

12. Language Engineering

This chapter will cover evaluation, trade-offs between methods that create large vs small models (e.g. n-gram tagging vs Brill tagging).

12.1 Problems with Tagging

- what context is relevant?
- how best to combine taggers?
- sensitivity to lexical identity?
- ngram tagging produces large models, uninterpretable cf Brill tagging, which has smaller, linguistically-interpretable models

12.1.1 Exercises

1. **Tagger context** (advanced):

N-gram taggers choose a tag for a token based on its text and the tags of the $n-1$ preceding tokens. This is a common context to use for tagging, but certainly not the only possible context.

- a) Construct a new tagger, sub-classed from **SequentialTagger**, that uses a different context. If your tagger's context contains multiple elements, then you should combine them in a tuple. Some possibilities for elements to include are: (i) the current word or a previous word; (ii) the length of the current word text or of the previous word; (iii) the first letter of the current word or the previous word; or (iv) the previous tag. Try to choose context elements that you believe will help the tagger decide which tag is appropriate. Keep in mind the trade-off between more specific taggers with accurate results; and more general taggers with broader coverage. Combine your tagger with other taggers using the backoff method.
- b) How does the combined tagger's accuracy compare to the basic tagger?
- c) How does the combined tagger's accuracy compare to the combined taggers you created in the previous exercise?

2. **Reverse sequential taggers** (advanced): Since sequential taggers tag tokens in order, one at a time, they can only use the predicted tags to the *left* of the current token to decide what tag to assign to a token. But in some cases, the *right* context may provide more useful

information than the left context. A reverse sequential tagger starts with the last word of the sentence and, proceeding in right-to-left order, assigns tags to words on the basis of the tags it has already predicted to the right. By reversing texts at appropriate times, we can use NLTK's existing sequential tagging classes to perform reverse sequential tagging: reverse the training text before training the tagger; and reverse the text being tagged both before and after.

- a) Use this technique to create a bigram reverse sequential tagger.
 - b) Measure its accuracy on a tagged section of the Brown corpus. Be sure to use a different section of the corpus for testing than you used for training.
 - c) How does its accuracy compare to a left-to-right bigram tagger, using the same training data and test data?
3. **Alternatives to backoff:** Create a new kind of tagger that combines several taggers using a new mechanism other than backoff (e.g. voting). For robustness in the face of unknown words, include a regexp tagger, a unigram tagger that removes a small number of prefix or suffix characters until it recognizes a word, or an n-gram tagger that does not consider the text of the token being tagged.

12.2 Evaluating Taggers

As we experiment with different taggers, it is important to have an objective performance measure. Fortunately, we already have manually verified training data (the original tagged corpus), so we can use that to score the accuracy of a tagger, and to perform systematic error analysis.

12.2.1 Scoring Accuracy

Consider the following sentence from the Brown Corpus. The 'Gold Standard' tags from the corpus are given in the second column, while the tags assigned by a unigram tagger appear in the third column. Two mistakes made by the unigram tagger are italicized.

Table 1: Evaluating Taggers

Sentence	Gold Standard	Unigram Tagger
The	at	at
President	nn-tl	nn-tl
said	vbd	vbd
he	pps	pps
will	md	md
ask	vb	vb
Congress	np	np
to	to	to
increase	vb	<i>nn</i>
grants	nns	nns

Table 1: Evaluating Taggers

Sentence	Gold Standard	Unigram Tagger
to	in	<i>to</i>
states	nns	nns
for	in	in
vocational	jj	jj
rehabilitation	nn	nn
.	.	.

The tagger correctly tagged 14 out of 16 words, so it gets a score of 14/16, or 87.5%. Of course, accuracy should be judged on the basis of a larger sample of data. NLTK provides a function called `tag.accuracy` to automate the task. In the simplest case, we can test the tagger using the same data it was trained on:

```
>>> acc = tag.accuracy(unigram_tagger, train_sents)
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 81.8%
```

However, testing a language processing system over its training data is unwise. A system which simply memorized the training data would get a perfect score without doing any linguistic modeling. Instead, we would like to reward systems that make good generalizations, so we should test against *unseen data*, and replace `train_sents` above with `unseen_sents`. We can then define the two sets of data as follows:

```
>>> train_sents = list(brown.tagged('a'))[:500]
>>> unseen_sents = list(brown.tagged('a'))[500:600] # sents 500-599
```

Now we train the tagger using `train_sents` and evaluate it using `unseen_sents`, as follows:

```
>>> unigram_tagger = tag.Unigram(backoff=nn_cd_tagger)
>>> unigram_tagger.train(train_sents)
>>> acc = tag.accuracy(unigram_tagger, unseen_sents)
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 74.7%
```

The accuracy scores produced by this evaluation method are lower, but they give a more realistic picture of the performance of the tagger. Note that the performance of any statistical tagger is highly dependent on the quality of its training set. In particular, if the training set is too small, it will not be able to reliably estimate the most likely tag for each word. Performance will also suffer if the training set is significantly different from the texts we wish to tag.

