Introduction

www.it-ebooks.info . NLTK is often used for rapid prototyping of text processing programs and can even be used in production applications.

Tokenizing text into sentences

Instructions on how   to do so are available at http://nltk.org/install.html. Make sure that tokenizers/punkt.zip is in the data directory and has been unpacked so that there's a file at tokenizers/punkt/PY3/english.pickle.

Tokenizing sentences into words

These differ from TreebankWordTokenizer by how they handle punctuation and contractions, but they all inherit from TokenizerI. 11 www.it-ebooks.info .

Tokenizing sentences using regular expressions

As regular expressions can get complicated very quickly, I only recommend using them if   the word tokenizers covered in the previous recipe are unacceptable. 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.

Training a sentence tokenizer

As you can see, this isn't your standard paragraph of sentences formatting, which makes   it a perfect case for training a sentence tokenizer. 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

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 . 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.

Looking up Synsets for a word in WordNet

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')]] 19 www.it-ebooks.info . 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.

Looking up lemmas and synonyms in WordNet

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'. for lemma in syn.lemmas(): ...

Calculating WordNet Synset similarity

This tree can be used for reasoning about the similarity between the Synsets it contains. The closer the two Synsets are in the tree, the more similar they are.

Discovering word collocations

f This can be used to inform your choice for min_score. Discovering collocations in this list of words means that we'll find common phrases that occur frequently throughout the text.

Introduction

All of these methods can be very useful for preprocessing text before search indexing, document classification, and text analysis. www.it-ebooks.info .

Stemming words

For example, the stem of cooking is cook, and a good stemming algorithm knows that the ing suffix can be removed. The resulting stem is often a shorter word, or at least a common form of the word, which has the same root meaning.

Lemmatizing words with WordNet

So unlike stemming, you are always left with a valid word that means the same thing. However, the word you end up with can be completely different.

Replacing words matching regular expressions

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... 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.

Removing repeating characters

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... 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

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... 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

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. 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...

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.   If not is found, then we try to find an antonym for the next word using replace().

Introduction

This chapter will introduce you to many more corpora. This information is essential for future chapters when we'll need to access the corpora as training data.

Setting up a custom corpus

Create this corpora directory within the nltk_data directory, so that the path is ~/nltk_data/corpora. For simplicity, I'll refer to the directory as   ~/nltk_data.

Creating a wordlist corpus

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). 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

The following is an excerpt from the brown corpus: The/at-tl expense/nn and/cc time/nn involved/vbn are/ber   astronomical/jj ./. For example, treebank tag   mappings are in nltk_data/taggers/universal_tagset/en-ptb.map.

Creating a chunked phrase corpus

.   O ends the sentence.

Creating a categorized text corpus

The following are two excerpts from the movie_reviews corpus: movie_pos.txt: f the thin red line is flawed but it provokes . 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(). 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 both')   if categories is not None: return self.fileids(categories)   else: return fileids All the following methods call the corresponding method of ConllCorpusReader with the value returned from _resolve().

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.

Creating a custom corpus view

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...   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.

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... 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. Once the lock is acquired, any other process that tries to access the file will have to wait until the lock is released.

Introduction

www.it-ebooks.info . The tag is a part-of-speech tag, and signifies whether the word is a noun, adjective, verb, and so on.

Default tagging

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. DefaultTagger is most useful when you choose   the most common part-of-speech tag.

Training a unigram part-of-speech tagger

The tag   with the highest frequency for a context is stored in the model. In other words, UnigramTagger   is a context-based tagger whose context is a single word, or unigram.

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.   But if it returns None, then the next tagger is tried, and so on until a tag is found, or else   None is returned.

Training and combining ngram taggers

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. 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.

Creating a model of likely word tags

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. 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.

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. wonderment   (r'.*ful$', 'JJ') # i.e.

Affix tagging

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. If a word is less than five characters, then None   is returned as the tag.

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.   Found 9869 useful rules.

Training the TnT tagger

However, don't assume that accuracy will not change if you decrease N; experiment with   your own data to be sure.   You must explicitly call the train() method after you've created it.

Using WordNet for tagging

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. 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.

Tagging proper names

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... If it isn't, we return None,   so the next tagger in the chain can tag the word.

Classifier-based 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. 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.

Training a tagger with NLTK-Trainer

Found 1304 useful rules. As we did earlier, we can boost the accuracy a bit by using a default tagger: $ python train_tagger.py treebank --no-pickle --default NN --fraction 0.75 --sequential u loading treebank 3914 tagged sents, training on 2936 training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff <DefaultTagger: tag=NN> evaluating UnigramTagger accuracy: 0.873462 116 www.it-ebooks.info Chapter 4 Now, let's try adding a bigram tagger and trigram tagger: $ python train_tagger.py treebank --no-pickle --default NN --fraction 0.75 --sequential ubt loading treebank 3914 tagged sents, training on 2936 training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff <DefaultTagger: tag=NN> training <class 'nltk.tag.sequential.BigramTagger'> tagger with backoff <UnigramTagger: size=8709> training <class 'nltk.tag.sequential.TrigramTagger'> tagger with backoff <BigramTagger: size=1836> evaluating TrigramTagger accuracy: 0.879012 The PYTHONHASHSEED environment variable has been omitted for clarity.

Introduction

The idea is that meaningful phrases can be extracted from a sentence by looking for particular patterns   of part-of-speech tags. www.it-ebooks.info .

Chunking and chinking with   regular expressions

}<VB.*>{ ... {<DT><NN.*><.*>*<NN.*>} ...

Merging and splitting chunks with   regular expressions

A SplitRule class will split a chunk into two chunks based on the specified split pattern. A MergeRule class is specified by flipping the curly braces, and will join chunks where the end of the first chunk matches the left pattern and the beginning of the next chunk matches the right pattern.

