www.it-ebooks.info

No text here

www.packtpub.com

No text here

Credits

No text here

About the Author

He is the author of Python Text Processing with NLTK 2.0 Cookbook, Packt Publishing, and has contributed a chapter to the Bad Data Handbook, O'Reilly Media. To demonstrate the capabilities of NLTK and natural language processing, he developed http://text-processing.com, which provides simple demos and NLP APIs for commercial use. Finally, this book wouldn't be possible without the fantastic NLTK project and team: http://www.nltk.org/.

About the Reviewers

Earlier, he graduated from the University of Southern California (USC) with a Master's degree in Computer Science. www.it-ebooks.info Lihang Li received his BE degree in Mechanical Engineering from Huazhong University of Science and Technology (HUST), China, in 2012, and now is pursuing his MS degree in Computer Vision at National Laboratory of Pattern Recognition (NLPR) from the Institute of Automation, Chinese Academy of Sciences (IACAS).

www.PacktPub.com

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks. Fully searchable across every book published by Packt f Copy and paste, print and bookmark content f On demand and accessible via web browser f Free Access for Packt account holders If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books.

Table of Contents

No text here

Preface

The following is the list of the packages in requirements format with the version number used while writing this book: NLTK>=3.0a4 f pyenchant>=1.6.5 f lockfile>=0.9.1 f numpy>=1.8.0 f scipy>=0.13.0 f scikit-learn>=0.14.1 f execnet>=1.1 f pymongo>=2.6.3 f redis>=2.8.0 f 2 www.it-ebooks.info Preface lxml>=3.2.3 f beautifulsoup4>=4.3.2 f python-dateutil>=2.0 f charade>=1.0.3 f You will also need NLTK-Trainer, which is available at the following link: https://github.com/japerk/nltk-trainer Beyond Python, there are a couple recipes that use MongoDB and Redis, both NoSQL databases. When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold: >>> doc.make_links_absolute('http://hello') >>> abslinks = list(doc.iterlinks()) >>> (el, attr, link, pos) = abslinks[0] >>> link 'http://hello/world' 3 www.it-ebooks.info Preface Any command-line input or output is written as follows: $ python train_chunker.py treebank_chunk New terms and important words are shown in bold.

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

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

['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: Tokenizerl 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. 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?'

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... 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. 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'. 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.

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.

Replacing and Correcting Words

No text here

Introduction

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

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...

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...

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... 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 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... 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... 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. 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.

Creating a wordlist corpus

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... 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

', r'. ', r'. ', r'. ', r'.

Creating a chunked phrase corpus

*\.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'), ('. 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'), ('. ', '.')] ', '.')]

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('. *\.txt', cat_pattern=r'movie_(\w+)\.txt') >>> reader.categories() ['neg', 'pos'] >>> reader.fileids(categories=['neg']) ['movie_neg.txt'] >>> reader.fileids(categories=['pos']) ['movie_pos.txt'] 64 www.it-ebooks.info Chapter 3 How it works... The 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('. *\.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. 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(). We'll start with the plain text methods: def raw(self, fileids=None, categories=None): return ConllCorpusReader.raw(self, self._resolve(fileids, categories)) def words(self, fileids=None, categories=None): return ConllCorpusReader.words(self, self._resolve(fileids, categories)) def sents(self, fileids=None, categories=None): return ConllCorpusReader.sents(self, self._resolve(fileids, categories)) 70 www.it-ebooks.info Chapter 3 The ConllCorpusReader class does not recognize paragraphs, so there are no *_paras() methods. Next will be the code for the tagged and chunked methods, as follows: def tagged_words(self, fileids=None, categories=None): return ConllCorpusReader.tagged_words(self, self._resolve(fileids, categories)) def tagged_sents(self, fileids=None, categories=None): return ConllCorpusReader.tagged_sents(self, self._resolve(fileids, categories)) def chunked_words(self, fileids=None, categories=None,

Lazy corpus loading

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...

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('. 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.

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...

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. 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.

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).

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. >>> 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. 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.

Combining taggers with backoff tagging

>>> 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. 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

>>> 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. >>> 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. 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'. >>> 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

>>> 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.

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. >>> 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! >>> 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

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.

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... >>> 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

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. >>> 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%.

Training a tagger with NLTK-Trainer

If we use the treebank corpus, the command and output should look something like this: $ python train_tagger.py treebank loading treebank 3914 tagged sents, training on 3914 training AffixTagger with affix -3 and backoff <DefaultTagger: tag=- None-> training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff <AffixTagger: size=2536> training <class 'nltk.tag.sequential.BigramTagger'> tagger with backoff <UnigramTagger: size=4933> 114 www.it-ebooks.info Chapter 4 training <class 'nltk.tag.sequential.TrigramTagger'> tagger with backoff <BigramTagger: size=2325> evaluating TrigramTagger accuracy: 0.992372 dumping TrigramTagger to /Users/jacob/nltk_data/taggers/treebank_aubt. Here's how to do that with train_tagger.py, and also skip dumping a pickle file: $ python train_tagger.py treebank --fraction 0.75 --no-pickle loading treebank 3914 tagged sents, training on 2936 training AffixTagger with affix -3 and backoff <DefaultTagger: tag=- None-> training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff <AffixTagger: size=2287> training <class 'nltk.tag.sequential.BigramTagger'> tagger with backoff <UnigramTagger: size=4176> training <class 'nltk.tag.sequential.TrigramTagger'> tagger with backoff <BigramTagger: size=1836> evaluating TrigramTagger accuracy: 0.906082 How it works... Now let's try a unigram tagger: $ python train_tagger.py treebank --no-pickle --fraction 0.75 --sequential u loading treebank 3914 tagged sents, training on 2936 training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff <DefaultTagger: tag=-None-> evaluating UnigramTagger accuracy: 0.855603 Specifying --sequential u tells train_tagger.py to train with a unigram tagger. 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. So, if we want to use an affix of -2 as well as an affix of -3, you can do the following: $ python train_tagger.py treebank --no-pickle --default NN --fraction 0.75 -a -3 -a -2 loading treebank 3914 tagged sents, training on 2936 training AffixTagger with affix -3 and backoff <DefaultTagger: tag=NN> training AffixTagger with affix -2 and backoff <AffixTagger: size=2143> training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff <AffixTagger: size=248> training <class 'nltk.tag.sequential.BigramTagger'> tagger with backoff <UnigramTagger: size=5204> training <class 'nltk.tag.sequential.TrigramTagger'> tagger with backoff <BigramTagger: size=1838> evaluating TrigramTagger accuracy: 0.908696 117 www.it-ebooks.info Part-of-speech Tagging The order of multiple -a arguments matters, and if you switch the order, the results and accuracy will change, because the backoff order changes: $ python train_tagger.py treebank --no-pickle --default NN --fraction 0.75 -a -2 -a -3 loading treebank 3914 tagged sents, training on 2936 training AffixTagger with affix -2 and backoff <DefaultTagger: tag=NN> training AffixTagger with affix -3 and backoff <AffixTagger: size=606> training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff <AffixTagger: size=1313> training <class 'nltk.tag.sequential.BigramTagger'> tagger with backoff <UnigramTagger: size=4169> training <class 'nltk.tag.sequential.TrigramTagger'> tagger with backoff <BigramTagger: size=1829> evaluating TrigramTagger accuracy: 0.914367 You can also train a Brill tagger using the --brill argument. $ python train_tagger.py treebank --no-pickle --default NN

Extracting Chunks

No text here

Introduction

This is different from full parsing in that we're interested in standalone chunks, or phrases, instead of full parse trees (for more on parse trees, see https://en.wikipedia.org/wiki/Parse_tree).