In the process of developing a tagger, we can use the accuracy score as an objective measure of the improvements made to the system. Initially, the accuracy score will go up quickly as we fix obvious shortcomings of the tagger. After a while, however, it becomes more difficult and improvements are small.

12.2.2 Baseline Performance

It is difficult to interpret an accuracy score in isolation. For example, is a person who scores 25% in a test likely to know a quarter of the course material? If the test is made up of 4-way multiple choice

questions, then this person has not performed any better than chance. Thus, it is clear that we should *interpret* an accuracy score relative to a *baseline*. The choice of baseline is somewhat arbitrary, but it usually corresponds to minimal knowledge about the domain.

In the case of tagging, a possible baseline score can be found by tagging every word with **NN**, the most likely tag.

```
>>> baseline_tagger = tag.Default('nn')
>>> acc = tag.accuracy(baseline_tagger, brown.tagged('a'))
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 13.1%
```

Unfortunately this is not a very good baseline. There are many high-frequency words which are not nouns. Instead we could use the standard unigram tagger to get a baseline of 75%. However, this does not seem fully legitimate: the unigram's model covers all words seen during training, which hardly seems like 'minimal knowledge'. Instead, let's only permit ourselves to store tags for the most frequent words.

The first step is to identify the most frequent words in the corpus, and for each of these words, identify the most likely tag:

```
>>> from nltk_lite.probability import *
>>> wordcounts = FreqDist()
>>> wordtags = ConditionalFreqDist()
>>> for sent in brown.tagged('a'):
...     for (w,t) in sent:
...         wordcounts.inc(w)      # count the word
...         wordtags[w].inc(t)    # count the word's tag
>>> frequent_words = wordcounts.sorted_samples()[:1000]
```

Now we can create a lookup table (a dictionary) which maps words to likely tags, just for these high-frequency words. We can then define a new baseline tagger which uses this lookup table:

```
>>> table = dict((word, wordtags[word].max()) for word in frequent_words)
>>> baseline_tagger = tag.Lookup(table, tag.Default('nn'))
>>> acc = tag.accuracy(baseline_tagger, brown.tagged('a'))
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 72.5%
```

This, then, would seem to be a reasonable baseline score for a tagger. When we build new taggers, we will only credit ourselves for performance exceeding this baseline.

12.2.3 Error Analysis

While the accuracy score is certainly useful, it does not tell us how to improve the tagger. For this we need to undertake error analysis. For instance, we could construct a *confusion matrix*, with a row and a column for every possible tag, and entries that record how often a word with tag T_i is incorrectly tagged as T_j . Another approach is to analyze the context of the errors, which is what we do now.

Consider the following program, which catalogs all errors along with the tag on the left and their frequency of occurrence.

```
>>> errors = {}
>>> for i in range(len(unseen_sents)):
...     raw_sent = tag.untag(unseen_sents[i])
```

```

...     test_sent = list(unigram_tagger.tag(raw_sent))
...     unseen_sent = unseen_sents[i]
...     for j in range(len(test_sent)):
...         if test_sent[j][1] != unseen_sent[j][1]:
...             test_context = test_sent[j-1:j+1]
...             gold_context = unseen_sent[j-1:j+1]
...             if None not in test_context:
...                 pair = (tuple(test_context), tuple(gold_context))
...                 if pair not in errors:
...                     errors[pair] = 0
...                 errors[pair] += 1