Expanding and removing chunks with regular expressions

These rules are as follows: ExpandLeftRule: Add unchunked (chink) words to the left of a chunk f ExpandRightRule: Add unchunked (chink) words to the right of a chunk   f UnChunkRule: Unchunk any matching chunk   f 133 www.it-ebooks.info Extracting Chunks How to do it...   That's because the final UnChunkRule undid the chunk created by the previous rules.

Partial parsing with regular expressions

{<IN>}   # chunk preposition ... Of the chunks   the chunker argument was able to guess, precision tells you how many were correct   and recall tells you how well the chunker did at finding correct chunks compared to   how many total chunks there were.

Training a tagger-based chunker

Here's the code from chunkers.py: from nltk.chunk import ChunkParserI from nltk.chunk.util import tree2conlltags, conlltags2tree from nltk.tag import UnigramTagger, BigramTagger from tag_util import backoff_tagger 139 www.it-ebooks.info Extracting Chunks def conll_tag_chunks(chunk_sents): tagged_sents = [tree2conlltags(tree) for tree in chunk_sents]   return [[(t, c) for (w, t, c) in sent] for sent in tagged_sents] class TagChunker(ChunkParserI):   def __init__(self, train_chunks, tagger_classes=[UnigramTagger, BigramTagger]):   train_sents = conll_tag_chunks(train_chunks) self.tagger = backoff_tagger(train_sents, tagger_classes)   def parse(self, tagged_sent):   if not tagged_sent: return None   (words, tags) = zip(*tagged_sent)   chunks = self.tagger.tag(tags)   wtc = zip(words, chunks)   return conlltags2tree([(w,t,c) for (w,(t,c)) in wtc]) Once we have our trained TagChunker, we can then evaluate the ChunkScore class   the same way we did for the RegexpParser class in the previous recipes: >>> from chunkers import TagChunker >>> from nltk.corpus import treebank_chunk >>> train_chunks = treebank_chunk.chunked_sents()[:3000] >>> test_chunks = treebank_chunk.chunked_sents()[3000:] >>> chunker = TagChunker(train_chunks) >>> score = chunker.evaluate(test_chunks) >>> score.accuracy() 0.9732039335251428 >>> score.precision() 0.9166534370535006 >>> score.recall() 0.9465573770491803 Pretty darn accurate! Training a chunker is clearly a great alternative to manually specified grammars and regular expressions. The tags are then tagged by the tagger to get IOB tags, which are   then recombined with the words and part-of-speech tags to create 3-tuples we can   pass to conlltags2tree() to return a final Tree.

Classification-based chunking

As a subclass of ChunkerParserI, it implements the parse() method, which converts the ((w, t), c) tuples produced by the internal tagger into Trees using conlltags2tree(): class ClassifierChunker(ChunkParserI):   def __init__(self, train_sents, feature_detector=prev_next_pos_iob, **kwargs):   if not feature_detector: feature_detector = self.feature_detector   train_chunks = chunk_trees2train_chunks(train_sents)   self.tagger = ClassifierBasedTagger(train=train_chunks,   feature_detector=feature_detector, **kwargs)   def parse(self, tagged_sent):   if not tagged_sent: return None   chunks = self.tagger.tag(tagged_sent)   return conlltags2tree([(w,t,c) for ((w,t),c) in chunks]) Using the same train_chunks and test_chunks from the treebank_chunk corpus in the previous recipe, we can evaluate this code from chunkers.py: >>> from chunkers import ClassifierChunker >>> chunker = ClassifierChunker(train_chunks) >>> score = chunker.evaluate(test_chunks) >>> score.accuracy() 0.9721733155838022 144 www.it-ebooks.info Chapter 5 >>> score.precision() 0.9258838793383068 >>> score.recall() 0.9359016393442623 Compared to the TagChunker class, all the scores have gone up a bit. To give the classifier as much information as we can, this feature set contains the current, previous, and next word and   part-of-speech tag, along with the previous IOB tag: def prev_next_pos_iob(tokens, index, history):   word, pos = tokens[index]   if index == 0:   prevword, prevpos, previob = ('<START>',)*3   else: prevword, prevpos = tokens[index-1]   previob = history[index-1] 143 www.it-ebooks.info Extracting Chunks   if index == len(tokens) - 1:   nextword, nextpos = ('<END>',)*2   else: nextword, nextpos = tokens[index+1]   feats = {   'word': word,   'pos': pos,   'nextword': nextword,   'nextpos': nextpos,   'prevword': prevword,   'prevpos': prevpos,   'previob': previob   }   return feats Now, we can define the ClassifierChunker class, which uses an internal ClassifierBasedTagger with features extracted using prev_next_pos_iob() and training sentences from chunk_trees2train_chunks().

Extracting named entities

If we get the sub_leaves() , we can see that Pierre Vinken is correctly combined into a single named entity: >>> subleaves(ne_chunk(treebank_chunk.tagged_sents()[0], binary=True), 'NE') [[('Pierre', 'NNP'), ('Vinken', 'NNP')]] 148 www.it-ebooks.info . Now, all named entities   will be tagged with NE: >>> ne_chunk(treebank_chunk.tagged_sents()[0], binary=True) Tree('S', [Tree('NE', [('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'), ('.', '.')]) So, binary in this case means that an arbitrary chunk either is or is not a named entity.

