4
Part-of-speech Tagging

In this chapter, we will cover the following recipes:

- Default tagging
- Training a unigram part-of-speech tagger
- Combining taggers with backoff tagging
- Training and combining ngram taggers
- Creating a model of likely word tags
- Tagging with regular expressions
- Affix tagging
- Training a Brill tagger
- Training the TnT tagger
- Using WordNet for tagging
- Tagging proper names
- Classifier-based tagging
- Training a tagger with NLTK-Trainer

Introduction

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

Part-of-speech tagging is a necessary step before chunking, which is covered in *Chapter 5*, *Extracting Chunks*. Without the part-of-speech tags, a chunker cannot know how to extract phrases from a sentence. But with part-of-speech tags, you can tell a chunker how to identify phrases based on tag patterns.

You can also use part-of-speech tags for grammar analysis and word sense disambiguation. For example, the word *duck* could refer to a bird, or it could be a verb indicating a downward motion. Computers cannot know the difference without additional information, such as part-of-speech tags. For more on word sense disambiguation, refer to the URL `https://en.wikipedia.org/wiki/Word_sense_disambiguation`.

Most of the taggers we'll cover are trainable. They use a list of tagged sentences as their training data, such as what you get from the `tagged_sents()` method of a `TaggedCorpusReader` class (see the *Creating a part-of-speech tagged word corpus* recipe in *Chapter 3*, *Creating Custom Corpora,* for more details). With these training sentences, the tagger generates an internal model that will tell it how to tag a word. Other taggers use external data sources or match word patterns to choose a tag for a word.

All taggers in NLTK are in the `nltk.tag` package and inherit from the `TaggerI` base class. `TaggerI` requires all subclasses to implement a `tag()` method, which takes a list of words as input and returns a list of tagged words as output. `TaggerI` also provides an `evaluate()` method for evaluating the accuracy of the tagger (covered at the end of the *Default tagging* recipe). Many taggers can also be combined into a backoff chain, so that if one tagger cannot tag a word, the next tagger is used, and so on.

Default tagging

Default tagging provides a baseline for part-of-speech tagging. It simply assigns the same part-of-speech tag to every token. We do this using the `DefaultTagger` class. This tagger is useful as a last-resort tagger, and provides a baseline to measure accuracy improvements.

Getting ready

We're going to use the `treebank` corpus for most of this chapter because it's a common standard and is quick to load and test. But everything we do should apply equally well to `brown`, `conll2000`, and any other part-of-speech tagged corpus.

How to do it...

The `DefaultTagger` class takes a single argument, the tag you want to apply. We'll give it `NN`, which is the tag for a singular noun. `DefaultTagger` is most useful when you choose the most common part-of-speech tag. Since nouns tend to be the most common types of words, a noun tag is recommended.

```
>>> from nltk.tag import DefaultTagger
>>> tagger = DefaultTagger('NN')
>>> tagger.tag(['Hello', 'World'])
[('Hello', 'NN'), ('World', 'NN')]
```

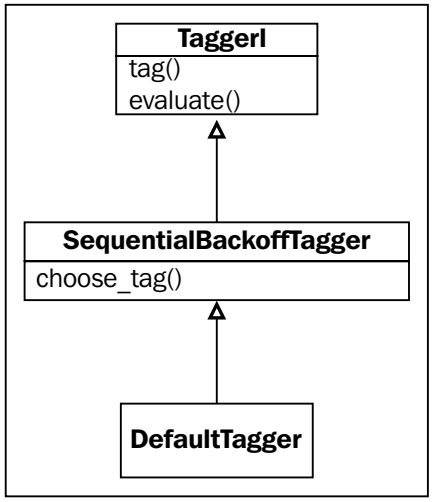
Every tagger has a `tag()` method that takes a list of tokens, where each token is a single word. This list of tokens is usually a list of words produced by a word tokenizer (see *Chapter 1*, *Tokenizing Text and WordNet Basics*, for more on tokenization). As you can see, `tag()` returns a list of tagged tokens, where a tagged token is a tuple of `(word, tag)`.

How it works...

`DefaultTagger` is a subclass of `SequentialBackoffTagger`. Every subclass of `SequentialBackoffTagger` must implement the `choose_tag()` method, which takes three arguments:

- The list of tokens
- The index of the current token whose tag we want to choose
- The history, which is a list of the previous tags

`SequentialBackoffTagger` implements the `tag()` method, which calls the `choose_tag()` method of the subclass for each index in the tokens list while accumulating a history of the previously tagged tokens. This history is the reason for the *Sequential* in `SequentialBackoffTagger`. We'll get to the backoff portion of the name in the *Combining taggers with backoff tagging* recipe. Here's a diagram showing the inheritance tree:



The `choose_tag()` method of `DefaultTagger` is very simple: it returns the tag we gave it at the time of initialization. It does not care about the current token or the history.

There's more...

There are a lot of different tags you could give to the `DefaultTagger` class. You can find a complete list of possible tags for the `treebank` corpus at `http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`. These tags are also documented in *Appendix*, *Penn Treebank Part-of-speech Tags*.

Evaluating accuracy

To know how accurate a tagger is, you can use the `evaluate()` method, which takes a list of tagged tokens as a gold standard to evaluate the tagger. Using our default tagger created earlier, we can evaluate it against a subset of the `treebank` corpus tagged sentences.

```
>>> from nltk.corpus import treebank
>>> test_sents = treebank.tagged_sents()[3000:]
>>> tagger.evaluate(test_sents)
0.14331966328512843
```

So, by just choosing `NN` for every tag, we can achieve 14 % accuracy testing on one-fourth of the `treebank` corpus. Of course, accuracy will be different if you choose a different default tag. We'll be reusing these same `test_sents` for evaluating more taggers in the upcoming recipes.

Tagging sentences

`TaggerI` also implements a `tag_sents()` method that can be used to tag a list of sentences, instead of a single sentence. Here's an example of tagging two simple sentences:

```
>>> tagger.tag_sents([['Hello', 'world', '.'], ['How', 'are', 'you',
'?']])
[[('Hello', 'NN'), ('world', 'NN'), ('.', 'NN')], [('How', 'NN'),
('are', 'NN'), ('you', 'NN'), ('?', 'NN')]]
```

The result is a list of two tagged sentences, and of course, every tag is `NN` because we're using the `DefaultTagger` class. The `tag_sents()` method can be quiet useful if you have many sentences you wish to tag all at once.