Chunking and chinking with regular expressions

>{ ... '') >>> chunker.parse([('the', 'DT'), ('book', 'NN'), ('has', 'VBZ'), ('many', 'JJ'), ('chapters', 'NNS')]) Tree('S', [Tree('NP', [('the', 'DT'), ('book', 'NN')]), ('has', 'VBZ'), Tree('NP', [('many', 'JJ'), ('chapters', 'NNS')])]) The grammar tells the RegexpParser class that there are two rules for parsing NP chunks. Here's some code to demonstrate this: >>> from nltk.chunk.regexp import ChunkString, ChunkRule, ChinkRule >>> from nltk.tree import Tree >>> t = Tree('S', [('the', 'DT'), ('book', 'NN'), ('has', 'VBZ'), 126 www.it-ebooks.info Chapter 5 'many', 'JJ'), ('chapters', 'NNS')]) >>> cs = ChunkString(t) >>> cs <ChunkString: '<DT><NN><VBZ><JJ><NNS>'> >>> ur = ChunkRule('<DT><NN.><.*>*<NN. *>', 'chunk determiners and nouns') >>> ur.apply(cs) >>> cs <ChunkString: '{<DT><NN><VBZ><JJ><NNS>}'> >>> ir = ChinkRule('<VB. *>', 'chink verbs') >>> ir.apply(cs) >>> cs <ChunkString: '{<DT><NN>}<VBZ>{<JJ><NNS>}'> >>> cs.to_chunkstruct() Tree('S', [Tree('CHUNK', [('the', 'DT'), ('book', 'NN')]), ('has', 'VBZ'), Tree('CHUNK', [('many', 'JJ'), ('chapters', 'NNS')])]) The tree diagrams shown earlier can be drawn at each step by calling cs.to_chunkstruct().draw(). >>> from nltk.chunk import RegexpChunkParser >>> chunker = RegexpChunkParser([ur, ir]) >>> chunker.parse(t) Tree('S', [Tree('NP', [('the', 'DT'), ('book', 'NN')]), ('has', 'VBZ'), Tree('NP', [('many', 'JJ'), ('chapters', 'NNS')])]) 127 www.it-ebooks.info Extracting Chunks Parsing different chunk types If you wanted to parse a different chunk type, then you could pass that in as chunk_label to RegexpChunkParser. Here's the same code that we saw in the previous section, but instead of NP subtrees, we'll call them CP for custom phrase: >>> from nltk.chunk import RegexpChunkParser >>> chunker = RegexpChunkParser([ur, ir], chunk_label='CP') >>> chunker.parse(t) Tree('S', [Tree('CP', [('the', 'DT'), ('book', 'NN')]), ('has', 'VBZ'), Tree('CP', [('many', 'JJ'), ('chapters', 'NNS')])]) The RegexpParser class does this internally when you specify multiple phrase types. *>} ... '') >>> chunker.parse(t) Tree('S', [Tree('NP', [('the', 'DT'), ('book', 'NN')]), ('has', 'VBZ'), Tree('NP', [('many', 'JJ'), ('chapters', 'NNS')])]) In fact, you could reduce the two chunk patterns into a single pattern. *>} ... '') >>> chunker.parse(t) Tree('S', [Tree('NP', [('the', 'DT'), ('book', 'NN')]), ('has', 'VBZ'), Tree('NP', [('many', 'JJ'), ('chapters', 'NNS')])]) How you create and combine patterns is really up to you. *>', 'chunk nouns only after determiners') >>> cs = ChunkString(t) >>> cs <ChunkString: '<DT><NN><VBZ><JJ><NNS>'> >>> ctx.apply(cs) >>> cs <ChunkString: '<DT>{<NN>}<VBZ><JJ><NNS>'> >>> cs.to_chunkstruct() Tree('S', [('the', 'DT'), Tree('CHUNK', [('book', 'NN')]), ('has', 'VBZ'), ('many', 'JJ'), ('chapters', 'NNS')]) This example only chunks nouns that follow a determiner, therefore ignoring the noun that follows an adjective. *>} ... '') >>> chunker.parse(t) Tree('S', [('the', 'DT'), Tree('NP', [('book', 'NN')]), ('has', 'VBZ'), ('many', 'JJ'), ('chapters', 'NNS')]) See also In the next recipe, we'll cover merging and splitting chunks.

Merging and splitting chunks with regular expressions

> ... '') >>> sent = [('the', 'DT'), ('sushi', 'NN'), ('roll', 'NN'), ('was', 'VBD'), ('filled', 'VBN'), ('with', 'IN'), ('the', 'DT'), ('fish', 'NN')] >>> chunker.parse(sent) Tree('S', [Tree('NP', [('the', 'DT'), ('sushi', 'NN'), ('roll', 'NN')]), Tree('NP', [('was', 'VBD'), ('filled', 'VBN'), ('with', 'IN')]), Tree('NP', [('the', 'DT'), ('fish', 'NN')])]) 131 www.it-ebooks.info Extracting Chunks And the final tree of NP chunks is shown in the following diagram: S NP NP NP the DT sushi NN roll NN was VBD filled VBN with IN with DT fish NN How it works... Here's a step-by-step walkthrough of how the original sentence is modified by applying each rule: >>> from nltk.chunk.regexp import MergeRule, SplitRule >>> cs = ChunkString(Tree('S', sent)) >>> cs <ChunkString: '<DT><NN><NN><VBD><VBN><IN><DT><NN>'> >>> ur = ChunkRule('<DT><.>*<NN. *>', 'chunk determiner to noun') >>> ur.apply(cs) >>> cs <ChunkString: '{<DT><NN><NN><VBD><VBN><IN><DT><NN>}'> >>> sr1 = SplitRule('<NN. *>', 'split after noun') >>> sr1.apply(cs) >>> cs <ChunkString: '{<DT><NN>}{<NN>}{<VBD><VBN><IN><DT><NN>}'> >>> sr2 = SplitRule('<. *>', '<DT>', 'split before determiner') >>> sr2.apply(cs) >>> cs <ChunkString: '{<DT><NN>}{<NN>}{<VBD><VBN><IN>}{<DT><NN>}'> >>> mr = MergeRule('<NN. *>', 'merge nouns') >>> mr.apply(cs) >>> cs <ChunkString: '{<DT><NN><NN>}{<VBD><VBN><IN>}{<DT><NN>}'> >>> cs.to_chunkstruct() Tree('S', [Tree('CHUNK', [('the', 'DT'), ('sushi', 'NN'), ('roll', 'NN')]), Tree('CHUNK', [('was', 'VBD'), ('filled', 'VBN'), ('with', 'IN')]), Tree('CHUNK', [('the', 'DT'), ('fish', 'NN')])]) 132 www.it-ebooks.info