Extracting proper noun chunks

This class can be   found in chunkers.py: from nltk.chunk import ChunkParserI from nltk.chunk.util import conlltags2tree from nltk.corpus import names 149 www.it-ebooks.info Extracting Chunks class PersonChunker(ChunkParserI):   def __init__(self):   self.name_set = set(names.words())   def parse(self, tagged_sent):   iobs = []   in_person = False   for word, tag in tagged_sent: if word in self.name_set and in_person: iobs.append((word, tag, 'I-PERSON'))   elif word in self.name_set: iobs.append((word, tag, 'B-PERSON'))   in_person = True   else: iobs.append((word, tag, 'O'))   in_person = False   return conlltags2tree(iobs) The PersonChunker class iterates over the tagged sentence, checking whether each word is in its names_set (constructed from the names corpus). is the name   of a month.

Extracting location chunks

The helper method   iob_locations() is where the IOB LOCATION tags are produced, and the parse() method converts these IOB tags into a Tree: from nltk.chunk import ChunkParserI from nltk.chunk.util import conlltags2tree from nltk.corpus import gazetteers class LocationChunker(ChunkParserI):   def __init__(self):   self.locations = set(gazetteers.words())   self.lookahead = 0   for loc in self.locations: nwords = loc.count(' ')   if nwords > self.lookahead: self.lookahead = nwords 151 www.it-ebooks.info Extracting Chunks   def iob_locations(self, tagged_sent):   i = 0   l = len(tagged_sent)   inside = False   while i < l: word, tag = tagged_sent[i]   j = i + 1   k = j + self.lookahead   nextwords, nexttags = [], []   loc = False   while j < k: if ' '.join([word] + nextwords) in self.locations: if inside: yield word, tag, 'I-LOCATION'   else: yield word, tag, 'B-LOCATION'   for nword, ntag in zip(nextwords, nexttags):   yield nword, ntag, 'I-LOCATION'   loc, inside = True, True   i = j   break   if j < l: nextword, nexttag = tagged_sent[j]   nextwords.append(nextword)   nexttags.append(nexttag)   j += 1   else: break   if not loc: inside = False   i += 1   yield word, tag, 'O'   def parse(self, tagged_sent):   iobs = self.iob_locations(tagged_sent)   return conlltags2tree(iobs) 152 www.it-ebooks.info Chapter 5 We can use the LocationChunker class to parse the following sentence into two   locations?San Francisco CA is cold compared to San Jose CA: >>> from chunkers import LocationChunker >>> t = loc.parse([('San', 'NNP'), ('Francisco', 'NNP'), ('CA', 'NNP'), ('is', 'BE'), ('cold', 'JJ'), ('compared', 'VBD'), ('to', 'TO'), ('San', 'NNP'), ('Jose', 'NNP'), ('CA', 'NNP')]) >>> sub_leaves(t, 'LOCATION') [[('San', 'NNP'), ('Francisco', 'NNP'), ('CA', 'NNP')], [('San', 'NNP'), ('Jose', 'NNP'), ('CA', 'NNP')]] And the result is that we get two LOCATION chunks, just as expected. The gazetteers corpus is   a WordListCorpusReader class that contains the following location words: Country names f   U.S.

Training a named entity chunker

Using the ieertree2conlltags() and ieer_chunked_sents() functions in chunkers.py, we can create named entity chunk trees from the ieer corpus to train   the ClassifierChunker class created in the Classification-based chunking recipe: import nltk.tag from nltk.chunk.util import conlltags2tree from nltk.corpus import ieer def ieertree2conlltags(tree, tag=nltk.tag.pos_tag):   words, ents = zip(*tree.pos())   iobs = []   prev = None   for ent in ents: if ent == tree.label():   iobs.append('O')   prev = None   elif prev == ent: iobs.append('I-%s' % ent)   else: iobs.append('B-%s' % ent)   prev = ent   words, tags = zip(*tag(words))   return zip(words, tags, iobs) def ieer_chunked_sents(tag=nltk.tag.pos_tag):   for doc in ieer.parsed_docs():   tagged = ieertree2conlltags(doc.text, tag)   yield conlltags2tree(tagged) 154 www.it-ebooks.info Chapter 5 We'll use 80 out of 94 sentences for training, and the rest for testing. Once we have all   the IOB tags, then we can get the part-of-speech tags of all the words and join the words,   part-of-speech tags, and IOB tags into 3-tuples using zip().

Training a chunker with NLTK-Trainer

Training classifier (47000 instances) training NaiveBayes classifier evaluating ClassifierChunker ChunkParse score: IOB Accuracy: 88.3%   Precision: 40.9%   Recall: 50.5%   F-Measure: 45.2% Training on a custom corpus If you have a custom corpus that you want to use for training a chunker, you can do that by passing in the path to the corpus and the classname of a corpus reader in the --reader argument. Here's an example of running train_chunker.py on treebank_chunk: $ python train_chunker.py treebank_chunk loading treebank_chunk 4009 chunks, training on 4009 training ub TagChunker evaluating TagChunker ChunkParse score: IOB Accuracy: 97.0%   Precision: 90.8%   Recall: 93.9%   F-Measure: 92.3% dumping TagChunker to /Users/jacob/nltk_data/chunkers/treebank_chunk_ ub.pickle 156 www.it-ebooks.info Chapter 5 Just like with train_tagger.py, we can use the --no-pickle argument to skip saving a pickled chunker, and the --fraction argument to limit the training set and evaluate the chunker against a test set: $ python train_chunker.py treebank_chunk --no-pickle --fraction 0.75 loading treebank_chunk 4009 chunks, training on 3007 training ub TagChunker evaluating TagChunker ChunkParse score: IOB Accuracy: 97.3%   Precision: 91.6%   Recall: 94.6%   F-Measure: 93.1% The score output you see is what you get when you print a ChunkScore object.

Introduction

www.it-ebooks.info . The tree transforms give you ways to modify and flatten deep parse trees.