Untagging a tagged sentence

Tagged sentences can be untagged using `nltk.tag.untag()`. Calling this function with a tagged sentence will return a list of words without the tags.

```
>>> from nltk.tag import untag
>>> untag([('Hello', 'NN'), ('World', 'NN')])
['Hello', 'World']
```

See also

For more on tokenization, see *Chapter 1*, *Tokenizing Text and WordNet Basics*. And to learn more about tagged sentences, see the *Creating a part-of-speech tagged word corpus* recipe in *Chapter 3*, *Creating Custom Corpora*. For a complete list of part-of-speech tags found in the `treebank` corpus, see *Appendix*, *Penn Treebank Part-of-speech Tags*.

Training a unigram part-of-speech tagger

A **unigram** generally refers to a single token. Therefore, a unigram tagger only uses a single word as its context for determining the part-of-speech tag.

`UnigramTagger` inherits from `NgramTagger`, which is a subclass of `ContextTagger`, which inherits from `SequentialBackoffTagger`. In other words, `UnigramTagger` is a context-based tagger whose context is a single word, or unigram.

How to do it...

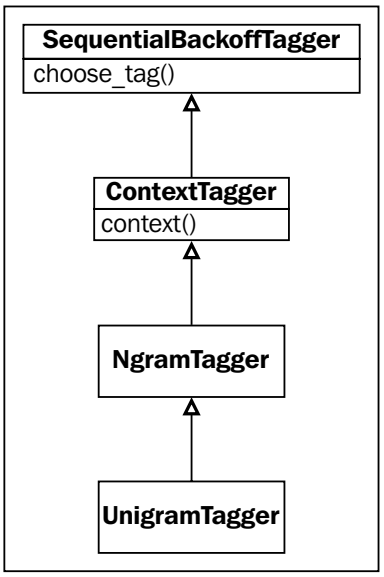
`UnigramTagger` can be trained by giving it a list of tagged sentences at initialization.

```
>>> 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.', '29',
'.']
>>> tagger.tag(treebank.sents()[0])
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'),
('years', 'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join',
'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'),
('nonexecutive', 'JJ'), ('director', 'NN'), ('Nov.', 'NNP'), ('29',
'CD'), ('.', '.')]
```

We use the first 3000 tagged sentences of the `treebank` corpus as the training set to initialize the `UnigramTagger` class. Then, we see the first sentence as a list of words, and can see how it is transformed by the `tag()` function into a list of tagged tokens.

How it works...

`UnigramTagger` builds a context model from the list of tagged sentences. Because `UnigramTagger` inherits from `ContextTagger`, instead of providing a `choose_tag()` method, it must implement a `context()` method, which takes the same three arguments as `choose_tag()`. The result of `context()` is, in this case, the word token. The context token is used to create the model, and also to look up the best tag once the model is created. Here's an inheritance diagram showing each class, starting at `SequentialBackoffTagger`:



Let's see how accurate the `UnigramTagger` class is on the test sentences (see the previous recipe for how `test_sents` is created).

```
>>> tagger.evaluate(test_sents)
0.8588819339520829
```

It has almost 86 % accuracy for a tagger that only uses single word lookup to determine the part-of-speech tag. All accuracy gains from here on will be much smaller.

Actual accuracy values may change each time you run the code. This is because the default iteration order in Python 3 is random. To get consistent accuracy values, run Python with the `PYTHONHASHSEED` environment variable set to `0` or any positive integer. For example:

```
$ PYTHONHASHSEED=0 python chapter4.py
```

All accuracy values in this book were calculated with `PYTHONHASHSEED=0`.

There's more...

The model building is actually implemented in `ContextTagger`. Given the list of tagged sentences, it calculates the frequency that a tag has occurred for each context. The tag with the highest frequency for a context is stored in the model.

Overriding the context model

All taggers that inherit from `ContextTagger` can take a pre-built model instead of training their own. This model is simply a Python `dict` mapping a context key to a tag. The context keys will depend on what the `ContextTagger` subclass returns from its `context()` method. For `UnigramTagger`, context keys are individual words. But for other `NgramTagger` subclasses, the context keys will be tuples.

Here's an example where we pass a very simple model to the `UnigramTagger` class instead of a training set.

```
>>> tagger = UnigramTagger(model={'Pierre': 'NN'})
>>> tagger.tag(treebank.sents()[0])
[('Pierre', 'NN'), ('Vinken', None), (',', None), ('61', None),
('years', None), ('old', None), (',', None), ('will', None), ('join',
None), ('the', None), ('board', None), ('as', None), ('a', None),
('nonexecutive', None), ('director', None), ('Nov.', None), ('29',
None), ('.', None)]
```

Since the model only contained the context key `Pierre`, only the first word got a tag. Every other word got `None` as the tag since the context word was not in the model. So, unless you know exactly what you are doing, let the tagger train its own model instead of passing in your own.

One good case for passing a self-created model to the `UnigramTagger` class is for when you have a dictionary of words and tags, and you know that every word should always map to its tag. Then, you can put this `UnigramTagger` as your first backoff tagger (covered in the next recipe) to look up tags for unambiguous words.

Minimum frequency cutoff

The `ContextTagger` class uses frequency of occurrence to decide which tag is most likely for a given context. By default, it will do this even if the context word and tag occurs only once. If you'd like to set a minimum frequency threshold, then you can pass a `cutoff` value to the `UnigramTagger` class.

```
>>> tagger = UnigramTagger(train_sents, cutoff=3)
>>> tagger.evaluate(test_sents)
0.7757392618173969
```

In this case, using `cutoff=3` has decreased accuracy, but there may be times when a cutoff is a good idea.

See also

In the next recipe, we'll cover **backoff tagging** to combine taggers, and in the *Creating a model of likely word tags* recipe, we'll learn how to statistically determine tags for very common words.

Combining taggers with backoff tagging

Backoff tagging is one of the core features of `SequentialBackoffTagger`. It allows you to chain taggers together so that if one tagger doesn't know how to tag a word, it can pass the word on to the next backoff tagger. If that one can't do it, it can pass the word on to the next backoff tagger, and so on until there are no backoff taggers left to check.

How to do it...

