'), ('expense', 'NOUN'), ('and', 'CONJ'), ...] Most NLTK tagged corpora are initialized with a known tagset, making conversion easy. The following is an example with the treebank corpus: >>> from nltk.corpus import treebank >>> treebank.tagged_words() [('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ...] >>> treebank.tagged_words(tagset='universal') [('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.

Creating a chunked phrase corpus

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'), ('. ', '.')])]] Sentence level trees look like Tree('S', [...]) while noun phrase trees look like Tree('NP', [...]). 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. 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. It has the same default sent_tokenizer and para_block_reader functions, but instead of a word_tokenizer function, it uses a str2chunktree() function. 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. If you want to customize chunk parsing, then you can pass in your own function for str2chunktree(). The third argument to ConllChunkCorpusReader should be a tuple or list specifying the types of chunks in the file, which in this case is ('NP', 'VP', 'PP'): >>> from nltk.corpus.reader import ConllChunkCorpusReader >>> conllreader = ConllChunkCorpusReader('. ', r'. *\.iob', ('NP', 'VP', 'PP')) >>> conllreader.chunked_words() [Tree('NP', [('Mr. ', 'NNP'), ('Meador', 'NNP')]), Tree('VP', [('had', 'VBD'), ('been', 'VBN')]), ...] >>> conllreader.chunked_sents() [Tree('S', [Tree('NP', [('Mr. ', 'NNP'), ('Meador', 'NNP')]), Tree('VP', [('had', 'VBD'), ('been', 'VBN')]), Tree('NP', [('executive', 'JJ'), ('vice', 'NN'), ('president', 'NN')]), Tree('PP', [('of', 'IN')]), Tree('NP', [('Balcor', 'NNP')]), ('. ', '.')])] >>> conllreader.iob_words() [('Mr. ', 'NNP', 'B-NP'), ('Meador', 'NNP', 'I-NP'), ...] >>> conllreader.iob_sents() [[('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'), ('. ', 'O')]] The previous code also shows the iob_words() and iob_sents() methods, which return lists of three tuples of (word, pos, iob). The inheritance diagram for ConllChunkCorpusReader looks like the following diagram, with most of the methods implemented by its superclass, ConllCorpusReader: CorpusReader fileids() ConllCorpusReader words() sents() tagged_words() tagged_sents() chunked_words() chunked_sents() ob_words() i ob_sents() i ConllChunkCorpusReader 62 www.it-ebooks.info Chapter 3 Tree leaves When it comes to chunk trees, the leaves of a tree are the tagged tokens. So if you want to get a list of all the tagged tokens in a tree, call the leaves() method using the following code: >>> reader.chunked_words()[0].leaves() [('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS')] >>> reader.chunked_sents()[0].leaves() [('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS'), ('have', 'VBP'), ('trimmed', 'VBN'), ('about', 'IN'), ('300', 'CD'), ('jobs', 'NNS'), (',', ','), ('the', 'DT'), ('spokesman', 'NN'), ('said', 'VBD'), ('. ', '.')] >>> reader.chunked_paras()[0][0].leaves() [('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS'), ('have', 'VBP'), ('trimmed', 'VBN'), ('about', 'IN'), ('300', 'CD'), ('jobs', 'NNS'), (',', ','), ('the', 'DT'), ('spokesman', 'NN'), ('said', 'VBD'), ('. ', '.')] In addition to Noun Phrases (NP), it also contains Verb Phrases (VP) and Prepositional Phrases (PP).

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. 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... This way, you could get all the pos sentences by calling reader.sents(categories=['pos']). 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

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. If no categories are given, _resolve() just returns the given fileids, which could be None, in which case all the files are read. 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 treebank corpus. All we're doing is making categories out of the fileids arguments, but the point is that you could use the same techniques to create your own categorized chunk corpus: >>> import nltk.data >>> from catchunked import CategorizedChunkedCorpusReader >>> path = nltk.data.find('corpora/treebank/tagged') >>> reader = CategorizedChunkedCorpusReader(path, r'wsj_. *)\.pos') >>> len(reader.categories()) == len(reader.fileids()) True >>> len(reader.chunked_sents(categories=['0001'])) 16 We use nltk.data.find() to search the data directories to get a FileSystemPathPointer class to the treebank corpus. from nltk.corpus.reader import CategorizedCorpusReader, ConllCorpusReader, ConllChunkCorpusReader class CategorizedConllChunkCorpusReader(CategorizedCorpusReader,   ConllChunkCorpusReader): def __init__(self, *args, **kwargs):   CategorizedCorpusReader.__init__(self, kwargs) ConllChunkCorpusReader.__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

Lazy corpus loading

And while you'll often want to specify a corpus reader in a common module, you don't always need to access it right away. 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. 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. The third argument to LazyCorpusLoader is the list of filenames and fileids that will be passed to WordListCorpusReader at initialization: >>> from nltk.corpus.util import LazyCorpusLoader >>> from nltk.corpus.reader import WordListCorpusReader >>> reader = LazyCorpusLoader('cookbook', WordListCorpusReader, ['wordlist']) >>> isinstance(reader, LazyCorpusLoader) True >>> reader.fileids() ['wordlist'] >>> isinstance(reader, LazyCorpusLoader) False >>> isinstance(reader, WordListCorpusReader) True How it works... The LazyCorpusLoader class stores all the arguments given, but otherwise does nothing until you try to access an attribute or method. So in the previous example code, before we call reader.fileids(), reader is an instance of LazyCorpusLoader, but after the call, reader becomes an instance of WordListCorpusReader.