Filtering insignificant words from a sentence

It defaults to filtering out any tags that end with DT or CC: def filter_insignificant(chunk, tag_suffixes=['DT', 'CC']):   good = [] 164 www.it-ebooks.info Chapter 6   for word, tag in chunk: ok = True   for suffix in tag_suffixes: if tag.endswith(suffix):   ok = False   break   if ok: good.append((word, tag))   return good And now we can use it on the part-of-speech tagged version of the terrible movie: >>> from transforms import filter_insignificant >>> filter_insignificant([('the', 'DT'), ('terrible', 'JJ'),   ('movie', 'NN')]) [('terrible', 'JJ'), ('movie', 'NN')] As you can see, the word the is eliminated from the chunk.   Looking through the treebank corpus for stopwords yields the following table of   insignificant words and tags: Word Tag a DT all PDT an DT and CC or CC that WDT the DT Other than CC, all the tags end with DT.

Correcting verb forms

def correct_verbs(chunk):   vbidx = first_chunk_index(chunk, tag_startswith('VB'))   # if no verb found, do nothing   if vbidx is None: return chunk   verb, vbtag = chunk[vbidx]   nnpred = tag_startswith('NN')   # find nearest noun to the right of verb   nnidx = first_chunk_index(chunk, nnpred, start=vbidx+1)   # if no noun found to right, look to the left   if nnidx is None: nnidx = first_chunk_index(chunk, nnpred, start=vbidx-1, step=-1)   # if no noun found, do nothing   if nnidx is None: return chunk 167 www.it-ebooks.info Transforming Chunks and Trees   noun, nntag = chunk[nnidx]   # get correct verb form and insert into chunk   if nntag.endswith('S'):   chunk[vbidx] = plural_verb_forms.get((verb, vbtag), (verb, vbtag))   else: chunk[vbidx] = singular_verb_forms.get((verb, vbtag), (verb, vbtag))   return chunk When we call the preceding function on a part-of-speech tagged is our children learning chunk, we get back the correct form, are our children learning. def tag_startswith(prefix):   def f(wt):   return wt[1].startswith(prefix)   return f The tag_startswith() function takes a tag prefix, such as NN, and returns a predicate function that will take a (word, tag) tuple and return True if the tag starts with the given prefix.

Swapping verb phrases

It uses the first_chunk_index() function defined in the previous recipe to find the   verb to pivot around. def swap_verb_phrase(chunk):   def vbpred(wt):   word, tag = wt   return tag != 'VBG' and tag.startswith('VB') and len(tag) > 2   vbidx = first_chunk_index(chunk, vbpred)   if vbidx is None: return chunk   return chunk[vbidx+1:] + chunk[:vbidx] Now we can see how it works on the part-of-speech tagged phrase the book was great: >>> swap_verb_phrase([('the', 'DT'), ('book', 'NN'), ('was', 'VBD'), ('great', 'JJ')]) [('great', 'JJ'), ('the', 'DT'), ('book', 'NN')] And the result is great the book.

Swapping noun cardinals

It swaps any   cardinal that occurs immediately after a noun with the noun so that the cardinal occurs immediately before the noun. It uses a helper function, tag_equals(), which is similar to tag_startswith(), but in this case, the function it returns does an equality comparison with the given tag: def tag_equals(tag):   def f(wt):   return wt[1] == tag   return f Now we can define swap_noun_cardinal(): def swap_noun_cardinal(chunk):   cdidx = first_chunk_index(chunk, tag_equals('CD'))   # cdidx must be > 0 and there must be a noun immediately before it   if not cdidx or not chunk[cdidx-1][1].startswith('NN'):   return chunk   noun, nntag = chunk[cdidx-1]   chunk[cdidx-1] = chunk[cdidx]   chunk[cdidx] = noun, nntag   return chunk Let's try it on a date, such as Dec 10, and another common phrase, the top 10.

Swapping infinitive phrases

The   swap_infinitive_phrase() function, defined in transforms.py, will return   a chunk that swaps the portion of the phrase after the IN word with the portion   before the IN word: def swap_infinitive_phrase(chunk):   def inpred(wt):   word, tag = wt   return tag == 'IN' and word != 'like'   inidx = first_chunk_index(chunk, inpred)   if inidx is None: return chunk   nnidx = first_chunk_index(chunk, tag_startswith('NN'), start=inidx, step=-1) or 0   return chunk[:nnidx] + chunk[inidx+1:] + chunk[nnidx:inidx] The function can now be used to transform book of recipes into recipes book: >>> from transforms import swap_infinitive_phrase >>> swap_infinitive_phrase([('book', 'NN'), ('of', 'IN'), ('recipes', 'NNS')]) [('recipes', 'NNS'), ('book', 'NN')] How it works... Instead, we want to insert recipes before the noun book but after the adjective delicious, hence the need to   find the nnidx occurring before the inidx.

Singularizing plural nouns

The transforms.py script defines a function called singularize_plural_noun()   which will depluralize a plural noun (tagged with NNS) that is followed by another noun: def singularize_plural_noun(chunk): nnsidx = first_chunk_index(chunk, tag_equals('NNS'))   if nnsidx is not None and nnsidx+1 < len(chunk) and chunk[nnsidx+1] [1][:2] == 'NN':   noun, nnstag = chunk[nnsidx]   chunk[nnsidx] = (noun.rstrip('s'), nnstag.rstrip('S'))   return chunk And using it on recipes book, we get the more correct form, recipe book. We can do another transform to correct these improper plural nouns.