Every subclass of `SequentialBackoffTagger` can take a backoff keyword argument whose value is another instance of a `SequentialBackoffTagger`. So, we'll use the `DefaultTagger` class from the *Default tagging* recipe in this chapter as the backoff to the `UnigramTagger` class covered in the previous recipe, *Training a unigram part-of-speech tagger*. Refer to both the recipes for details on `train_sents` and `test_sents`.

```
>>> 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%!

How it works...

When a `SequentialBackoffTagger` class is initialized, it creates an internal list of backoff taggers with itself as the first element. If a backoff tagger is given, then the backoff tagger's internal list of taggers is appended. Here's some code to illustrate this:

```
>>> tagger1._taggers == [tagger1]
True
>>> tagger2._taggers == [tagger2, tagger1]
True
```

The `_taggers` list is the internal list of backoff taggers that the `SequentialBackoffTagger` class uses when the `tag()` method is called. It goes through its list of taggers, calling `choose_tag()` on each one. As soon as a tag is found, it stops and returns that tag. This means that if the primary tagger can tag the word, then that's the tag that will be returned. But if it returns `None`, then the next tagger is tried, and so on until a tag is found, or else `None` is returned. Of course, `None` will never be returned if your final backoff tagger is a `DefaultTagger`.

There's more...

While most of the taggers included in NLTK are subclasses of `SequentialBackoffTagger`, not all of them are. There's a few taggers that we'll cover in the later recipes that cannot be used as part of a backoff tagging chain, such as the `BrillTagger` class. However, these taggers generally take another tagger to use as a baseline, and a `SequentialBackoffTagger` class is often a good choice for that baseline.

Saving and loading a trained tagger with pickle

Since training a tagger can take a while, and you generally only need to do the training once, pickling a trained tagger is a useful way to save it for later usage. 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.

See also

In the next recipe, we'll combine more taggers with backoff tagging. Also, see the previous two recipes for details on the `DefaultTagger` and `UnigramTagger` classes.

Training and combining ngram taggers

In addition to `UnigramTagger`, there are two more `NgramTagger` subclasses: `BigramTagger` and `TrigramTagger`. The `BigramTagger` subclass uses the previous tag as part of its context, while the `TrigramTagger` subclass uses the previous two tags. An **ngram** is a subsequence of *n* items, so the `BigramTagger` subclass looks at two items (the previous tagged word and the current word), and the `TrigramTagger` subclass looks at three items.

These two taggers are good at handling words whose part-of-speech tag is context-dependent. Many words have a different part of speech depending on how they are used. For example, we've been talking about taggers that tag words. In this case, *tag* is used as a verb. But the result of tagging is a part-of-speech tag, so *tag* can also be a noun. The idea with the `NgramTagger` subclasses is that by looking at the previous words and part-of-speech tags, we can better guess the part-of-speech tag for the current word. Internally, each tagger maintains a context dictionary (implemented in the `ContextTagger` parent class) that is used to guess that tag based on the context. In the case of `NgramTagger` subclasses, the context is some number of previous tagged words.

Getting ready

Refer to the first two recipes of this chapter for details on constructing `train_sents` and `test_sents`.

How to do it...

By themselves, `BigramTagger` and `TrigramTagger` perform quite poorly. This is partly because they cannot learn context from the first word(s) in a sentence. Since a `UnigramTagger` class doesn't care about the previous context, it is able to have higher baseline accuracy by simply guessing the most common tag for each word.

```
>>> from nltk.tag import BigramTagger, TrigramTagger
>>> bitagger = BigramTagger(train_sents)
>>> bitagger.evaluate(test_sents)
0.11310166199007123
>>> tritagger = TrigramTagger(train_sents)
>>> tritagger.evaluate(test_sents)
0.0688107058061731
```

Where `BigramTagger` and `TrigramTagger` can make a contribution is when we combine them with backoff tagging. This time, instead of creating each tagger individually, we'll create a function that will take `train_sents`, a list of `SequentialBackoffTagger` classes, and an optional final backoff tagger, then train each tagger with the previous tagger as a backoff. Here's the code from `tag_util.py`:

```
def backoff_tagger(train_sents, tagger_classes, backoff=None):
  for cls in tagger_classes:
    backoff = cls(train_sents, backoff=backoff)

  return backoff
```

And to use it, we can do the following:

```
>>> from tag_util import backoff_tagger
>>> backoff = DefaultTagger('NN')
>>> tagger = backoff_tagger(train_sents, [UnigramTagger, BigramTagger,
TrigramTagger], backoff=backoff)
>>> tagger.evaluate(test_sents)
0.8806820634578028
```

So, we've gained almost 1% accuracy by including the `BigramTagger` and `TrigramTagger` subclasses in the backoff chain. For corpora other than `treebank`, the accuracy gain may be more or less significant, depending on the nature of the text.

How it works...

The `backoff_tagger` function creates an instance of each tagger class in the list, giving it `train_sents` and the previous tagger as a backoff. The order of the list of tagger classes is quite important: the first class in the list (`UnigramTagger`) will be trained first and given the initial backoff tagger (the `DefaultTagger`). This tagger will then become the backoff tagger for the next tagger class in the list. The final tagger returned will be an instance of the last tagger class in the list (`TrigramTagger`). 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`. Then, the next backoff will be a `UnigramTagger`, whose backoff is the `DefaultTagger`.

There's more...

The `backoff_tagger` function doesn't just work with `NgramTagger` classes, it can also be used for constructing a chain containing any subclasses of `SequentialBackoffTagger`.

`BigramTagger` and `TrigramTagger`, because they are subclasses of `NgramTagger` and `ContextTagger`, can also take a model and cutoff argument, just like the `UnigramTagger`. But unlike for `UnigramTagger`, the context keys of the model must be two tuples, where the first element is a section of the history and the second element is the current token. For the `BigramTagger`, an appropriate context key looks like `((prevtag,), word)`, and for `TrigramTagger`, it looks like `((prevtag1, prevtag2), word)`.

Quadgram tagger

The `NgramTagger` class can be used by itself to create a tagger that uses more than three ngrams for its context key.

```
>>> from nltk.tag import NgramTagger
>>> quadtagger = NgramTagger(4, train_sents)
>>> quadtagger.evaluate(test_sents)
0.058234405352903085
```

It's even worse than the `TrigramTagger`! Here's an alternative implementation of a `QuadgramTagger` class that we can include in a list to `backoff_tagger`. This code can be found in `taggers.py`.