Expanding and removing chunks with regular expressions

Here's some code demonstrating the usage with the RegexpChunkParser class: >>> from nltk.chunk.regexp import ChunkRule, ExpandLeftRule, ExpandRightRule, UnChunkRule >>> from nltk.chunk import RegexpChunkParser >>> ur = ChunkRule('<NN>', 'single noun') >>> el = ExpandLeftRule('<DT>', '<NN>', 'get left determiner') >>> er = ExpandRightRule('<NN>', '<NNS>', 'get right plural noun') >>> un = UnChunkRule('<DT><NN. *>*', 'unchunk everything') >>> chunker = RegexpChunkParser([ur, el, er, un]) >>> sent = [('the', 'DT'), ('sushi', 'NN'), ('rolls', 'NNS')] >>> chunker.parse(sent) Tree('S', [('the', 'DT'), ('sushi', 'NN'), ('rolls', 'NNS')]) You'll notice that the end result is a flat sentence, which is exactly what we started with. Unchunk every chunk that is a determiner + noun + plural noun, resulting in the original sentence tree: S the DT sushi NN rolls NNS Here's the code showing each step: >>> from nltk.chunk.regexp import ChunkString >>> from nltk.tree import Tree >>> cs = ChunkString(Tree('S', sent)) >>> cs 135 www.it-ebooks.info

Partial parsing with regular expressions

*>} # chunk optional modal with verb ... ''') >>> from nltk.corpus import conll2000 >>> score = chunker.evaluate(conll2000.chunked_sents()) >>> score.accuracy() 0.6148573545757688 When we call evaluate() on the chunker argument, we give it a list of chunked sentences and get back a ChunkScore object, which can give us the accuracy of the chunker along with a number of other metrics. 137 www.it-ebooks.info Extracting Chunks There's more... You can also evaluate this chunker argument on the treebank_chunk corpus: >>> from nltk.corpus import treebank_chunk >>> treebank_score = chunker.evaluate(treebank_chunk.chunked_sents()) >>> treebank_score.accuracy() 0.49033970276008493 The treebank_chunk corpus is a special version of the treebank corpus that provides a chunked_sents() method. These can be useful to figure out how to improve your chunk grammar: >>> len(score.missed()) 47161 >>> len(score.incorrect()) 47967 >>> len(score.correct()) 119720 >>> len(score.guessed()) 120526 As you can see by the number of incorrect chunks, and by comparing guessed() and correct(), our chunker guessed that there were more chunks than actually existed.

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! For example, here's the TagChunker class using just a UnigramTagger class: >>> from nltk.tag import UnigramTagger >>> uni_chunker = TagChunker(train_chunks, tagger_ classes=[UnigramTagger]) >>> score = uni_chunker.evaluate(test_chunks) >>> score.accuracy() 0.9674925924335466 The tagger_classes argument will be passed directly into the backoff_tagger() function, which means they must be subclasses of SequentialBackoffTagger.

Classification-based chunking

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(). 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.

Extracting named entities

', 'NNP'), ('29', 'CD'), ('. ', '.')]) ', 'NNP'), ('29', 'CD'), ('. ', '.')])

Extracting proper noun chunks

Then, we can test this on the first tagged sentence of treebank_chunk to compare the results with the previous recipe: >>> chunker = RegexpParser(r'' ... NAME: ... {<NNP>+} ... '') >>> sub_leaves(chunker.parse(treebank_chunk.tagged_sents()[0]), 'NAME') [[('Pierre', 'NNP'), ('Vinken', 'NNP')], [('Nov. 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). Using it on the same tagged sentence as before, we get the following result: >>> from chunkers import PersonChunker >>> chunker = PersonChunker() >>> sub_leaves(chunker.parse(treebank_chunk.tagged_sents()[0]), 'PERSON') [[('Pierre', 'NNP')]] We no longer get Nov., but we've also lost Vinken, as it is not found in the names corpus.

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.

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. Then, we can see how it does on the first sentence of the treebank_chunk corpus: >>> from chunkers import ieer_chunked_sents, ClassifierChunker >>> from nltk.corpus import treebank_chunk >>> ieer_chunks = list(ieer_chunked_sents()) >>> len(ieer_chunks) 94 >>> chunker = ClassifierChunker(ieer_chunks[:80]) >>> chunker.parse(treebank_chunk.tagged_sents()[0]) Tree('S', [Tree('LOCATION', [('Pierre', 'NNP'), ('Vinken', 'NNP')]), (',', ','), Tree('DURATION', [('61', 'CD'), ('years', 'NNS')]), Tree('MEASURE', [('old', 'JJ')]), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'), Tree('DATE', [('Nov.

Training a chunker with NLTK-Trainer

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. Here's how to train a UnigramTagger based chunker: $ python train_chunker.py treebank_chunk --no-pickle --fraction 0.75 --sequential u loading treebank_chunk 4009 chunks, training on 3007 training u TagChunker evaluating TagChunker 157 www.it-ebooks.info Extracting Chunks ChunkParse score: IOB Accuracy: 96.7% Precision: 89.7% Recall: 93.1% F-Measure: 91.3% And here's how to twith additional BigramTagger and TrigramTagger classes: $ python train_chunker.py treebank_chunk --no-pickle --fraction 0.75 --sequential ubt loading treebank_chunk 4009 chunks, training on 3007 training ubt TagChunker evaluating TagChunker ChunkParse score: IOB Accuracy: 97.2% Precision: 91.6% Recall: 94.4% F-Measure: 93.0% You can also train a classifier-based chunker, which was covered in the previous recipe, Classification-based chunking. reader.ChunkedCorpusReader --no-pickle --fraction 0.75 loading corpora/treebank/tagged 51002 chunks, training on 38252 training ub TagChunker evaluating TagChunker ChunkParse score: 159 www.it-ebooks.info Extracting Chunks IOB Accuracy: 98.4% Precision: 97.7% Recall: 98.9% F-Measure: 98.3% Training on parse trees The train_chunker.py script supports two arguments that allow it to train on full parse trees from a corpus reader's parsed_sents() method instead of using chunked sentences. $ python train_chunker.py treebank --no-pickle --fraction 0.75 --flatten- deep-tree loading treebank flattening deep trees from treebank 3914 chunks, training on 2936 training ub TagChunker evaluating TagChunker ChunkParse score: IOB Accuracy: 72.4% Precision: 51.6% Recall: 52.2% F-Measure: 51.9% We use the treebank corpus instead of treebank_chunk, because it has full parse trees accessible with the parsed_sents() method. $ python train_chunker.py treebank --no-pickle --fraction 0.75 --shallow- tree loading treebank creating shallow trees from treebank 3914 chunks, training on 2936 training ub TagChunker evaluating TagChunker ChunkParse score: IOB Accuracy: 73.1% Precision: 60.0% Recall: 56.2% F-Measure: 58.0% These options are more useful for corpora that don't provide chunked sentences, such as cess_cat and cess_esp.

Transforming Chunks and Trees

No text here

Introduction

The tree transforms give you ways to modify and flatten deep parse trees. The functions detailed in these recipes modify data, as opposed to learning from it. A thorough knowledge of the data you want to transform, along with a few experiments, should help you decide which functions to apply and when.

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. The tag suffixes would then be PRP and PRP$: >>> filter_insignificant([('your', 'PRP$'), ('book', 'NN'), ('is', 'VBZ'), ('great', 'JJ')], tag_suffixes=['PRP', 'PRP$']) [('book', 'NN'), ('is', 'VBZ'), ('great', 'JJ')] Filtering insignificant words can be a good complement to stopword filtering for purposes such as search engine indexing and querying and text classification.