Creating 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. 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 TaggedCorpusView class is a subclass of StreamBackedCorpusView that knows to split paragraphs of word/tag into (word, tag) tuples. 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... Reads one block with read_blankline_block(), which then reads the heading as a paragraph, and moves the stream's file position forward to the next block. 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. Subclass StreamBackedCorpusView and override the read_block() method. Unless otherwise mentioned, each block reader function takes a single argument: the stream argument to read from: read_whitespace_block(): This will read 20 lines from the stream, splitting f each line into tokens by whitespace. read_wordpunct_block(): This reads 20 lines from the stream, splitting each f line using nltk.tokenize.wordpunct_tokenize(). read_line_block(): This reads 20 lines from the stream and returns them as f a list, with each line as a token. read_regexp_block(): This takes two additional arguments, which must be f regular expressions that can be passed to re.match(): start_re and end_re.

Creating a MongoDB-backed corpus reader

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. 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 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. The sent_tokenizer is loaded on demand with LazyLoader, which is a wrapper around nltk.data.load(), analogous to LazyCorpusLoader. The LazyConcatentation class is a subclass of AbstractLazySequence too, and produces a flat list from a given list of lists (each list may also be lazy). 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

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. Instead of using with lockfile.FileLock(fname), you can also get a lock by calling lock = lockfile.FileLock(fname), then call lock.acquire() to acquire the lock, and lock.release() to release the lock. How it works... 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. The easiest way to do this while writing the changes back to the file, is to use a temporary file to hold the changes, then copy that file back into the original file using shutil.copyfileobj(). 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.

Part-of-speech Tagging

No text here

Introduction

Part-of-speech tagging is the process of converting a sentence, in the form of a list of words, into a list of tuples, where each tuple is of the form (word, tag). The tag is a part-of-speech tag, and signifies whether the word is a noun, adjective, verb, and so on.

Default tagging

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. This list of tokens is usually a list of words produced by a word tokenizer (see Chapter 1, Tokenizing Text and WordNet Basics, for more on tokenization). As you can see, tag() returns a list of tagged tokens, where a tagged token is a tuple of (word, tag). Every subclass of SequentialBackoffTagger must implement the choose_tag() method, which takes three arguments: The list of tokens f The index of the current token whose tag we want to choose f The history, which is a list of the previous tags f SequentialBackoffTagger implements the tag() method, which calls the choose_tag() method of the subclass for each index in the tokens list while accumulating a history of the previously tagged tokens. 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. 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. >>> 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. 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', '. [[('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. >>> from nltk.tag import untag >>> untag([('Hello', 'NN'), ('World', 'NN')]) ['Hello', 'World'] 88 www.it-ebooks.info

Training a unigram part-of-speech tagger

>>> from nltk.tag import UnigramTagger >>> from nltk.corpus import treebank >>> train_sents = treebank.tagged_sents()[:3000] >>> tagger = UnigramTagger(train_sents) >>> treebank.sents()[0] ['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutive', 'director', 'Nov. >>> tagger.tag(treebank.sents()[0]) [('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'), ('Nov. ', 'NNP'), ('29', 'CD'), ('. ', '.')] Then, we see the first sentence as a list of words, and can see how it is transformed by the tag() function into a list of tagged tokens. 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(). The result of context() is, in this case, the word token. 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 = 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. Then, you can put this UnigramTagger as your first backoff tagger (covered in the next recipe) to look up tags for unambiguous words. 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

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%! 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. 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. If your trained tagger is called tagger, then here's how to dump and load it with pickle: >>> import pickle >>> f = open('tagger.pickle', 'wb') >>> pickle.dump(tagger, f) >>> f.close() >>> f = open('tagger.pickle', 'rb') >>> tagger = pickle.load(f) If your tagger pickle file is located in an NLTK data directory, you could also use nltk.data.load('tagger.pickle') to load the tagger.