```
from nltk.tag import NgramTagger

class QuadgramTagger(NgramTagger):
  def __init__(self, *args, **kwargs):
    NgramTagger.__init__(self, 4, *args, **kwargs)
```

This is essentially how `BigramTagger` and `TrigramTagger` are implemented: simple subclasses of `NgramTagger` that pass in the number of ngrams to look at in the `history` argument of the `context()` method.

Now, let's see how it does as part of a backoff chain.

```
>>> 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`. So, the lesson is that too much context can have a negative effect on accuracy.

See also

The previous two recipes cover the `UnigramTagger` and backoff tagging.

Creating a model of likely word tags

As previously mentioned in the *Training a unigram part-of-speech tagger* recipe, using a custom model with a `UnigramTagger` class should only be done if you know exactly what you're doing. In this recipe, we're going to create a model for the most common words, most of which always have the same tag no matter what.

How to do it...

To find the most common words, we can use `nltk.probability.FreqDist` to count word frequencies in the `treebank` corpus. Then, we can create a `ConditionalFreqDist` class for tagged words, where we count the frequency of every tag for every word. Using these counts, we can construct a model of the 200 most frequent words as keys, with the most frequent tag for each word as a value. Here's the model creation function defined in `tag_util.py`.

```
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`. Let's try adding it to our backoff chain.

```
>>> default_tagger = DefaultTagger('NN')
>>> likely_tagger = UnigramTagger(model=model, backoff=default_tagger)
>>> tagger = backoff_tagger(train_sents, [UnigramTagger, BigramTagger,
TrigramTagger], backoff=likely_tagger)
>>> tagger.evaluate(test_sents)
0.8806820634578028
```

The final accuracy is exactly the same as without the `likely_tagger`. This is because the frequency calculations we did to create the model are almost exactly the same as what happens when we train a `UnigramTagger` class.

How it works...

The `word_tag_model()` function takes a list of all words, a list of all tagged words, and the maximum number of words we want to use for our model. We give the list of words to a `FreqDist` class, which counts the frequency of each word. Then, we get the top 200 words from the `FreqDist` class by calling `fd.most_common()`, which obviously returns a list of the most common words and counts. The `FreqDist` class is actually a subclass of `collections.Counter`, which provides the `most_common()` method.

Next, we give the list of tagged words to `ConditionalFreqDist`, which creates a `FreqDist` class of tags for each word, with the word as the condition. Finally, we return a `dict` of the top 200 words mapped to their most likely tag.

In the previous edition of this book, we used the `keys()` method of the `FreqDist` class because in NLTK2, the keys were returned in sorted order, from the most frequent to the least. But in NLTK3, `FreqDist` inherits from `collections.Counter`, and the `keys()` method does not use any predictable ordering.

There's more...

It may seem useless to include this tagger as it does not change the accuracy. But the point of this recipe is to demonstrate how to construct a useful model for a `UnigramTagger` class. Custom model construction is a way to create a manual override of trained taggers that are otherwise black boxes. 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.

See also

The *Training a unigram part-of-speech tagger* recipe has details on the `UnigramTagger` class and a simple custom model example. See the earlier *recipes Combining taggers with backoff tagging* and *Training and combining ngram taggers* for details on backoff tagging.

Tagging with regular expressions

You can use regular expression matching to tag words. For example, you can match numbers with `\d` to assign the tag **CD** (which refers to a Cardinal number). Or you could match on known word patterns, such as the suffix "ing". There's a lot of flexibility here, but be careful of over-specifying since language is naturally inexact, and there are always exceptions to the rule.

Getting ready

For this recipe to make sense, you should be familiar with the regular expression syntax and Python's `re` module.

How to do it...

The `RegexpTagger` class expects a list of two tuples, where the first element in the tuple is a regular expression and the second element is the tag. The patterns shown in the following code can be found in `tag_util.py`:

```
patterns = [
  (r'^\d+$', 'CD'),
  (r'.*ing$', 'VBG'), # gerunds, i.e. wondering
  (r'.*ment$', 'NN'), # i.e. wonderment
  (r'.*ful$', 'JJ') # i.e. wonderful
]
```

Once you've constructed this list of patterns, you can pass it into `RegexpTagger`.

```
>>> 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. For example, it could be positioned just before a `DefaultTagger` class, to tag words that the ngram tagger(s) missed.

How it works...

The `RegexpTagger` class saves the patterns given at initialization, then on each call to `choose_tag()`, it iterates over the patterns and returns the tag for the first expression that matches the current word using `re.match()`. This means that if you have two expressions that could match, the tag of the first one will always be returned, and the second expression won't even be tried.

There's more...

The `RegexpTagger` class can replace the `DefaultTagger` class if you give it a pattern such as `(r'.*', 'NN')`. This pattern should, of course, be last in the list of patterns, otherwise no other patterns will match.

See also

In the next recipe, we'll cover the `AffixTagger` class, which learns how to tag based on prefixes and suffixes of words. See the *Default tagging* recipe for details on the `DefaultTagger` class.

Affix tagging

The `AffixTagger` class is another `ContextTagger` subclass, but this time the context is either the prefix or the suffix of a word. This means the `AffixTagger` class is able to learn tags based on fixed-length substrings of the beginning or ending of a word.

How to do it...

The default arguments for an `AffixTagger` class specify three-character suffixes, and that words must be at least five characters long. If a word is less than five characters, then `None` is returned as the tag.

```
>>> from nltk.tag import AffixTagger
>>> tagger = AffixTagger(train_sents)
>>> tagger.evaluate(test_sents)
0.27558817181092166
```

So, it does ok by itself with the default arguments. Let's try it by specifying three-character prefixes.

```
>>> prefix_tagger = AffixTagger(train_sents, affix_length=3)
>>> prefix_tagger.evaluate(test_sents)
0.23587308439456076
```

To learn on two-character suffixes, the code will look like this:

```
>>> suffix_tagger = AffixTagger(train_sents, affix_length=-2)
>>> suffix_tagger.evaluate(test_sents)
0.31940427368875457
```

How it works...

A positive value for `affix_length` means that the `AffixTagger` class will learn word prefixes, essentially `word[:affix_length]`. If `affix_length` is negative, then suffixes are learned using `word[affix_length:]`.

There's more...

You can combine multiple affix taggers in a backoff chain if you want to learn on multiple character length affixes. 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.

The ordering in the previous block of code is not the best, nor is it the worst. I'll leave it to you to explore the possibilities and discover the best backoff chain of values for `AffixTagger` and `affix_length`.