```

The `errors` dictionary has keys of the form $((t1, t2), (g1, g2))$, where $(t1, t2)$ are the test tags, and $(g1, g2)$ are the gold-standard tags. The values in the `errors` dictionary are simple counts of how often each error occurred. With some further processing, we construct the list `counted_errors` containing tuples consisting of counts and errors, and then do a reverse sort to get the most significant errors first:

```

>>> counted_errors = [(errors[k], k) for k in errors.keys()]
>>> counted_errors.sort()
>>> counted_errors.reverse()
>>> for err in counted_errors[:5]:
...     print err
(32, (((), ()))
(5, (((('the', 'at'), ('Rev.', 'nn')),
        (('the', 'at'), ('Rev.', 'np')))))
(5, (((('Assemblies', 'nn'), ('of', 'in')),
        (('Assemblies', 'nns-tl'), ('of', 'in-tl')))))
(4, (((('of', 'in'), ('God', 'nn')),
        (('of', 'in-tl'), ('God', 'np-tl')))))
(3, (((('to', 'to'), ('form', 'nn')),
        (('to', 'to'), ('form', 'vb')))))

```

The fifth line of output records the fact that there were 3 cases where the unigram tagger mistakenly tagged a verb as a noun, following the word *to*. (We encountered the inverse of this mistake for the word *increase* in the above evaluation table, where the unigram tagger tagged *increase* as a verb instead of a noun since it occurred more often in the training data as a verb.) Here, when *form* appears after the word *to*, it is invariably a verb. Evidently, the performance of the tagger would improve if it was modified to consider not just the word being tagged, but also the tag of the word on the left. Such taggers are known as bigram taggers, and we consider them in the next section.

12.2.4 Exercises

1. **Evaluating a Unigram Tagger:** Apply our evaluation methodology to the unigram tagger developed in the previous section. Discuss your findings.

12.3 Sparse Data and Smoothing

[Introduction to NLTK's support for statistical estimation.]

12.4 The Brill Tagger

A potential issue with n-gram taggers is the size of their n-gram table (or language model). If tagging is to be employed in a variety of language technologies deployed on mobile computing devices, it is important to strike a balance between model size and tagger performance. An n-gram tagger with backoff may store trigram and bigram tables, large sparse arrays which may have hundreds of millions of entries.

A second issue concerns context. The only information an n-gram tagger considers from prior context is tags, even though words themselves might be a useful source of information. It is simply impractical for n-gram models to be conditioned on the identities of words in the context. In this section we examine Brill tagging, a statistical tagging method which performs very well using models that are only a tiny fraction of the size of n-gram taggers.

Brill tagging is a kind of *transformation-based learning*. The general idea is very simple: guess the tag of each word, then go back and fix the mistakes. In this way, a Brill tagger successively transforms a bad tagging of a text into a better one. As with n-gram tagging, this is a *supervised learning* method, since we need annotated training data. However, unlike n-gram tagging, it does not count observations but compiles a list of transformational correction rules.

The process of Brill tagging is usually explained by analogy with painting. Suppose we were painting a tree, with all its details of boughs, branches, twigs and leaves, against a uniform sky-blue background. Instead of painting the tree first then trying to paint blue in the gaps, it is simpler to paint the whole canvas blue, then “correct” the tree section by over-painting the blue background. In the same fashion we might paint the trunk a uniform brown before going back to over-paint further details with even finer brushes. Brill tagging uses the same idea: begin with broad brush strokes then fix up the details, with successively finer changes. The following table illustrates this process, first tagging with the unigram tagger, then fixing the errors.

Table 2: Steps in Brill Tagging

Sentence:	Gold:	Uni-gram:	Replace nn with vb when the previous word is to	Replace to with in when the next tag is nns
The	at	at		
President	nn-tl	nn-tl		
said	vbd	vbd		
he	pps	pps		
will	md	md		
ask	vb	vb		
Congress	np	np		
to	to	to		
increase	vb	nn	vb	
grants	nns	nns		
to	in	to	to	in
states	nns	nns		
for	in	in		
vocational	jj	jj		

Table 2: Steps in Brill Tagging

Sentence:	Gold:	Uni-gram:	Replace nn with vb when the previous word is to	Replace to with in when the next tag is nns
rehabilitation	nn	nn		