Chaining chunk transformations

It calls each transform function on the chunk, one at a time, and returns the final chunk: def transform_chunk(chunk, chain=[filter_insignificant, swap_verb_ phrase, swap_infinitive_phrase, singularize_plural_noun], trace=0):   for f in chain: chunk = f(chunk)   if trace: print f.__name__, ':', chunk   return chunk Using it on the phrase the book of recipes is delicious, we get delicious recipe book: >>> from transforms import transform_chunk >>> transform_chunk([('the', 'DT'), ('book', 'NN'), ('of', 'IN'), ('recipes', 'NNS'), ('is', 'VBZ'), ('delicious', 'JJ')]) [('delicious', 'JJ'), ('recipe', 'NN'), ('book', 'NN')] 174 www.it-ebooks.info Chapter 6 How it works... The transform_chunk() function defaults to chaining the following functions in the   given order: filter_insignificant()   f swap_verb_phrase()   f swap_infinitive_phrase()   f singularize_plural_noun()   f Each function transforms the chunk that results from the previous function, starting with   the original chunk.

Converting a chunk tree to text

This is implemented in the chunk_tree_to_sent() function found in transforms.py: import re punct_re = re.compile(r'\s([,\.;\?])') def chunk_tree_to_sent(tree, concat=' '):   s = concat.join([w for w, t in tree.leaves()])   return re.sub(punct_re, r'\g<1>', s) Using chunk_tree_to_sent() results in a cleaner sentence, with no space before each punctuation mark: >>> from transforms import chunk_tree_to_sent >>> chunk_tree_to_sent(tree) 'Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. To correct the extra spaces in front of the punctuation, we create a regular expression,   punct_re, that will match a space followed by any of the known punctuation characters.

Flattening a deep tree

It uses a helper function, flatten_childtrees(), to do most of the work: from nltk.tree import Tree def flatten_childtrees(trees):   children = []   for t in trees: if t.height() < 3:   children.extend(t.pos())   elif t.height() == 3:   children.append(Tree(t.label(), t.pos()))   else: children.extend(flatten_childtrees([c for c in t]))   return children def flatten_deeptree(tree):   return Tree(tree.label(), flatten_childtrees([c for c in tree])) We can use it on the first parsed sentence of the treebank corpus to get a flatter tree: >>> from nltk.corpus import treebank >>> from transforms import flatten_deeptree >>> flatten_deeptree(treebank.parsed_sents()[0]) Tree('S', [Tree('NP', [('Pierre', 'NNP'), ('Vinken', 'NNP')]), (',', ','), Tree('NP', [('61', 'CD'), ('years', 'NNS')]), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), Tree('NP', [('the', 'DT'), ('board', 'NN')]), ('as', 'IN'), Tree('NP', [('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN')]), Tree('NP-TMP', [('Nov.', 'NNP'), ('29', 'CD')]), ('.', '.')]) 178 www.it-ebooks.info Chapter 6 The result is a much flatter Tree that only includes NP phrases.   The main difference is that the rightmost NP Tree is separated into two subtrees above,   one of them named NP-TMP.

Creating a shallow tree

29 a nonexecutive director The shallow_tree() function defined in transforms.py eliminates all the nested subtrees, keeping only the top subtree labels: from nltk.tree import Tree def shallow_tree(tree):   children = []   for t in tree: if t.height() < 3:   children.extend(t.pos())   else: children.append(Tree(t.label(), t.pos()))   return Tree(tree.label(), children) Using it on the first parsed sentence in treebank results in a Tree with only two subtrees: >>> from transforms import shallow_tree >>> shallow_tree(treebank.parsed_sents()[0]) Tree('S', [Tree('NP-SBJ', [('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'), ('old', 'JJ'), (',', ',')]), Tree('VP', [('will', 'MD'), ('join', 'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'), ('Nov.', 'NNP'), ('29', 'CD')]), ('.', '.')]) 182 www.it-ebooks.info .   In this recipe, we'll keep only the highest level subtrees instead.

Converting tree labels

It returns a new Tree with all matching labels replaced based on the values in the mapping: from nltk.tree import Tree def convert_tree_labels(tree, mapping):   children = []   for t in tree: if isinstance(t, Tree):   children.append(convert_tree_labels(t, mapping))   else: children.append(t)   label = mapping.get(tree.label(), tree.label())   return Tree(label, children) 184 www.it-ebooks.info Chapter 6 Using the mapping table we saw earlier, we can pass it in as a dict to convert_tree_labels() and convert the first parsed sentence from treebank: >>> from transforms import convert_tree_labels >>> mapping = {'NP-SBJ': 'NP', 'NP-TMP': 'NP'} >>> convert_tree_labels(treebank.parsed_sents()[0], mapping) Tree('S', [Tree('NP', [Tree('NP', [Tree('NNP', ['Pierre']), Tree('NNP', ['Vinken'])]), Tree(',', [',']), Tree('ADJP', [Tree('NP', [Tree('CD', ['61']), Tree('NNS', ['years'])]), Tree('JJ', ['old'])]), Tree(',', [','])]), Tree('VP', [Tree('MD', ['will']), Tree('VP', [Tree('VB', ['join']), Tree('NP', [Tree('DT', ['the']), Tree('NN', ['board'])]), Tree('PP-CLR', [Tree('IN', ['as']), Tree('NP', [Tree('DT', ['a']), Tree('JJ', ['nonexecutive']), Tree('NN', ['director'])])]), Tree('NP', [Tree('NNP', ['Nov.']), Tree('CD', ['29'])])])]), Tree('.', ['.'])]) As you can see in the following diagram, the NP-* subtrees have been replaced with   NP subtrees: S NP VP . Let's convert both of those to NP.

Introduction

The text can either be one label or another, but not both, whereas a multi-label classifier can assign one or more labels to a piece of text. www.it-ebooks.info .