Working with min_stem_length

The `AffixTagger` class also takes a `min_stem_length` keyword argument, with a default value of `2`. If the word length is less than `min_stem_length` plus the absolute value of `affix_length`, then `None` is returned by the `context()` method. Increasing `min_stem_length` forces the `AffixTagger` class to only learn on longer words, while decreasing `min_stem_length` will allow it to learn on shorter words. Of course, for shorter words, the `affix_length` argument could be equal to or greater than the word length, and `AffixTagger` would essentially be acting like a `UnigramTagger` class.

See also

You can manually specify prefixes and suffixes using regular expressions, as shown in the previous recipe. The *Training a unigram part-of-speech tagger* and *Training and combining ngram taggers* recipes have details on `NgramTagger` subclasses, which are also subclasses of `ContextTagger`.

Training a Brill tagger

The `BrillTagger` class is a transformation-based tagger. It is the first tagger that is not a subclass of `SequentialBackoffTagger`. Instead, the `BrillTagger` class uses a series of rules to correct the results of an initial tagger. These rules are scored based on how many errors they correct minus the number of new errors they produce.

How to do it...

Here's a function from `tag_util.py` that trains a `BrillTagger` class using `BrillTaggerTrainer`. It requires an `initial_tagger` and `train_sents`.

```
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])),
```

```
        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`.

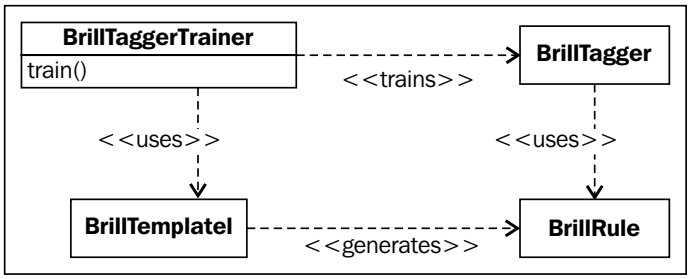
How it works...

The `BrillTaggerTrainer` class takes an `initial_tagger` argument and a list of templates. These templates must implement the `BrillTemplateI` interface, which is found in the `nltk.tbl.template` module. The `brill.Template` class is such an implementation, and is actually imported from `nltk.tbl.template`. The `brill.Pos` and `brill.Word` classes are subclasses of `nltk.tbl.template.Feature`, and they describe what kind of features to use in the template, in this case, one or more part-of-speech tags or words.

The templates specify how to learn transformation rules. For example, `brill.Template(brill.Pos([-1]))` means that a rule can be generated using the previous part-of-speech tag. The `brill.Template(brill.Pos([1]))` statement means that you can look at the next part-of-speech tag to generate a rule. And `brill.Template(brill.Word([-2, -1]))` means you can look at the combination of the previous two words to learn a transformation rule.

The thinking behind a transformation-based tagger is this: given the correct training sentences, the output of the initial tagger, and the templates specifying features, try to generate transformation rules that correct the initial tagger's output to be more in-line with the training sentences. The job of `BrillTaggerTrainer` is to produce these rules, and to do so in a way that increases accuracy. A transformation rule that fixes one problem may cause an error in another condition; thus, every rule must be measured by how many errors it corrects versus how many new errors it introduces.

The workflow looks something like this:



There's more...

You can control the number of rules generated using the `max_rules` keyword argument to the `BrillTaggerTrainer.train()` method. The default value is `200`. You can also control the quality of rules used with the `min_score` keyword argument. The default value is `2`, though `3` can be a good choice as well. The score is a measure of how well a rule corrects errors compared to how many new errors it introduces.

Increasing `max_rules` or `min_score` will greatly increase training time, without necessarily increasing accuracy. Change these values with care.

Tracing

You can watch the `BrillTaggerTrainer` class do its work by passing `trace=True` into the constructor, for example, `trainer = brill.BrillTaggerTrainer(initial_tagger, templates, deterministic=True, trace=True)`. This will give you the following output:

```
TBL train (fast) (seqs: 3000; tokens: 77511; tpls: 18; min score: 2;
min acc: None)
    Finding initial useful rules...
        Found 9869 useful rules.
    Selecting rules...
```

This means it found `77511` rules with a score of at least `min_score`, and then it selects the best rules, keeping no more than `max_rules`.

The default is `trace=False`, which means the trainer will work silently without printing its status.

See also

The *Training and combining ngram taggers* recipe details the construction of the `initial_tagger` argument used earlier, and the *Default tagging* recipe explains the `default_tagger` argument.

Training the TnT tagger

TnT stands for **Trigrams'n'Tags**. It is a statistical tagger based on second order Markov models. The details of this are out of the scope of this book, but you can read more about the original implementation at `http://www.coli.uni-saarland.de/~thorsten/tnt/`.

How to do it...

The `TnT` tagger has a slightly different API than the previous taggers we've encountered. You must explicitly call the `train()` method after you've created it. Here's a basic example.

```
>>> from nltk.tag import tnt
>>> tnt_tagger = tnt.TnT()
>>> tnt_tagger.train(train_sents)
>>> tnt_tagger.evaluate(test_sents)
0.8756313403842003
```

It's quite a good tagger all by itself, only slightly less accurate than the `BrillTagger` class from the previous recipe. But if you do not call `train()` before `evaluate()`, you'll get an accuracy of 0%.

How it works...

The `TnT` tagger maintains a number of internal `FreqDist` and `ConditionalFreqDist` instances based on the training data. These frequency distributions count unigrams, bigrams, and trigrams. Then, during tagging, the frequencies are used to calculate the probabilities of possible tags for each word. So, instead of constructing a backoff chain of `NgramTagger` subclasses, the `TnT` tagger uses all the ngram models together to choose the best tag. It also tries to guess the tags for the whole sentence at once by choosing the most likely model for the entire sentence, based on the probabilities of each possible tag.

Training is fairly quick, but tagging is significantly slower than the other taggers we've covered. This is due to all the floating point math that must be done to calculate the tag probabilities of each word.

There's more...

The `TnT` tagger accepts a few optional keyword arguments. You can pass in a tagger for unknown words as `unk`. If this tagger is already trained, then you must also pass in `Trained=True`. Otherwise, it will call `unk.train(data)` with the same data you pass into the `train()` method. Since none of the previous taggers have a public `train()` method, I recommend always passing `Trained=True` if you also pass an `unk` tagger. Here's an example using a `DefaultTagger` class, which does not require any training.