In this table we see two rules. All such rules are generated from a template of the following form: form “replace T_1 with T_2 in the context C ”. Typical contexts are the identity or the tag of the preceding or following word, or the appearance of a specific tag within 2-3 words of of the current word. During its training phase, the tagger guesses values for T_1 , T_2 and C , to create thousands of candidate rules. Each rule is scored according to its net benefit: the number of incorrect tags that it corrects, less the number of correct tags it incorrectly modifies. This process is best illustrated by a listing of the output from the NLTK Brill tagger (here run on tagged Wall Street Journal text from the Penn Treebank).

```

Loading tagged data...
Training unigram tagger: [accuracy: 0.820940]
Training Brill tagger on 37168 tokens...

Iteration 1: 1482 errors; ranking 23989 rules;
  Found: "Replace POS with VBZ if the preceding word is tagged PRP"
  Apply: [changed 39 tags: 39 correct; 0 incorrect]

Iteration 2: 1443 errors; ranking 23662 rules;
  Found: "Replace VBP with VB if one of the 3 preceding words is tagged MD"
  Apply: [changed 36 tags: 36 correct; 0 incorrect]

Iteration 3: 1407 errors; ranking 23308 rules;
  Found: "Replace VBP with VB if the preceding word is tagged TO"
  Apply: [changed 24 tags: 23 correct; 1 incorrect]

Iteration 4: 1384 errors; ranking 23057 rules;
  Found: "Replace NN with VB if the preceding word is to"
  Apply: [changed 67 tags: 22 correct; 45 incorrect]
  ...
Iteration 20: 1138 errors; ranking 20717 rules;
  Found: "Replace RBR with JJR if one of the 2 following words is tagged NNS"
  Apply: [changed 14 tags: 10 correct; 4 incorrect]

Iteration 21: 1128 errors; ranking 20569 rules;
  Found: "Replace VBD with VBN if the preceding word is tagged VBD"
  [insufficient improvement; stopping]

Brill accuracy: 0.835145

```

Brill taggers have another interesting property: the rules are linguistically interpretable. Compare this with the n-gram taggers, which employ a potentially massive table of n-grams. We cannot learn much from direct inspection of such a table, in comparison to the rules learned by the Brill tagger.

12.4.1 Exercises

1. Try the Brill tagger demonstration, as follows:

```
from nltk_lite.tag import brill
brill.demo()
```

2. Consult the documentation for the demo function, using `help(brill.demo)`. Experiment with the tagger by setting different values for the parameters. Is there any trade-off between training time (corpus size) and performance?
3. (Advanced) Inspect the diagnostic files created by the tagger `rules.out` and `errors.out`. Obtain the demonstration code (`nltk_lite/tag/brill.py`) and create your own version of the Brill tagger.
 - a) Delete some of the rule templates, based on what you learned from inspecting `rules.out`.
 - b) Add some new rule templates which employ contexts that might help to correct the errors you saw in `errors.out`.

We are grateful to Christopher Maloof for developing NLTK's Brill tagger, and Trevor Cohn for developing NLTK's HMM tagger.

12.5 The HMM Tagger

[Overview of NLTK's HMM tagger.]

12.6 Evaluating Chunk Parsers

An easy way to evaluate a chunk parser is to take some already chunked text, strip off the chunks, rechunk it, and compare the result with the original chunked text. The `ChunkScore.score()` function takes the correctly chunked sentence as its first argument, and the newly chunked version as its second argument, and compares them. It reports the fraction of actual chunks that were found (recall), the fraction of hypothesized chunks that were correct (precision), and a combined score, the F-measure (the harmonic mean of precision and recall).

A number of different metrics can be used to evaluate chunk parsers. We will concentrate on a class of metrics that can be derived from two sets:

- **guessed**: The set of chunks returned by the chunk parser.
- **correct**: The correct set of chunks, as defined in the test corpus.

The evaluation method we will use comes from the field of information retrieval, and considers the performance of a document retrieval system. We will set up an analogy between the correct set of chunks and a user's so-called "information need", and between the set of returned chunks and a system's returned documents. Consider the following diagram.

The intersection of these sets defines four regions: the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Two standard measures are *precision*, the fraction of guessed chunks that were correct $TP/(TP+FP)$, and *recall*, the fraction of correct chunks that were identified $TP/(TP+FN)$. A third measure, the *F measure*, is the harmonic mean of precision and recall, i.e. $1/(0.5/Precision + 0.5/Recall)$.