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.

Swapping verb phrases

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 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...

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.

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... There's more... You can pass trace=1 into transform_chunk() to get an output at each step: >>> from transforms import transform_chunk >>> transform_chunk([('the', 'DT'), ('book', 'NN'), ('of', 'IN'), ('recipes', 'NNS'), ('is', 'VBZ'), ('delicious', 'JJ')], trace=1) filter_insignificant : [('book', 'NN'), ('of', 'IN'), ('recipes', 'NNS'), ('is', 'VBZ'), ('delicious', 'JJ')] swap_verb_phrase : [('delicious', 'JJ'), ('book', 'NN'), ('of', 'IN'), ('recipes', 'NNS')] swap_infinitive_phrase : [('delicious', 'JJ'), ('recipes', 'NNS'), ('book', 'NN')] singularize_plural_noun : [('delicious', 'JJ'), ('recipe', 'NN'), ('book', 'NN')] [('delicious', 'JJ'), ('recipe', 'NN'), ('book', 'NN')] This shows you the result of each transform function, which is then passed in to the next transform until a final chunk is returned.

Converting a chunk tree to text

The obvious first step is to join all the words in the tree with a space: >>> from nltk.corpus import treebank_chunk >>> tree = treebank_chunk.chunked_sents()[0] >>> ' '.join([w for w, t in tree.leaves()]) 'Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .' 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.

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.

Creating a shallow tree

NNP NNP NP JJ VB NP PP-CLR NP-TMP , , will CD NNS DT NN IN NP join NNP CD Pierre Vinken old as the board 61 years DT JJ NN Nov. 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.

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.

Text Classification

No text here

Introduction

By examining the word usage in a piece of text, classifiers can decide what class label to assign to it. 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.

Bag of words feature extraction

The bag_of_words() function in featx.py looks like this: def bag_of_words(words): return dict([(word, True) for word in words]) We can use it with a list of words; in this case, the tokenized sentence the quick brown fox: >>> from featx import bag_of_words >>> bag_of_words(['the', 'quick', 'brown', 'fox']) {'quick': True, 'brown': True, 'the': True, 'fox': True} The resulting dict is known as a bag of words because the words are not in order, and it doesn't matter where in the list of words they occurred, or how many times they occurred. Here's an example where we filter the word the from the quick brown fox: >>> from featx import bag_of_words_not_in_set >>> bag_of_words_not_in_set(['the', 'quick', 'brown', 'fox'], ['the']) {'quick': True, 'brown': True, 'fox': True} As expected, the resulting dict has quick, brown, and fox, but not the. Using this function produces the same result as the previous example: >>> from featx import bag_of_non_stopwords >>> bag_of_non_stopwords(['the', 'quick', 'brown', 'fox']) {'quick': True, 'brown': True, 'fox': True} Here, the is a stopword, so it is not present in the returned dict. Using the same example words as we did earlier, we get all words plus every bigram: >>> from featx import bag_of_bigrams_words >>> bag_of_bigrams_words(['the', 'quick', 'brown', 'fox']) {'brown': True, ('brown', 'fox'): True, ('the', 'quick'): True, 'fox': True, ('quick', 'brown'): True, 'quick': True, 'the': True} You can change the maximum number of bigrams found by altering the keyword argument n. 190 www.it-ebooks.info

Training a Naive Bayes classifier

The split_label_feats() function in featx.py takes a mapping returned from label_feats_from_corpus() and splits each list of feature sets into labeled training and testing instances: def split_label_feats(lfeats, split=0.75): train_feats = [] test_feats = [] for label, feats in lfeats.items(): cutoff = int(len(feats) * split) train_feats.extend([(feat, label) for feat in feats[:cutoff]]) test_feats.extend([(feat, label) for feat in feats[cutoff:]]) return train_feats, test_feats Using these functions with the movie_reviews corpus gives us the lists of labeled feature sets we need to train and test a classifier: >>> from nltk.corpus import movie_reviews >>> from featx import label_feats_from_corpus, split_label_feats >>> movie_reviews.categories() ['neg', 'pos'] >>> lfeats = label_feats_from_corpus(movie_reviews) >>> lfeats.keys() 192 www.it-ebooks.info Chapter 7 dict_keys(['neg', 'pos']) >>> train_feats, test_feats = split_label_feats(lfeats, split=0.75) >>> len(train_feats) 1500 >>> len(test_feats) 500 So there are 1000 pos files, 1000 neg files, and we end up with 1500 labeled training instances and 500 labeled testing instances, each composed of equal parts of pos and neg. The following diagram shows other methods, which will be covered shortly: ClassifierI labels() classify() prob_classify() NaiveBayesClassifier most_informative_feature () s show_most_informative_features() train() There's more... We can test the accuracy of the classifier using nltk.classify.util.accuracy() and the test_feats variable created previously: >>> from nltk.classify.util import accuracy >>> accuracy(nb_classifier, test_feats) 0.728 194 www.it-ebooks.info Chapter 7 This tells us that the classifier correctly guessed the label of nearly 73% of the test feature sets. 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

The following is the code for training and evaluating the accuracy of a DecisionTreeClassifier class: >>> from nltk.classify import DecisionTreeClassifier >>> dt_classifier = DecisionTreeClassifier.train(train_feats, binary=True, entropy_cutoff=0.8, depth_cutoff=5, support_cutoff=30) >>> accuracy(dt_classifier, test_feats) 0.688 The DecisionTreeClassifier class can take much longer to train than the NaiveBayesClassifier class. The value of 'pos' is kept at 30, while the value of 'neg' is manipulated to show that when 'neg' is close to 'pos', entropy increases, but when it is closer to 1, entropy decreases: >>> from nltk.probability import FreqDist, MLEProbDist, entropy >>> fd = FreqDist({'pos': 30, 'neg': 10}) >>> entropy(MLEProbDist(fd)) 0.8112781244591328 >>> fd['neg'] = 25 >>> entropy(MLEProbDist(fd)) 0.9940302114769565 >>> fd['neg'] = 30 >>> entropy(MLEProbDist(fd)) 1.0 >>> fd['neg'] = 1 >>> entropy(MLEProbDist(fd)) 0.20559250818508304 What this all means is that if the label occurrence is very skewed one way or the other, the tree doesn't need to be refined because entropy/uncertainty is low.