```
>>> from nltk.tag import DefaultTagger
>>> unk = DefaultTagger('NN')
>>> tnt_tagger = tnt.TnT(unk=unk, Trained=True)
>>> tnt_tagger.train(train_sents)
>>> tnt_tagger.evaluate(test_sents)
0.892467083962875
```

So, we got an almost 2% increase in accuracy! You must use a tagger that can tag a single word without having seen that word before. This is because the unknown tagger's `tag()` method is only called with a single word sentence. Other good candidates for an unknown tagger are `RegexpTagger` and `AffixTagger`. Passing in a `UnigramTagger` class that's been trained on the same data is pretty much useless, as it will have seen the exact same words and, therefore, have the same unknown word blind spots.

Controlling the beam search

Another parameter you can modify for `TnT` is `N`, which controls the number of possible solutions the tagger maintains while trying to guess the tags for a sentence. `N` defaults to `1000`. Increasing it will greatly increase the amount of memory used during tagging, without necessarily increasing the accuracy. Decreasing `N` will decrease memory usage, but could also decrease accuracy. Here's what happens when the value is changed to `N=100`.

```
>>> 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. However, don't assume that accuracy will not change if you decrease `N`; experiment with your own data to be sure.

Significance of capitalization

You can pass `C=True` to the `TnT` constructor if you want capitalization of words to be significant. The default is `C=False`, which means all words are lowercase. The documentation on `C` says that treating capitalization as significant probably will not increase accuracy. In my own testing, there was a very slight (< 0.01%) increase in accuracy with `C=True`, probably because case-sensitivity can help identify proper nouns.

See also

We have covered the `DefaultTagger` class in the *Default tagging* recipe, backoff tagging in the *Combining taggers with backoff tagging* recipe, `NgramTagger` subclasses in the *Training a unigram part-of-speech tagger* and *Training and combining ngram taggers* recipes, `RegexpTagger` in the *Tagging with regular expressions recipe*, and the `AffixTagger` class in the *Affix tagging recipe*.

Using WordNet for tagging

If you remember from the *Looking up Synsets for a word in WordNet* recipe in *Chapter 1*, *Tokenizing Text and WordNet Basics*, WordNet Synsets specify a part-of-speech tag. It's a very restricted set of possible tags, and many words have multiple Synsets with different part-of-speech tags, but this information can be useful for tagging unknown words. WordNet is essentially a giant dictionary, and it's likely to contain many words that are not in your training data.

Getting ready

First, we need to decide how to map WordNet part-of-speech tags to the Penn Treebank part-of-speech tags we've been using. The following is a table mapping one to the other. See the *Looking up Synsets for a word in WordNet* recipe in *Chapter 1*, *Tokenizing Text and WordNet Basics*, for more details. The `s`, which was not shown before, is just another kind of adjective, at least for tagging purposes.

WordNet tag	Treebank tag
n	NN
a	JJ
s	JJ
r	RB
v	VB

How to do it...

Now we can create a class that will look up words in WordNet, and then choose the most common tag from the Synsets it finds. The `WordNetTagger` class defined in the following code can be found in `taggers.py`:

```
from nltk.tag import SequentialBackoffTagger
from nltk.corpus import wordnet
from nltk.probability import FreqDist

class WordNetTagger(SequentialBackoffTagger):
  '''
  >>> wt = WordNetTagger()
  >>> wt.tag(['food', 'is', 'great'])
  [('food', 'NN'), ('is', 'VB'), ('great', 'JJ')]
  '''
  def __init__(self, *args, **kwargs):
    SequentialBackoffTagger.__init__(self, *args, **kwargs)

    self.wordnet_tag_map = {
      'n': 'NN',
      's': 'JJ',
      'a': 'JJ',
      'r': 'RB',
      'v': 'VB'
    }

  def choose_tag(self, tokens, index, history):
    word = tokens[index]
    fd = FreqDist()

    for synset in wordnet.synsets(word):
      fd[synset.pos()] += 1

    return self.wordnet_tag_map.get(fd.max())
```

Another way the `FreqDist` API has changed between NLTK2 and NLTK3 is that the `inc()` method has been removed. Instead, you must use `fd[key] += 1`. Since `FreqDist` inherits from `collections.Counter`, it's ok if `fd[key]` doesn't exist the first time you increment.

How it works...

The `WordNetTagger` class simply counts the number of each part-of-speech tag found in the Synsets for a word. The most common tag is then mapped to a `treebank` tag using internal mapping. Here's some sample usage code:

```
>>> from taggers import WordNetTagger
>>> wn_tagger = WordNetTagger()
>>> wn_tagger.evaluate(train_sents)
0.17914876598160262
```

So, it's not too accurate, but that's to be expected. We only have enough information to produce four different kinds of tags, while there are 36 possible tags in `treebank`. There are many words that can have different part-of-speech tags depending on their context. But if we put the `WordNetTagger` class at the end of an `NgramTagger` backoff chain, then we can improve accuracy over the `DefaultTagger` class.

```
>>> from tag_util import backoff_tagger
>>> from nltk.tag import UnigramTagger, BigramTagger, TrigramTagger
>>> tagger = backoff_tagger(train_sents, [UnigramTagger, BigramTagger,
TrigramTagger], backoff=wn_tagger)
>>> tagger.evaluate(test_sents)
0.8848262464925534
```

See also

The *Looking up Synsets for a word in WordNet* recipe in *Chapter 1*, *Tokenizing Text and WordNet Basics*, details how to use the `wordnet` corpus and what kinds of part-of-speech tags it knows about. And in the *Combining taggers with backoff tagging* and *Training and combining ngram taggers* recipes, we went over backoff tagging with ngram taggers.

Tagging proper names

Using the included `names` corpus, we can create a simple tagger for tagging names as proper nouns.

How to do it...

The `NamesTagger` class is a subclass of `SequentialBackoffTagger` as it's probably only useful near the end of a backoff chain. At initialization, we create a set of all names in the `names` corpus, lower-casing each name to make lookup easier. Then, we implement the `choose_tag()` method, which simply checks whether the current word is in the `names_set` list. If it is, we return the `NNP` tag (which is the tag for proper nouns). If it isn't, we return `None`, so the next tagger in the chain can tag the word. The following code can be found in `taggers.py`:

```
from nltk.tag import SequentialBackoffTagger
from nltk.corpus import names

class NamesTagger(SequentialBackoffTagger):
  def __init__(self, *args, **kwargs):
    SequentialBackoffTagger.__init__(self, *args, **kwargs)
    self.name_set = set([n.lower() for n in names.words()])

    def choose_tag(self, tokens, index, history):
      word = tokens[index]

      if word.lower() in self.name_set:
        return 'NNP'
      else:
        return None
```

How it works...