Bag of words feature extraction

The bag_of_bigrams_words() function found in featx.py will return a dict of all words along with the 200 most significant bigrams: from nltk.collocations import BigramCollocationFinder from nltk.metrics import BigramAssocMeasures def bag_of_bigrams_words(words, score_fn=BigramAssocMeasures.chi_sq, n=200):   bigram_finder = BigramCollocationFinder.from_words(words)   bigrams = bigram_finder.nbest(score_fn, n)   return bag_of_words(words + bigrams) The bigrams will be present in the returned dict as (word1, word2) and will have   the value as True. The idea is to convert a list of words into a dict, where each word becomes a key   with the value True.

Training a Naive Bayes classifier

Getting a fair sample should eliminate this possible bias: import collections def label_feats_from_corpus(corp, feature_detector=bag_of_words):   label_feats = collections.defaultdict(list)   for label in corp.categories():   for fileid in corp.fileids(categories=[label]):   feats = feature_detector(corp.words(fileids=[fileid]))   label_feats[label].append(feats)   return label_feats Once we can get a mapping of label **|** feature sets, we want to construct a list of labeled training instances and testing instances. In our case,   the feature value will always be True: >>> nb_classifier.most_informative_features(n=5) [('magnificent', True), ('outstanding', True), ('insulting', True), ('vulnerable', True), ('ludicrous', True)] 195 www.it-ebooks.info Text Classification The show_most_informative_features() method will print out the results from   most_informative_features() and will also include the probability of a feature   pair belonging to each label: >>> nb_classifier.show_most_informative_features(n=5) Most Informative Features   magnificent = True pos : neg = 15.0 : 1.0   outstanding = True   pos : neg = 13.6 : 1.0   insulting = True   neg : pos = 13.0 : 1.0   vulnerable = True   pos : neg = 12.3 : 1.0   ludicrous = True   neg : pos = 11.8 : 1.0 The informativeness, or information gain, of each feature pair is based on the prior   probability of the feature pair occurring for each label.

Training a decision tree classifier

We pass binary=True because all of our features are binary: either the word is   present or it's not. The default   value is 100, which means that classification may require up to 100 decisions before reaching a leaf node.

Training a maximum entropy classifier

The only difference between these two algorithms that really matters is that gis is much faster than iis.   In other words, if the word worst is found in the feature set, then there's a strong possibility that the text should be classified neg.

Training scikit-learn classifiers

Here's the complete class code, minus all comments, docstrings, and most imports: from sklearn.feature_extraction import DictVectorizer from sklearn.preprocessing import LabelEncoder class SklearnClassifier(ClassifierI):   def __init__(self, estimator, dtype=float, sparse=True):   self._clf = estimator   self._encoder = LabelEncoder()   self._vectorizer = DictVectorizer(dtype=dtype, sparse=sparse) 206 www.it-ebooks.info Chapter 7   def batch_classify(self, featuresets):   X = self._vectorizer.transform(featuresets)   classes = self._encoder.classes_   return [classes[i] for i in self._clf.predict(X)]   def batch_prob_classify(self, featuresets):   X = self._vectorizer.transform(featuresets)   y_proba_list = self._clf.predict_proba(X)   return [self._make_probdist(y_proba) for y_proba in y_proba_ list]   def labels(self):   return list(self._encoder.classes_)   def train(self, labeled_featuresets):   X, y = list(compat.izip(*labeled_featuresets))   X = self._vectorizer.fit_transform(X)   y = self._encoder.fit_transform(y)   self._clf.fit(X, y)   return self   def _make_probdist(self, y_proba):   classes = self._encoder.classes_   return DictionaryProbDist(dict((classes[i], p) for i, p in enumerate(y_proba))) The class is initialized with an estimator, which is the algorithm we pass in, such as MultinomialNB. >>> from sklearn.linear_model import LogisticRegression >>> sk_classifier = SklearnClassifier(LogisticRegression()) <SklearnClassifier(LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,   intercept_scaling=1, penalty='l2', random_state=None, tol=0.0001))> >>> sk_classifier.train(train_feats) >>> accuracy(sk_classifier, test_feats) 0.892 208 www.it-ebooks.info Chapter 7 Again, we see that the sklearn algorithm has better performance than NLTK's MaxentClassifier, which only had 72.2% accuracy.

Measuring precision and recall of   a classifier

The precision_recall() function in classification.py looks like this: import collections from nltk import metrics 210 www.it-ebooks.info Chapter 7 def precision_recall(classifier, testfeats):   refsets = collections.defaultdict(set)   testsets = collections.defaultdict(set)   for i, (feats, label) in enumerate(testfeats):   refsets[label].add(i)   observed = classifier.classify(feats)   testsets[observed].add(i)   precisions = {}   recalls = {}   for label in classifier.labels():   precisions[label] = metrics.precision(refsets[label], testsets[label])   recalls[label] = metrics.recall(refsets[label], testsets[label])   return precisions, recalls This function takes two arguments: The trained classifier   f Labeled test features, also known as a gold standard   f These are the same arguments you pass to accuracy().   Two of the most common are precision and recall.