Training a maximum entropy classifier

These parameters will be explained in more detail later: >>> from nltk.classify import MaxentClassifier >>> me_classifier = MaxentClassifier.train(train_feats, trace=0, max_ iter=1, min_lldelta=0.5) >>> accuracy(me_classifier, test_feats) 0.5 The reason this classifier has such a low accuracy is because I set the parameters such that it is unable to learn a more accurate model. A better algorithm is gis, which can be trained like this: >>> me_classifier = MaxentClassifier.train(train_feats, algorithm='gis', trace=0, max_iter=10, min_lldelta=0.5) >>> accuracy(me_classifier, test_feats) 0.722 The gis algorithm is a bit faster and generally more accurate than the default iis algorithm, and can be allowed to run for up to 10 iterations in a reasonable amount of time. How it works... Like the previous classifiers, MaxentClassifier inherits from ClassifierI, as shown in the following diagram: 202 www.it-ebooks.info Chapter 7 ClassifierI labels() classify() prob_classify() MaxentClassifier show_most_informative_feature () s train() Depending on the algorithm, MaxentClassifier.train() calls one of the training functions in the nltk.classify.maxent module. There's more... Like the NaiveBayesClassifier class, you can see the most informative features by calling the show_most_informative_features() method: >>> me_classifier.show_most_informative_features(n=4) -0.740 worst==True and label is 'pos' 0.740 worst==True and label is 'neg' 0.715 bad==True and label is 'neg' -0.715 bad==True and label is 'pos' The numbers shown are the weights for each feature. Or, if megam can be found in the standard executable paths, NLTK will configure it automatically: >>> me_classifier = MaxentClassifier.train(train_feats, algorithm='megam', trace=0, max_iter=10) [Found megam: /usr/local/bin/megam] >>> accuracy(me_classifier, test_feats) 0.86799999999999999 204 www.it-ebooks.info

Training scikit-learn classifiers

Refer to the earlier recipe, Training a Naive Bayes classifier, for details on constructing train_feats and test_feats: >>> from nltk.classify.scikitlearn import SklearnClassifier >>> from sklearn.naive_bayes import MultinomialNB >>> sk_classifier = SklearnClassifier(MultinomialNB()) >>> sk_classifier.train(train_feats) <SklearnClassifier(MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True))> Now that we have a trained classifier, we can evaluate the accuracy: >>> accuracy(sk_classifier, test_feats) 0.83 How it works... 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. Our features are actually already binarized, because the feature values are True or False: >>> from sklearn.naive_bayes import BernoulliNB >>> sk_classifier = SklearnClassifier(BernoulliNB()) >>> sk_classifier.train(train_feats) <SklearnClassifier(BernoulliNB(alpha=1.0, binarize=0.0, class_ prior=None, fit_prior=True))> >>> accuracy(sk_classifier, test_feats) 0.812 Clearly, the sklearn algorithm performs better than NLTK's Naive Bayes implementation. >>> 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. Here are some examples of using the sklearn implementations: >>> from sklearn.svm import SVC >>> sk_classifier = SklearnClassifier(svm.SVC()) >>> sk_classifier.train(train_feats) <SklearnClassifier(SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0, kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False))> >>> accuracy(sk_classifier, test_feats) 0.69 >>> from sklearn.svm import LinearSVC >>> sk_classifier = SklearnClassifier(LinearSVC()) >>> sk_classifier.train(train_feats) <SklearnClassifier(LinearSVC(C=1.0, class_weight=None, dual=True, fit_ intercept=True, intercept_scaling=1, loss='l2', multi_class='ovr', penalty='l2', random_state=None, tol=0.0001, verbose=0))> >>> accuracy(sk_classifier, test_feats) 0.864 >>> from sklearn.svm import NuSVC >>> sk_classifier = SklearnClassifier(svm.NuSVC()) >>> sk_classifier.train(train_feats) /Users/jacob/py3env/lib/python3.3/site-packages/scipy/sparse/ compressed.py:119: UserWarning: indptr array has non-integer dtype (float64) % self.indptr.dtype.name) <SklearnClassifier(NuSVC(cache_size=200, coef0=0.0, degree=3, gamma=0.0, kernel='rbf', max_iter=-1, nu=0.5, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False))> >>> accuracy(sk_classifier, test_feats) 0.882 209 www.it-ebooks.info

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(). Here's an example usage with nb_classifier and test_feats we created in the Training a Naive Bayes classifier recipe earlier: >>> from classification import precision_recall >>> nb_precisions, nb_recalls = precision_recall(nb_classifier, test_feats) >>> nb_precisions['pos'] 0.6413612565445026 >>> nb_precisions['neg'] 0.9576271186440678 >>> nb_recalls['pos'] 0.98 >>> nb_recalls['neg'] 0.452 This tells us that while the NaiveBayesClassifier class can correctly identify most of the pos feature sets (high recall), it also classifies many of the neg feature sets as pos (low precision). There's more... Let's try it with the MaxentClassifier class of GIS, which we trained in the Training a maximum entropy classifier recipe: >>> me_precisions, me_recalls = precision_recall(me_classifier, test_feats) >>> me_precisions['pos'] 0.6456692913385826 >>> me_precisions['neg'] 0.9663865546218487 >>> me_recalls['pos'] 0.984 >>> me_recalls['neg'] 0.46 212 www.it-ebooks.info Chapter 7 This classifier is just as biased as the NaiveBayesClassifier class. Now, let's try the SklearnClassifier class of NuSVC from the previous recipe, Training scikit-learn classifiers: >>> sk_precisions, sk_recalls = precision_recall(sk_classifier, test_feats) >>> sk_precisions['pos'] 0.9063829787234042 >>> sk_precisions['neg'] 0.8603773584905661 >>> sk_recalls['pos'] 0.852 >>> sk_recalls['neg'] 0.912 In this case, the label bias is much less significant, and the reason is that the SklearnClassifier class of NuSVC weighs its features according to its own internal model.

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. >>> from featx import high_information_words, bag_of_words_in_set >>> labels = movie_reviews.categories() >>> labeled_words = [(l, movie_reviews.words(categories=[l])) for l in labels] >>> high_info_words = set(high_information_words(labeled_words)) >>> feat_det = lambda words: bag_of_words_in_set(words, high_info_ words) >>> lfeats = label_feats_from_corpus(movie_reviews, feature_ detector=feat_det) >>> train_feats, test_feats = split_label_feats(lfeats) Now that we have new training and testing feature sets, let's train and evaluate a NaiveBayesClassifier class: >>> nb_classifier = NaiveBayesClassifier.train(train_feats) >>> accuracy(nb_classifier, test_feats) 0.91 >>> nb_precisions, nb_recalls = precision_recall(nb_classifier, test_feats) >>> nb_precisions['pos'] 0.8988326848249028 215 www.it-ebooks.info Text Classification >>> nb_precisions['neg'] 0.9218106995884774 >>> nb_recalls['pos'] 0.924 >>> nb_recalls['neg'] 0.896 While the neg precision and pos recall have both decreased somewhat, neg recall and pos precision have increased drastically. The MaxentClassifier class with high information words Let's evaluate the MaxentClassifier class using the high information words feature sets: >>> me_classifier = MaxentClassifier.train(train_feats, algorithm='gis', trace=0, max_iter=10, min_lldelta=0.5) >>> accuracy(me_classifier, test_feats) 0.912 >>> me_precisions, me_recalls = precision_recall(me_classifier, test_feats) >>> me_precisions['pos'] 0.8992248062015504 >>> me_precisions['neg'] 0.9256198347107438 >>> me_recalls['pos'] 0.928 >>> me_recalls['neg'] 0.896 This also led to significant improvements for MaxentClassifier. The DecisionTreeClassifier class with high information words Now, let's evaluate the DecisionTreeClassifier class: >>> dt_classifier = DecisionTreeClassifier.train(train_feats, binary=True, depth_cutoff=20, support_cutoff=20, entropy_cutoff=0.01) >>> accuracy(dt_classifier, test_feats) 0.68600000000000005 >>> dt_precisions, dt_recalls = precision_recall(dt_classifier, test_ feats) >>> dt_precisions['pos'] 0.6741573033707865 >>> dt_precisions['neg'] 217 www.it-ebooks.info Text Classification 0.69957081545064381 >>> dt_recalls['pos'] 0.71999999999999997 >>> dt_recalls['neg'] 0.65200000000000002 The accuracy is about the same, even with a larger depth_cutoff, and smaller support_cutoff and entropy_cutoff. The SklearnClassifier class with high information words Let's evaluate the LinearSVC SklearnClassifier with the same train_feats function: >>> sk_classifier = SklearnClassifier(LinearSVC()).train(train_feats) >>> accuracy(sk_classifier, test_feats) 0.86 >>> sk_precisions, sk_recalls = precision_recall(sk_classifier, test_feats) >>> sk_precisions['pos'] 0.871900826446281 >>> sk_precisions['neg'] 0.8488372093023255 >>> sk_recalls['pos'] 0.844 >>> sk_recalls['neg'] 0.876 Its accuracy before was 86.4%, so we actually got a very slight decrease.