The `NamesTagger` class should be pretty self-explanatory. The usage is also simple.

```
>>> from taggers import NamesTagger
>>> nt = NamesTagger()
>>> nt.tag(['Jacob'])
[('Jacob', 'NNP')]
```

It's probably best to use the `NamesTagger` class right before a `DefaultTagger` class, so it's at the end of a backoff chain. But it could probably go anywhere in the chain since it's unlikely to mis-tag a word.

See also

The *Combining taggers with backoff tagging* recipe goes over the details of using the `SequentialBackoffTagger` subclasses.

Classifier-based tagging

The `ClassifierBasedPOSTagger` class uses classification to do part-of-speech tagging. Features are extracted from words, and then passed to an internal classifier. The classifier classifies the features and returns a label, in this case, a part-of-speech tag. Classification will be covered in detail in *Chapter 7*, *Text Classification*.

The `ClassifierBasedPOSTagger` class is a subclass of `ClassifierBasedTagger` that implements a feature detector that combines many of the techniques of the previous taggers into a single feature set. The feature detector finds multiple length suffixes, does some regular expression matching, and looks at the unigram, bigram, and trigram history to produce a fairly complete set of features for each word. The feature sets it produces are used to train the internal classifier, and are used for classifying words into part-of-speech tags.

How to do it...

The basic usage of the `ClassifierBasedPOSTagger` class is much like any other `SequentialBackoffTaggger`. You pass in training sentences, it trains an internal classifier, and you get a very accurate tagger.

```
>>> from nltk.tag.sequential import ClassifierBasedPOSTagger
>>> tagger = ClassifierBasedPOSTagger(train=train_sents)
>>> tagger.evaluate(test_sents)
0.9309734513274336
```

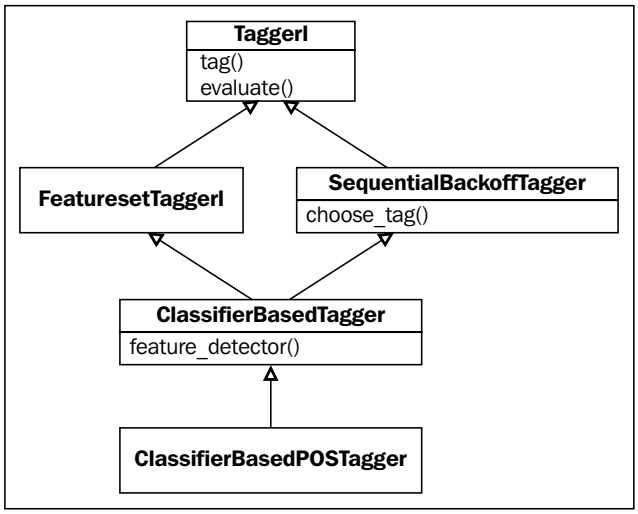
Notice a slight modification to initialization: `train_sents` must be passed in as the `train` keyword argument.

How it works...

The `ClassifierBasedPOSTagger` class inherits from `ClassifierBasedTagger` and only implements a `feature_detector()` method. All the training and tagging is done in `ClassifierBasedTagger`. It defaults to training a `NaiveBayesClassifier` class with the given training data. Once this classifier is trained, it is used to classify word features produced by the `feature_detector()` method.

The `ClassifierBasedTagger` class is often the most accurate tagger, but it's also one of the slowest taggers. If speed is an issue, you should stick with a `BrillTagger` class based on a backoff chain of `NgramTagger` subclasses and other simple taggers.

The `ClassifierBasedTagger` class also inherits from `FeatursetTaggerI` (which is just an empty class), creating an inheritance tree that looks like this:



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`. If you have SciPy and NumPy installed, training will be faster than normal, but still slower than `NaiveBayesClassifier`.

Detecting features with a custom feature detector

If you want to do your own feature detection, there are two ways to do it:

1. Subclass `ClassifierBasedTagger` and implement a `feature_detector()` method.
2. Pass a function as the `feature_detector` keyword argument into `ClassifierBasedTagger` at initialization.

Either way, you need a feature detection method that can take the same arguments as `choose_tag(): tokens, index, history`. But instead of returning a tag, you return a `dict` of key-value features, where the key is the feature name and the value is the feature value. A very simple example would be a unigram feature detector (found in `tag_util.py`).

```
def unigram_feature_detector(tokens, index, history):
  return {'word': tokens[index]}
```

Then, using the second method, you'd pass this into `ClassifierBasedTagger` as `feature_detector`.

```
>>> from nltk.tag.sequential import ClassifierBasedTagger
>>> from tag_util import unigram_feature_detector
>>> tagger = ClassifierBasedTagger(train=train_sents, feature_
detector=unigram_feature_detector)
>>> tagger.evaluate(test_sents)
0.8733865745737104
```

Setting a cutoff probability

Because a classifier will always return the best result it can, passing in a backoff tagger is useless unless you also pass in a `cutoff_prob` argument to specify the probability threshold for classification. Then, if the probability of the chosen tag is less than `cutoff_prob`, the backoff tagger will be used. 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%.

Using a pre-trained classifier

If you want to use a classifier that's already been trained, then you can pass that into `ClassifierBasedTagger` or `ClassifierBasedPOSTagger` as the `classifier`. In this case, the `classifier_builder` argument is ignored and no training takes place. However, you must ensure that the classifier has been trained on and can classify feature sets produced by whatever `feature_detector()` method you use.

See also

Chapter 7, *Text Classification*, will cover classification in depth.

Training a tagger with NLTK-Trainer

As you can tell from all the previous recipes in this chapter, there are many different ways to train taggers, and it's impossible to know which methods and parameters will work best without doing training experiments. But training experiments can be tedious, since they often involve many small code changes (and lots of cut and paste) before you converge on an optimal tagger. In an effort to simplify the process, and make my own work easier, I created a project called `NLTK-Trainer`.

NLTK-Trainer is a collection of scripts that give you the ability to run training experiments without writing a single line of code. The project is available on GitHub at `https://github.com/japerk/nltk-trainer` and has documentation at `http://nltk-trainer.readthedocs.org/`. This recipe will introduce the tagging related scripts, and will show you how to combine many of the previous recipes into a single training command. For download and installation instructions, please go to `http://nltk-trainer.readthedocs.org/`.