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. Internally, each tagger maintains a context dictionary (implemented in the ContextTagger parent class) that is used to guess that tag based on the context. >>> 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. 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. Here's some code to clarify this chain: >>> tagger._taggers[-1] == backoff True >>> isinstance(tagger._taggers[0], TrigramTagger) True >>> isinstance(tagger._taggers[1], BigramTagger) True So, we get a TrigramTagger, whose first backoff is a BigramTagger. BigramTagger and TrigramTagger, because they are subclasses of NgramTagger and ContextTagger, can also take a model and cutoff argument, just like the UnigramTagger. For the BigramTagger, an appropriate context key looks like ((prevtag,), word), and for TrigramTagger, it looks like ((prevtag1, prevtag2), word). >>> from nltk.tag import NgramTagger >>> quadtagger = NgramTagger(4, train_sents) >>> quadtagger.evaluate(test_sents) 0.058234405352903085 It's even worse than the TrigramTagger! 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, [UnigramTagger, BigramTagger, TrigramTagger, QuadgramTagger], backoff=backoff) >>> quadtagger.evaluate(test_sents) 0.8806388948845241 It's actually slightly worse than before, when we stopped with the TrigramTagger.

Creating a model of likely word tags

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. The word_tag_model() function takes a list of all words, a list of all tagged words, and   the maximum number of words we want to use for our model. 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. The FreqDist class is actually a subclass   of collections.Counter, which provides the most_common() method. In the previous edition of this book, we used the keys() method of the FreqDist class because in NLTK2, the keys were returned in sorted order, from the most frequent to the least. 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

The patterns shown in the following code can be found in tag_util.py: patterns = [(r'^\d+$', 'CD'), (r'. *ing$', 'VBG'), # gerunds, i.e. wonderful] Once you've constructed this list of patterns, you can pass it into RegexpTagger. >>> from tag_util import patterns >>> from nltk.tag import RegexpTagger >>> tagger = RegexpTagger(patterns) >>> tagger.evaluate(test_sents) 0.037470321605870924 So, it's not too great with just a few patterns, but since RegexpTagger is a subclass of SequentialBackoffTagger, it can be a useful part of a backoff chain.

Affix tagging

>>> from nltk.tag import AffixTagger >>> tagger = AffixTagger(train_sents) >>> tagger.evaluate(test_sents) 0.27558817181092166 So, it does ok by itself with the default arguments. >>> 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]. Here's an example of four AffixTagger classes learning on 2 and 3 character prefixes and suffixes: >>> pre3_tagger = AffixTagger(train_sents, affix_length=3) >>> pre3_tagger.evaluate(test_sents) 0.23587308439456076 >>> pre2_tagger = AffixTagger(train_sents, affix_length=2, backoff=pre3_tagger) >>> pre2_tagger.evaluate(test_sents) 0.29786315562270665 >>> suf2_tagger = AffixTagger(train_sents, affix_length=-2, backoff=pre2_tagger) >>> suf2_tagger.evaluate(test_sents) 0.32467083962875026 >>> suf3_tagger = AffixTagger(train_sents, affix_length=-3, backoff=suf2_tagger) >>> suf3_tagger.evaluate(test_sents) 0.3590761925318368 As you can see, the accuracy goes up each time.

Training a Brill tagger

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. >>> default_tagger = DefaultTagger('NN') >>> initial_tagger = backoff_tagger(train_sents, [UnigramTagger, BigramTagger, TrigramTagger], backoff=default_tagger) >>> initial_tagger.evaluate(test_sents) 0.8806820634578028 >>> from tag_util import train_brill_tagger >>> brill_tagger = train_brill_tagger(initial_tagger, train_sents) >>> brill_tagger.evaluate(test_sents) 0.8827541549751781 So, the BrillTagger class has slightly increased accuracy over the initial_tagger. 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. Word([-2, -1])) means you can look at the combination of the previous two words to learn a transformation 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. The workflow looks something like this: BrillTaggerTrainer train() BrillTagger <<trains>> <<uses>> <<uses>> BrillTemplatel BrillRule <<generates>> There's more... You can control the number of rules generated using the max_rules keyword argument to the BrillTaggerTrainer.train() method. 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... Found 9869 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. But if you do not call train() before evaluate(), you'll get an accuracy of 0%. 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! 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. >>> 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

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. 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. Here's some sample usage code: >>> from taggers import WordNetTagger >>> wn_tagger = WordNetTagger() >>> wn_tagger.evaluate(train_sents) 0.17914876598160262 So, it's not too accurate, but that's to be expected. >>> 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

Then, we implement the choose_tag() method, which simply checks whether the current word is in the names_set list. 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.

Classifier-based tagging

>>> 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. The ClassifierBasedTagger class also inherits from FeatursetTaggerI (which is just an empty class), creating an inheritance tree that looks like this: TaggerI tag() evaluate() SequentialBackoffTagger choose_tag() FeaturesetTaggerI ClassifierBasedTagger feature_detector() ClassifierBasedPOSTagger There's more... You can use a different classifier instead of NaiveBayesClassifier by passing in your own classifier_builder function. 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. Either way, you need a feature detection method that can take the same arguments as choose_tag(): tokens, index, history. 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%.