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. >>> from classification import MaxVoteClassifier >>> mv_classifier = MaxVoteClassifier(nb_classifier, dt_classifier, me_classifier, sk_classifier) >>> mv_classifier.labels() ['neg', 'pos'] >>> accuracy(mv_classifier, test_feats) 0.894 >>> mv_precisions, mv_recalls = precision_recall(mv_classifier, test_feats) >>> mv_precisions['pos'] 0.9156118143459916 >>> mv_precisions['neg'] 0.8745247148288974 >>> mv_recalls['pos'] 0.868 >>> mv_recalls['neg'] 0.92 These metrics are about on-par with the best sklearn classifiers, as well as the MaxentClassifier and NaiveBayesClassifier classes with high information features.

Classifying with multiple binary classifiers

It defaults to using bag_of_words() as its feature_detector, but we will be overriding this using bag_of_words_in_set() to use only the high information words: def reuters_train_test_feats(feature_detector=bag_of_words): train_feats = [] test_feats = [] for fileid in reuters.fileids(): if fileid.startswith('training'): featlist = train_feats else: # fileid.startswith('test') featlist = test_feats feats = feature_detector(reuters.words(fileid)) labels = reuters.categories(fileid) featlist.append((feats, labels)) return train_feats, test_feats We can use these two functions to get a list of multi-labeled training and testing feature sets. 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 classification import train_binary_classifiers >>> trainf = lambda train_feats: SklearnClassifier(LogisticRegressi on()).train(train_feats) >>> labelset = set(reuters.categories()) >>> classifiers = train_binary_classifiers(trainf, multi_train_feats, labelset) >>> len(classifiers) 90 223 www.it-ebooks.info Text Classification Also in classification.py, we can define a MultiBinaryClassifier class, which takes a list of labeled classifiers of the form [(label, classifier)], where the classifier is assumed to be a binary classifier that either returns the label or something else if the label doesn't apply. 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. import collections from nltk import metrics def multi_metrics(multi_classifier, test_feats): mds = [] 224 www.it-ebooks.info Chapter 7 refsets = collections.defaultdict(set) testsets = collections.defaultdict(set) for i, (feat, labels) in enumerate(test_feats): for label in labels: refsets[label].add(i) guessed = multi_classifier.classify(feat) for label in guessed: testsets[label].add(i) mds.append(metrics.masi_distance(set(labels), guessed)) avg_md = sum(mds) / float(len(mds)) precisions = {} recalls = {} for label in multi_classifier.labels(): precisions[label] = metrics.precision(refsets[label], testsets[label]) recalls[label] = metrics.recall(refsets[label], testsets[label]) return precisions, recalls, avg_md Using this with the multi_classifier function we just created gives us the following results: >>> from classification import multi_metrics >>> multi_precisions, multi_recalls, avg_md = multi_metrics (multi_classifier, multi_test_feats) >>> avg_md 0.23310715863026216 So our average masi distance isn't too bad. Let's take a look at a few precisions and recalls: >>> multi_precisions['soybean'] 0.7857142857142857 >>> multi_recalls['soybean'] 0.3333333333333333 >>> len(reuters.fileids(categories=['soybean'])) 111 225 www.it-ebooks.info Text Classification >>> multi_precisions['sunseed'] 1.0 >>> multi_recalls['sunseed'] 2.0 >>> len(reuters.fileids(categories=['crude'])) 16 In general, the labels that have more feature sets will have higher precision and recall, and those with less feature sets will

Training a classifier with NLTK-Trainer

Here's an example of running train_classifier.py on the movie_reviews corpus: $ python train_classifier.py movie_reviews loading movie_reviews 2 labels: ['neg', 'pos'] using bag of words feature extraction 2000 training feats, 2000 testing feats training NaiveBayes classifier accuracy: 0.967000 neg precision: 1.000000 neg recall: 0.934000 neg f-measure: 0.965874 pos precision: 0.938086 pos recall: 1.000000 pos f-measure: 0.968054 dumping NaiveBayesClassifier to ~/nltk_data/classifiers/movie_ reviews_NaiveBayes.pickle We can use the --no-pickle argument to skip saving the classifier and the --fraction argument to limit the training set and evaluate the classifier against a test set. 228 www.it-ebooks.info Chapter 7 $ python train_classifier.py movie_reviews --no-pickle --fraction 0.75 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 You can see that not only do we get accuracy, we also get the precision and recall of each class, like we covered earlier in the recipe, Measuring precision and recall of a classifier. Here's an example using sentences from the movie_reviews corpus: $ python train_classifier.py movie_reviews --no-pickle --fraction 0.75 --instances sents loading movie_reviews 2 labels: ['neg', 'pos'] using bag of words feature extraction 50820 training feats, 16938 testing feats training NaiveBayes classifier accuracy: 0.638623 230 www.it-ebooks.info Chapter 7 neg precision: 0.694942 neg recall: 0.470786 neg f-measure: 0.561313 pos precision: 0.610546 pos recall: 0.800580 pos f-measure: 0.692767 To use paragraphs instead of files or sentences, you can do --instances paras. 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. In the Training scikit-learn classifiers recipe, we covered the LogisticRegression classifier, so here's how to do it with train_classifier.py: $ python train_classifier.py movie_reviews --no-pickle --fraction 0.75 --classifier sklearn.LogisticRegression loading movie_reviews 2 labels: ['neg', 'pos'] using bag of words feature extraction 1500 training feats, 500 testing feats training sklearn.LogisticRegression with {'penalty': 'l2', 'C': 1.0} using dtype bool training sklearn.LogisticRegression classifier accuracy: 0.856000 neg precision: 0.847656 neg recall: 0.868000 neg f-measure: 0.857708 pos precision: 0.864754 pos recall: 0.844000 pos f-measure: 0.854251 232 www.it-ebooks.info Chapter 7 SVMs SVM classifiers were introduced in the Training scikit-learn classifiers recipe, and can also be used with train_classifier.py. Here's the parameters for LinearSVC: $ python train_classifier.py movie_reviews --no-pickle --fraction 0.75 --classifier

Distributed Processing and Handling Large Datasets

No text here

Introduction

However, there are times when you have a lot of data to process and want to take advantage of multiple CPUs, multicore CPUs, and even multiple computers. Or, you might want to store frequencies and probabilities in a persistent, shared database so multiple processes can access it simultaneously. For the first case, we'll be using execnet to do parallel and distributed processing with NLTK. For the second case, you'll learn how to use the Redis data structure server/database to store frequency distributions and more.