Calculating high information words

We can do this using the high_information_words() function in featx.py: from nltk.metrics import BigramAssocMeasures from nltk.probability import FreqDist, ConditionalFreqDist def high_information_words(labelled_words, score_ fn=BigramAssocMeasures.chi_sq, min_score=5): word_fd = FreqDist()   label_word_fd = ConditionalFreqDist()   for label, words in labelled_words: for word in words: word_fd[word] += 1   label_word_fd[label][word] += 1   n_xx = label_word_fd.N()   high_info_words = set()   for label in label_word_fd.conditions():   n_xi = label_word_fd[label].N()   word_scores = collections.defaultdict(int) 214 www.it-ebooks.info Chapter 7   for word, n_ii in label_word_fd[label].items():   n_ix = word_fd[word]   score = score_fn(n_ii, (n_ix, n_xi), n_xx)   word_scores[word] = score   bestwords = [word for word, score in word_scores.items() if score >= min_score]   high_info_words |= set(bestwords) return high_info_words It takes one argument from a list of two tuples of the form [(label, words)] where label is the classification label, and words is a list of words that occur under that label. Once we have the high information words, we use the feature detector function   bag_of_words_in_set(), also found in featx.py, which will let us filter out   all low information words.

Combining classifiers with voting

In the classification.py module, there is a MaxVoteClassifier class: import itertools from nltk.classify import ClassifierI from nltk.probability import FreqDist class MaxVoteClassifier(ClassifierI):   def __init__(self, *classifiers):   self._classifiers = classifiers   self._labels = sorted(set(itertools.chain(*[c.labels() for c   in classifiers])))   def labels(self):   return self._labels   def classify(self, feats):   counts = FreqDist()   for classifier in self._classifiers: counts[classifier.classify(feats)] += 1   return counts.max() 219 www.it-ebooks.info Text Classification To create it, you pass in a list of classifiers that you want to combine. The   f MaxVoteClassifier class iterates over its classifiers and calls classify()   on each of them, recording their label as a vote in a FreqDist variable.

Classifying with multiple binary classifiers

The train_binary_classifiers() function in classification.py takes a training function, a list of multi-label feature sets, and a set of possible labels to return a dict of label : binary classifier: def train_binary_classifiers(trainf, labelled_feats, labelset):   pos_feats = collections.defaultdict(list)   neg_feats = collections.defaultdict(list)   classifiers = {}   for feat, labels in labelled_feats: for label in labels: pos_feats[label].append(feat)   for label in labelset - set(labels):   neg_feats[label].append(feat)   for label in labelset: postrain = [(feat, label) for feat in pos_feats[label]]   negtrain = [(feat, '!%s' % label) for feat in neg_feats[label]]   classifiers[label] = trainf(postrain + negtrain)   return classifiers To use this function, we need to provide a training function that takes a single argument, which is the training data. from nltk.classify import MultiClassifierI class MultiBinaryClassifier(MultiClassifierI):   def __init__(self, *label_classifiers):   self._label_classifiers = dict(label_classifiers)   self._labels = sorted(self._label_classifiers.keys())   def labels(self):   return self._labels   def classify(self, feats):   lbls = set()   for label, classifier in self._label_classifiers.items():   if classifier.classify(feats) == label: lbls.add(label)   return lbls Now we can construct this class using the binary classifiers we just created: >>> from classification import MultiBinaryClassifier >>> multi_classifier = MultiBinaryClassifier(*classifiers.items()) To evaluate this classifier, we can use precision and recall, but not accuracy.

Training a classifier with NLTK-Trainer

This can also be done as an argument in train_classifier.py: $ python train_classifier.py movie_reviews --no-pickle --fraction 0.75 --show-most-informative 5 loading movie_reviews 2 labels: ['neg', 'pos'] using bag of words feature extraction 1500 training feats, 500 testing feats training NaiveBayes classifier accuracy: 0.726000 neg precision: 0.952000 neg recall: 0.476000 neg f-measure: 0.634667 pos precision: 0.650667 pos recall: 0.976000 pos f-measure: 0.780800 5 most informative features Most Informative Features   finest = True   pos : neg   =   13.4 : 1.0   astounding = True   pos : neg   =   11.0 : 1.0   avoids = True   pos : neg   =   11.0 : 1.0   inject = True   neg : pos   =   10.3 : 1.0   strongest = True   pos : neg   =   10.3 : 1.0 231 www.it-ebooks.info Text Classification The Maxent and LogisticRegression classifiers In the Training a maximum entropy classifier recipe, we covered the MaxentClassifier class with the GIS algorithm. Here's how to use train_classifier.py to do this: $ python train_classifier.py movie_reviews --no-pickle --fraction 0.75 --classifier GIS --max_iter 10 --min_lldelta 0.5 loading movie_reviews 2 labels: ['neg', 'pos'] using bag of words feature extraction 1500 training feats, 500 testing feats training GIS classifier   ==> Training (10 iterations) accuracy: 0.712000 neg precision: 0.964912 neg recall: 0.440000 neg f-measure: 0.604396 pos precision: 0.637306 pos recall: 0.984000 pos f-measure: 0.773585 If you have scikit-learn installed, then you can use many different sklearn algorithms for classification.

Introduction

www.it-ebooks.info . Or, you might want to store frequencies and probabilities in a persistent, shared database so multiple processes can access it simultaneously.

Distributed tagging with execnet

The gateway's remote_exec() method takes a single argument that can be one of   the following three types: A string of code to execute remotely   f The name of a pure function that will be serialized and executed remotely   f The name of a pure module whose source will be executed remotely   f 239 www.it-ebooks.info Distributed Processing and Handling Large Datasets We use option three with the remote_tag.py module, which is defined as follows: import pickle   if __name__ == '__channelexec__':   tagger = pickle.loads(channel.receive())   for sentence in channel: channel.send(tagger.tag(sentence)) A pure module is a module that is self-contained: it can only access Python modules   that are available where it executes, and does not have access to any variables or   states that exist wherever the gateway is initially created.   channel = next(channels) ...