During evaluation of a chunk parser, it is useful to flatten a chunk structure into a tree consisting only of a root node and leaves:

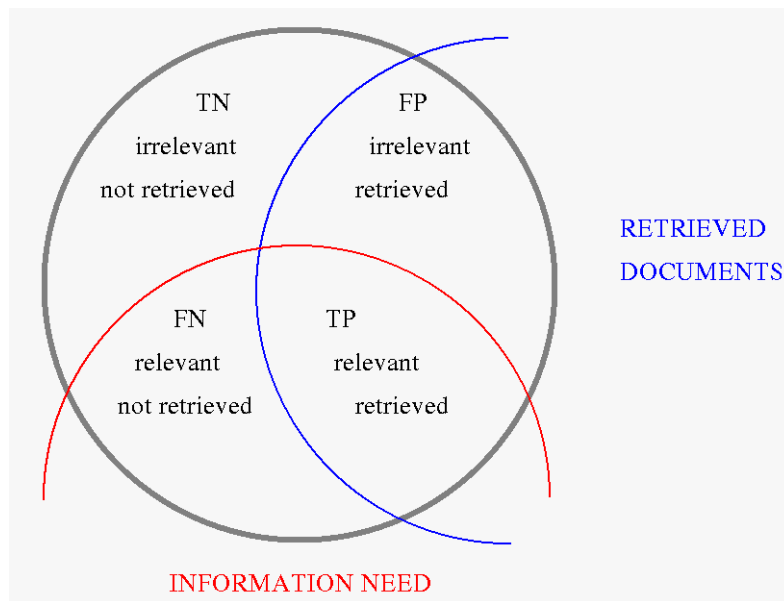


Figure 1: True and False Positives and Negatives

```
>>> correct = tree.chunk(
...     "[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]")
>>> correct.flatten()
(S: ('the', 'DT') ('little', 'JJ') ('cat', 'NN') ('sat', 'VBD')
 ('on', 'IN') ('the', 'DT') ('mat', 'NN'))
```

We run a chunker over this flattened data, and compare the resulting chunked sentences with the originals, as follows:

```
>>> from nltk_lite import parse
>>> chunkscore = parse.ChunkScore()
>>> rule = parse.ChunkRule('<PRP|DT|POS|JJ|CD|N.*>+',
...     "Chunk items that often occur in NPs")
>>> chunkparser = parse.RegexpChunk([rule], chunk_node='NP')
>>> guess = chunkparser.parse(correct.flatten())
>>> chunkscore.score(correct, guess)
>>> print chunkscore
ChunkParse score:
Precision: 100.0%
Recall:    100.0%
F-Measure: 100.0%
```

ChunkScore is a class for scoring chunk parsers. It can be used to evaluate the output of a chunk parser, using precision, recall, f-measure, missed chunks, and incorrect chunks. It can also be used to combine the scores from the parsing of multiple texts. This is quite useful if we are parsing a text one sentence at a time. The following program listing shows a typical use of the **ChunkScore** class. In this example, **chunkparser** is being tested on each sentence from the Wall Street Journal tagged files.

```
>>> from itertools import islice
>>> rule = parse.ChunkRule('<DT|JJ|NN>+', "Chunk sequences of DT, JJ, and NN")
```

```

>>> chunkparser = parse.RegexpChunk([rule], chunk_node='NP')
>>> chunkscore = parse.ChunkScore()
>>> for chunk_struct in islice(treebank.chunked(), 10):
...     test_sent = chunkparser.parse(chunk_struct.flatten())
...     chunkscore.score(chunk_struct, test_sent)
>>> print chunkscore
ChunkParse score:
Precision: 48.6%
Recall:    34.0%
F-Measure: 40.0%

```

The overall results of the evaluation can be viewed by printing the `ChunkScore`. Each evaluation metric is also returned by an accessor method: `precision()`, `recall`, `f_measure`, `missed`, and `incorrect`. The `missed` and `incorrect` methods can be especially useful when trying to improve the performance of a chunk parser. Here are the missed chunks:

```

>>> from random import shuffle
>>> missed = chunkscore.missed()
>>> shuffle(missed)
>>> print missed[:10]
[(('A', 'DT'), ('Lorillard', 'NNP'), ('spokeswoman', 'NN')),
 (('even', 'RB'), ('brief', 'JJ'), ('exposures', 'NNS')),
 (('its', 'PRP$'), ('Micronite', 'NN'), ('cigarette', 'NN'), ('filters', 'NNS')),
 (('30', 'CD'), ('years', 'NNS')),
 (('workers', 'NNS'),),
 (('preliminary', 'JJ'), ('findings', 'NNS')),
 (('Medicine', 'NNP'),),
 (('Consolidated', 'NNP'), ('Gold', 'NNP'), ('Fields', 'NNP'), ('PLC', 'NNP')),
 (('its', 'PRP$'), ('Micronite', 'NN'), ('cigarette', 'NN'), ('filters', 'NNS')),
 (('researchers', 'NNS'),)]