Distributed tagging with execnet

>>> import execnet, remote_tag, nltk.tag, nltk.data >>> from nltk.corpus import treebank >>> import pickle >>> pickled_tagger = pickle.dumps(nltk.data.load(nltk.tag._POS_ TAGGER)) >>> gw = execnet.makegateway() >>> channel = gw.remote_exec(remote_tag) >>> channel.send(pickled_tagger) >>> channel.send(treebank.sents()[0]) >>> tagged_sentence = channel.receive() >>> tagged_sentence == treebank.tagged_sents()[0] True >>> gw.exit() Visually, the communication process looks like this: remote_tag Local Process pickled Tagger sentence tagged sentence How it works... 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. Here's the code: >>> import itertools >>> gw1 = execnet.makegateway() >>> gw2 = execnet.makegateway() >>> ch1 = gw1.remote_exec(remote_tag) >>> ch1.send(pickled_tagger) >>> ch2 = gw2.remote_exec(remote_tag) >>> ch2.send(pickled_tagger) >>> mch = execnet.MultiChannel([ch1, ch2]) >>> queue = mch.make_receive_queue() >>> channels = itertools.cycle(mch) >>> for sentence in treebank.sents()[:4]: ... channel = next(channels) ... channel.send(sentence) >>> tagged_sentences = [] >>> for i in range(4): ... channel, tagged_sentence = queue.get() ... tagged_sentences.append(tagged_sentence) >>> len(tagged_sentences) 4 >>> gw1.exit() >>> gw2.exit() In the example code, we're only sending four sentences, but in real-life, you'd want to send thousands.

Distributed chunking with execnet

Finally, we exit the gateway: >>> import execnet, remote_chunk >>> import nltk.data, nltk.tag, nltk.chunk >>> import pickle >>> from nltk.corpus import treebank_chunk 242 www.it-ebooks.info Chapter 8 >>> tagger = pickle.dumps(nltk.data.load(nltk.tag._POS_TAGGER)) >>> chunker = pickle.dumps(nltk.data.load(nltk.chunk._MULTICLASS_NE_ CHUNKER)) >>> gw = execnet.makegateway() >>> channel = gw.remote_exec(remote_chunk) >>> channel.send(tagger) >>> channel.send(chunker) >>> channel.send(treebank_chunk.sents()[0]) >>> chunk_tree = pickle.loads(channel.receive()) >>> chunk_tree Tree('S', [Tree('PERSON', [('Pierre', 'NNP')]), Tree('ORGANIZATION', [('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.

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

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. Here's an outline of each method that uses a specific Redis command: __len__(): This uses the hlen command to get the number of elements in the f hash map __contains__(): This uses the hexists command to check if an element exists f in the hash map __getitem__(): This uses the hget command to get a value from the hash map f __setitem__(): This uses the hset command to set a value in the hash map f __delitem__(): This uses the hdel command to remove a value from the f hash map keys(): This uses the hkeys command to get all the keys in the hash map f values(): This uses the hvals command to get all the values in the hash map f items(): This uses the hgetall command to get a dictionary containing all the f keys and values in the hash map clear(): This uses the delete command to remove the entire hash map f from Redis Extending collections.MutableMapping provides a number of other dict compatible methods based on the previous methods, such as update() and setdefault(), so we don't have to implement them ourselves. 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. After that, it works just like a ConditionalFreqDist: >>> from redis import Redis >>> from redisprob import RedisConditionalHashFreqDist >>> r = Redis() >>> rchfd = RedisConditionalHashFreqDist(r, 'condhash') >>> rchfd.N() 0 >>> rchfd.conditions() [] >>> rchfd['cond1']['foo'] += 1 >>> rchfd.N() 1 >>> rchfd['cond1']['foo'] 1 >>> rchfd.conditions() ['cond1'] >>> rchfd.clear() 252 www.it-ebooks.info

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.

Distributed word scoring with Redis and execnet

>>> from dist_featx import score_words >>> from nltk.corpus import movie_reviews >>> labels = movie_reviews.categories() >>> labelled_words = [(l, movie_reviews.words(categories=[l])) for l in labels] >>> word_scores = score_words(labelled_words) >>> len(word_scores) 39767 >>> topn_words = word_scores.keys(end=1000) 257 www.it-ebooks.info Distributed Processing and Handling Large Datasets >>> topn_words[0:5] [b'bad', b',', b'and', b'? The code itself is as follows: import itertools, execnet, remote_word_count from nltk.metrics import BigramAssocMeasures from redis import Redis from redisprob import RedisHashFreqDist, RedisConditionalHashFreqDist from rediscollections import RedisOrderedDict def score_words(labelled_words, score_fn=BigramAssocMeasures.chi_sq, host='localhost', specs=[('popen', 2)]): gateways = [] channels = [] for spec, count in specs: for i in range(count): gw = execnet.makegateway(spec) gateways.append(gw) channel = gw.remote_exec(remote_word_count) channel.send((host, 'word_fd', 'label_word_fd')) channels.append(channel) cyc = itertools.cycle(channels) for label, words in labelled_words: channel = next(cyc) channel.send((label, list(words))) for channel in channels: channel.send('done') assert 'done' == channel.receive() channel.waitclose(5) for gateway in gateways: gateway.exit() r = Redis(host) fd = RedisHashFreqDist(r, 'word_fd') cfd = RedisConditionalHashFreqDist(r, 'label_word_fd') word_scores = RedisOrderedDict(r, 'word_scores') n_xx = cfd.N() for label in cfd.conditions(): n_xi = cfd[label].N() 259 www.it-ebooks.info Distributed Processing and Handling Large Datasets for word, n_ii in cfd[label].iteritems(): word = word.decode() n_ix = fd[word] if n_ii and n_ix and n_xi and n_xx: score = score_fn(n_ii, (n_ix, n_xi), n_xx) word_scores[word] = score return word_scores Note that this scoring method will only be accurate for comparing two labels. The remote_word_count.py module looks like the following code: from redis import Redis from redisprob import RedisHashFreqDist, RedisConditionalHashFreqDist if __name__ == '__channelexec__': host, fd_name, cfd_name = channel.receive() r = Redis(host) fd = RedisHashFreqDist(r, fd_name) cfd = RedisConditionalHashFreqDist(r, cfd_name) for data in channel: if data == 'done': channel.send('done') break label, words = data for word in words: fd[word] += 1 cfd[label][word] += 1 260 www.it-ebooks.info Chapter 8 You'll notice that this is not a pure module, as it requires being able to import both redis and redisprob.

Parsing Specific Data Types

No text here

Introduction

These libraries can be great complements to NLTK: dateutil provides datetime parsing and timezone conversion f lxml and BeautifulSoup can parse, clean, and convert HTML f charade and UnicodeDammit can detect and convert text character encoding f These libraries can be useful for preprocessing text before passing it to an NLTK object, or postprocessing text that has been processed and extracted using NLTK.