Distributed chunking with execnet

This Tree is then pickled   and sent back over the channel: import pickle if __name__ == '__channelexec__':   tagger = pickle.loads(channel.receive())   chunker = pickle.loads(channel.receive()) 243 www.it-ebooks.info . This will be very similar to the tagging in the previous recipe, but we'll be sending two objects instead of   one, and we will be receiving a Tree instead of a list, which requires pickling and unpickling for serialization.

Parallel list processing with execnet

Once we have all the expected results, we can exit the gateways and return the results: import itertools, execnet def map(mod, args, specs=[('popen', 2)]):   gateways = []   channels = []   for spec, count in specs: for i in range(count):   gw = execnet.makegateway(spec) gateways.append(gw)   channels.append(gw.remote_exec(mod))   cyc = itertools.cycle(channels)   for i, arg in enumerate(args):   channel = next(cyc)   channel.send((i, arg))   mch = execnet.MultiChannel(channels)   queue = mch.make_receive_queue()   l = len(args)   results = [None] * l # creates a list of length l, where every element is None   for i in range(l):   channel, (i, result) = queue.get()   results[i] = result   for gw in gateways: gw.exit()   return results 246 www.it-ebooks.info . if __name__ == '__channelexec__':   for (i, arg) in channel: channel.send((i, arg * 2)) To use this module to double every element in a list, we import the plists module   (explained in the How it works...

Storing a frequency distribution in Redis

This RedisHashFreqDist (defined in redisprob.py) sums all the values in the hash map for the N() method: from rediscollections import RedisHashMap class RedisHashFreqDist(RedisHashMap):   def N(self):   return int(sum(self.values()))   def __missing__(self, key):   return 0   def __getitem__(self, key):   return int(RedisHashMap.__getitem__(self, key) or 0)   def values(self):   return [int(v) for v in RedisHashMap.values(self)]   def items(self):   return [(k, int(v)) for (k, v) in RedisHashMap.items(self)] We can use this class just like a FreqDist. import collections, re white = re.compile('[\s&]+') def encode_key(key):   return white.sub('_', key.strip()) class RedisHashMap(collections.MutableMapping):   def __init__(self, r, name):   self._r = r   self._name = encode_key(name)   def __iter__(self):   return self.items()   def __len__(self):   return self._r.hlen(self._name)   def __contains__(self, key):   return self._r.hexists(self._name, encode_key(key))   def __getitem__(self, key):   return self._r.hget(self._name, encode_key(key))   def __setitem__(self, key, val):   self._r.hset(self._name, encode_key(key), val)   def __delitem__(self, key):   self._r.hdel(self._name, encode_key(key))   def keys(self):   return self._r.hkeys(self._name)   def values(self):   return self._r.hvals(self._name)   def items(self):   return self._r.hgetall(self._name).items()   def get(self, key, default=0):   return self[key] or default   def clear(self):   self._r.delete(self._name) 250 www.it-ebooks.info .

Storing a conditional frequency distribution in Redis

We override __getitem__() so we can create an instance of RedisHashFreqDist instead of a FreqDist: from nltk.probability import ConditionalFreqDist from rediscollections import encode_key 251 www.it-ebooks.info Distributed Processing and Handling Large Datasets class RedisConditionalHashFreqDist(ConditionalFreqDist):   def __init__(self, r, name, cond_samples=None):   self._r = r   self._name = name   ConditionalFreqDist.__init__(self, cond_samples)   for key in self._r.keys(encode_key('%s:*' % name)):   condition = key.split(':')[1]   self[condition] # calls self.__getitem__(condition)   def __getitem__(self, condition):   if condition not in self._fdists: key = '%s:%s' % (self._name, condition)   val = RedisHashFreqDist(self._r, key)   super(RedisConditionalHashFreqDist, self).__setitem__(condition, val)   return super(RedisConditionalHashFreqDist, self).__getitem__ (condition)   def clear(self):   for fdist in self.values():   fdist.clear() An instance of this class can be created by passing in a Redis connection and a base name. Here, we'll create an API-compatible class on top of Redis using the RedisHashFreqDist from the previous recipe.

Storing an ordered dictionary in Redis

Then, it implements all the key methods that require Redis   ordered set (also known as Zset) commands: class RedisOrderedDict(collections.MutableMapping):   def __init__(self, r, name):   self._r = r   self._name = encode_key(name)   def __iter__(self):   return iter(self.items())   def __len__(self):   return self._r.zcard(self._name)   def __getitem__(self, key):   return self._r.zscore(self._name, encode_key(key))   def __setitem__(self, key, score):   self._r.zadd(self._name, encode_key(key), score)   def __delitem__(self, key):   self._r.zrem(self._name, encode_key(key))   def keys(self, start=0, end=-1):   # we use zrevrange to get keys sorted by high value instead of by lowest   return self._r.zrevrange(self._name, start, end)   def values(self, start=0, end=-1):   return [v for (k, v) in self.items(start=start, end=end)]   def items(self, start=0, end=-1):   return self._r.zrevrange(self._name, start, end, withscores=True)   def get(self, key, default=0):   return self[key] or default   def iteritems(self):   return iter(self)   def clear(self):   self._r.delete(self._name) 254 www.it-ebooks.info Chapter 8 You can create an instance of RedisOrderedDict by passing in a Redis connection and a unique name: >>> from redis import Redis >>> from rediscollections import RedisOrderedDict >>> r = Redis() >>> rod = RedisOrderedDict(r, 'test') >>> rod.get('bar') >>> len(rod) 0 >>> rod['bar'] = 5.2 >>> rod['bar'] 5.2000000000000002 >>> len(rod) 1 >>> rod.items() [(b'bar', 5.2)] >>> rod.clear() By default, keys are returned as binary strings.   f items(): This uses the zrevrange command to get the scores of each key in order   f to return a list of 2-tuples ordered by the highest score.