```

Here are the incorrect chunks:

```

>>> incorrect = chunkscore.incorrect()
>>> shuffle(incorrect)
>>> print incorrect[:10]
[(('New', 'JJ'), ('York-based', 'JJ')),
 (('Micronite', 'NN'), ('cigarette', 'NN')),
 (('a', 'DT'), ('forum', 'NN'), ('likely', 'JJ')),
 (('later', 'JJ'),),
 (('preliminary', 'JJ'),),
 (('New', 'JJ'), ('York-based', 'JJ')),
 (('resilient', 'JJ'),),
 (('group', 'NN'),),
 (('the', 'DT'),),
 (('Micronite', 'NN'), ('cigarette', 'NN'))]

```

As we saw with tagging, we need to interpret the performance scores for a chunker relative to a baseline. Perhaps the most naive chunking method is to classify every tag in the training data as to whether it occurs inside or outside a chunk more often. We can do this easily using a chunked corpus and a conditional frequency distribution as shown below:

```

>>> from nltk_lite.probability import ConditionalFreqDist
>>> from nltk_lite.parse import Tree

```

```

>>> import re
>>> cfdist = ConditionalFreqDist()
>>> chunk_data = list(treebank.chunked())
>>> split = len(chunk_data)*9/10
>>> train, test = chunk_data[:split], chunk_data[split:]
>>> for chunk_struct in train:
...     for constituent in chunk_struct:
...         if isinstance(constituent, Tree):
...             for (word, tag) in constituent.leaves():
...                 cfdist[tag].inc(True)
...         else:
...             (word, tag) = constituent
...             cfdist[tag].inc(False)

>>> chunk_tags = [tag for tag in cfdist.conditions() if cfdist[tag].max() == True]
>>> chunk_tags = [re.sub(r'(\W)', r'\\1', tag) for tag in chunk_tags]
>>> tag_pattern = '<' + '|'.join(chunk_tags) + '>+'
>>> print 'Chunking:', tag_pattern
Chunking: <PRP\$|VBG\|NN|POS|WDT|JJ|WP|DT|\#|\$|NN|FW|PRP|NNS|NNP|LS|PDT|RBS|CD|EX

```

Now, in the evaluation phase we chunk any sequence of those tags:

```

>>> rule = parse.ChunkRule(tag_pattern, 'Chunk any sequence involving commonly chunked tags')
>>> chunkparser = parse.RegexpChunk([rule], chunk_node='NP')
>>> chunkscore = parse.ChunkScore()
>>> for chunk_struct in test:
...     test_sent = chunkparser.parse(chunk_struct.flatten())
...     chunkscore.score(chunk_struct, test_sent)
>>> print chunkscore
ChunkParse score:
Precision: 90.7%
Recall: 94.0%
F-Measure: 92.3%

```

12.6.1 Exercises

1. **Chunker Evaluation:** Carry out the following evaluation tasks for any of the chunkers you have developed earlier. (Note that most chunking corpora contain some internal inconsistencies, such that any reasonable rule-based approach will produce errors.)
 - a) Evaluate your chunker on 100 sentences from a chunked corpus, and report the precision, recall and F-measure.
 - b) Use the `chunkscore.missed()` and `chunkscore.incorrect()` methods to identify the errors made by your chunker. Discuss.
 - c) Compare the performance of your chunker to the baseline chunker discussed in the evaluation section of this chapter.
2. (Advanced) **Transformation-Based Chunking:** Apply the n-gram and Brill tagging methods to IOB chunk tagging. Instead of assigning POS tags to words, here we will assign IOB tags to the POS tags. E.g. if the tag `DT` (determiner) often occurs at the start of

a chunk, it will be tagged **B** (begin). Evaluate the performance of these chunking methods relative to the regular expression chunking methods covered in this chapter.

12.7 Conclusions

[To be written]

About this document...

This chapter is a draft from *Introduction to Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2006 the authors. It is distributed with the *Natural Language Toolkit* [<http://nltk.sourceforge.net>], Version 0.7.1, under the terms of the *Creative Commons Attribution-ShareAlike License* [<http://creativecommons.org/licenses/by-sa/2.5/>].