Parsing dates and times with dateutil

How to do it... Let's dive into a few parsing examples: >>> from dateutil import parser >>> parser.parse('Thu Sep 25 10:36:28 2010') datetime.datetime(2010, 9, 25, 10, 36, 28) >>> parser.parse('Thursday, 25. September 2010 10:36AM') datetime.datetime(2010, 9, 25, 10, 36) >>> parser.parse('9/25/2010 10:36:28') datetime.datetime(2010, 9, 25, 10, 36, 28) >>> parser.parse('9/25/2010') datetime.datetime(2010, 9, 25, 0, 0) >>> parser.parse('2010-09-25T10:36:28Z') datetime.datetime(2010, 9, 25, 10, 36, 28, tzinfo=tzutc()) As you can see, all it takes is importing the parser module and calling the parse() function with a datetime string. >>> parser.parse('25/9/2010', dayfirst=True) datetime.datetime(2010, 9, 25, 0, 0) Another ordering issue can occur with two-digit years. But if you pass yearfirst=True into parse(), it will be parsed to the year 2010: >>> parser.parse('10-9-25') datetime.datetime(2025, 10, 9, 0, 0) >>> parser.parse('10-9-25', yearfirst=True) datetime.datetime(2010, 9, 25, 0, 0) There's more... But if fuzzy=True, then a datetime object can usually be returned: >>> try: ... parser.parse('9/25/2010 at about 10:36AM') ... except ValueError: ... 'cannot parse' 'cannot parse' >>> parser.parse('9/25/2010 at about 10:36AM', fuzzy=True) datetime.datetime(2010, 9, 25, 10, 36) See also In the next recipe, we'll use the tz module of dateutil to do timezone lookup and conversion.

Timezone lookup and conversion

This can be done by calling tz.tzutc(), and you can check that the offset is 0 by calling the utcoffset() method with a UTC datetime object: >>> from dateutil import tz >>> tz.tzutc() tzutc() >>> import datetime >>> tz.tzutc().utcoffset(datetime.datetime.utcnow()) datetime.timedelta(0) 266 www.it-ebooks.info Chapter 9 To get tzinfo objects for other timezones, you can pass in a timezone file path to the gettz() function: >>> tz.gettz('US/Pacific') tzfile('/usr/share/zoneinfo/US/Pacific') >>> tz.gettz('US/Pacific').utcoffset(datetime.datetime.utcnow()) datetime.timedelta(-1, 61200) >>> tz.gettz('Europe/Paris') tzfile('/usr/share/zoneinfo/Europe/Paris') >>> tz.gettz('Europe/Paris').utcoffset(datetime.datetime.utcnow()) datetime.timedelta(0, 7200) You can see that the UTC offsets are timedelta objects, where the first number is days and the second number is seconds. >>> pst = tz.gettz('US/Pacific') Y >>> dt = datetime.datetime(2010, 9, 25, 10, 36) >>> dt.tzinfo >>> dt.astimezone(tz.tzutc()) Traceback (most recent call last): File "/usr/lib/python2.6/doctest.py", line 1248, in __run compileflags, 1) in test.globs File "<doctest __main__[22]>", line 1, in <module> dt.astimezone(tz.tzutc()) ValueError: astimezone() cannot be applied to a naive datetime >>> dt.replace(tzinfo=pst) datetime.datetime(2010, 9, 25, 10, 36, tzinfo=tzfile('/usr/share/ zoneinfo/US/Pacific')) >>> dt.replace(tzinfo=pst).astimezone(tz.tzutc()) datetime.datetime(2010, 9, 25, 17, 36, tzinfo=tzutc()) The tzfile paths vary across operating systems, so your tzfile paths may differ from the examples. There's more... You can pass a tzinfos keyword argument into the dateutil parser to detect the otherwise unrecognized timezones: >>> parser.parse('Wednesday, Aug 4, 2010 at 6:30 p.m. (CDT)', fuzzy=True) datetime.datetime(2010, 8, 4, 18, 30) >>> tzinfos = {'CDT': tz.gettz('US/Central')} >>> parser.parse('Wednesday, Aug 4, 2010 at 6:30 p.m. (CDT)', fuzzy=True, tzinfos=tzinfos) datetime.datetime(2010, 8, 4, 18, 30, tzinfo=tzfile('/usr/share/ zoneinfo/US/Central')) In the first instance, we get a naïve datetime since the timezone is not recognized.

Extracting URLs from HTML with lxml

Here's some code to demonstrate: >>> from lxml import html >>> doc = html.fromstring('Hello world') >>> links = list(doc.iterlinks()) >>> len(links) 1 269 www.it-ebooks.info Parsing Specific Data Types >>> (el, attr, link, pos) = links[0] >>> attr 'href' >>> link '/world' >>> pos 0 How it works... lxml parses the HTML into an ElementTree. We can make it absolute by calling the make_links_absolute() method with a base URL before extracting the links: >>> doc.make_links_absolute('http://hello') >>> abslinks = list(doc.iterlinks()) >>> (el, attr, link, pos) = abslinks[0] >>> link 'http://hello/world' Extracting links directly If you don't want to do anything other than extract links, you can call the iterlinks() function with an HTML string: >>> links = list(html.iterlinks('Hello world')) >>> links[0][2] '/world' Parsing HTML from URLs or files Instead of parsing an HTML string using the fromstring() function, you can call the parse() function with a URL or filename; for example, html.parse('http://my/url') or html.parse('/path/to/file').

Cleaning and stripping HTML

How to do it... We can use the clean_html() function in the lxml.html.clean module to remove unnecessary HTML tags and embedded JavaScript from an HTML string: >>> import lxml.html.clean >>> lxml.html.clean.clean_html('<html><head></head><body onload=loadfunc()>my text</body></html>') ' <body>my text</body> ' The result is much cleaner and easier to deal with.

Converting HTML entities with BeautifulSoup

It's quite simple: create an instance of BeautifulSoup given a string containing HTML entities, then get the string attribute: >>> from bs4 import BeautifulSoup >>> BeautifulSoup('<').string '<' >>> BeautifulSoup('&').string '&' However, the reverse is not true. You first create the soup with an HTML string, call the findAll() method with 'a' to get all anchor tags, and pull out the 'href' attribute to get the URLs: >>> from bs4 import BeautifulSoup >>> soup = BeautifulSoup('Hello world') >>> [a['href'] for a in soup.findAll('a')] ['/world'] See also In the Extracting URLs from HTML with lxml recipe, we covered how to use lxml to extract URLs from an HTML string, and we also covered the Cleaning and stripping HTML recipe after that.

Detecting and converting character encodings

-*- coding: utf-8 -*- import charade def detect(s): try: if isinstance(s, str): return charade.detect(s.encode()) else: return charade.detect(s) except UnicodeDecodeError: return charade.detect(s.encode('utf-8')) def convert(s): if isinstance(s, str): s = s.encode() encoding = detect(s)['encoding'] if encoding == 'utf-8': return s.decode() else: return s.decode(encoding) 274 www.it-ebooks.info Chapter 9 And here's some example code using detect() to determine character encoding: >>> import encoding >>> encoding.detect('ascii') {'confidence': 1.0, 'encoding': 'ascii'} >>> encoding.detect('abcdé') {'confidence': 0.505, 'encoding': 'utf-8'} >>> encoding.detect(bytes('\222\222\223\225', 'latin-1')) {'confidence': 0.5, 'encoding': 'windows-1252'} To convert a string to a standard unicode encoding, call encoding.convert().