How to do it...

The simplest way to run `train_tagger.py` is with the name of an `NLTK` corpus. 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>
```

```
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.
pickle
```

That's all it takes to train a tagger on `treebank` and have it dumped to a `pickle` file at `~/nltk_data/taggers/treebank_aubt.pickle`. "Wow, and it's over 99% accurate!" I hear you saying. But look closely at the second line of output: `3914 tagged sents, training on 3914`. This means that the tagger was trained on the entire `treebank` corpus, and then tested against those same training sentences. This is a very misleading way to evaluate any trained model. In the previous recipes, we used the first 3000 sentences for training and the remaining 914 sentences for testing, or about a 75% split. 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...

The `train_tagger.py` script roughly performers the following steps:

1. Construct training and testing sentences from corpus arguments.
2. Build tagger training function from tagger arguments.
3. Train a tagger on the training sentences using the training function.
4. Evaluate and/or save the tagger.

The first argument to the script is `corpus`. This could be the name of an `NLTK` corpus that can be found in the `nltk.corpus` module, such as `treebank` or `brown`. It could also be the path to a custom corpus directory. If it's a path to a custom corpus, then you'll also need to use the `--reader` argument to specify the corpus reader class, such as `nltk.corpus.reader.tagged.TaggedCorpusReader`.

The default training algorithm is `aubt`, which is shorthand for a sequential backoff tagger composed of `AffixTagger + UnigramTagger + BigramTagger + TrigramTagger`. It's probably easiest to understand by replicating many of the previous recipes using `train_tagger.py`. Let's start with a default tagger.

```
$ python train_tagger.py treebank --no-pickle --default NN --sequential
''
loading treebank
3914 tagged sents, training on 3914
evaluating DefaultTagger
accuracy: 0.130776
```

Using `--default NN` lets us assign a default tag of `NN`, while `--sequential ''` disables the default `aubt` sequential backoff algorithm. The `--fraction` argument is omitted in this case because there's not actually any training happening.

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
```

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. This means that when you run `train_tagger.py`, your output and accuracy may vary. To get consistent accuracy values, run `train_tagger.py` like this:

```
$ PYTHONHASHSEED=0 python train_tagger.py treebank …
```

The default training algorithm is `--sequential aubt`, and the default affix is `-3`. But you can modify this with one or more `-a` arguments. 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
```

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. The template bounds the default to `(1, 1)` but can be customized with the `--template_bounds` argument.

```
$ python train_tagger.py treebank --no-pickle --default NN --fraction
0.75 --brill
loading treebank
3914 tagged sents, training on 2936
training AffixTagger with affix -3 and backoff <DefaultTagger: tag=NN>
training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff
<AffixTagger: size=2143>
training <class 'nltk.tag.sequential.BigramTagger'> tagger with backoff
<UnigramTagger: size=4179>
training <class 'nltk.tag.sequential.TrigramTagger'> tagger with backoff
<BigramTagger: size=1824>
Training Brill tagger on 2936 sentences...
Finding initial useful rules...
    Found 1304 useful rules.
Selecting rules...
evaluating BrillTagger
accuracy: 0.909138
```

Finally, you can train a classifier-based tagger with the `--classifier` argument, which specifies the name of a classifier. Be sure to also pass in `--sequential ''` because, as we learned previously, training a sequential backoff tagger in addition to a classifier-based tagger is useless. The `--default` argument is also useless, because the classifier will always guess something.

```
$ python train_tagger.py treebank --no-pickle --fraction 0.75
--sequential '' --classifier NaiveBayes
loading treebank
3914 tagged sents, training on 2936
training ['NaiveBayes'] ClassifierBasedPOSTagger
Constructing training corpus for classifier.
Training classifier (75814 instances)
training NaiveBayes classifier
evaluating ClassifierBasedPOSTagger
accuracy: 0.928646
```

There are a few other classifier algorithms available besides `NaiveBayes`, and even more if you have NumPy and SciPy installed.

While classifier-based taggers tend to be more accurate, they are also slower to train, and much slower at tagging. If speed is important to you, I recommend sticking with sequential taggers.

There's more...

The `train_tagger.py` script supports many other arguments not shown here, all of which you can see by running the script with `--help`. A few additional arguments are presented next, followed by an introduction to two other tagging-related scripts available in `NLTK-Trainer`.

Saving a pickled tagger

Without the `--no-pickle` argument, `train_tagger.py` will save a pickled tagger at `~/nltk_data/taggers/NAME.pickle`, where `NAME` is a combination of the corpus name and training algorithm. You can specify a custom filename for your tagger using the `--filename` argument like this:

```
$ python train_tagger.py treebank --filename path/to/tagger.pickle
```

Training on a custom corpus

If you have a custom corpus that you want to use for training a tagger, you can do that by passing in the path to the corpus and the classname of a corpus reader in the `--reader` argument. The corpus path can either be absolute or relative to a `nltk_data` directory. The corpus reader class must provide a `tagged_sents()` method. Here's an example using a relative path to the `treebank` tagged corpus:

```
$ python train_tagger.py corpora/treebank/tagged --reader nltk.corpus.
reader.ChunkedCorpusReader --no-pickle --fraction 0.75
loading corpora/treebank/tagged
51002 tagged sents, training on 38252
training AffixTagger with affix -3 and backoff <DefaultTagger: tag=-
None->
training <class 'nltk.tag.sequential.UnigramTagger'> tagger with backoff
<AffixTagger: size=2092>
training <class 'nltk.tag.sequential.BigramTagger'> tagger with backoff
<UnigramTagger: size=4121>
training <class 'nltk.tag.sequential.TrigramTagger'> tagger with backoff
<BigramTagger: size=1627>
evaluating TrigramTagger
accuracy: 0.883175
```

Training with universal tags

You can train a tagger with the universal tagset using the `--tagset` argument as follows:

```
$ python train_tagger.py treebank --no-pickle --fraction 0.75 --tagset
universal
loading treebank
using universal tagset
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=2889>
training <class 'nltk.tag.sequential.TrigramTagger'> tagger with backoff
<BigramTagger: size=1014>
evaluating TrigramTagger
accuracy: 0.934800
```

Because the universal tagset has fewer tags, these taggers tend to be more accurate; this will only work on a corpus that has universal tagset mappings. The universal tagset was covered in the *Creating a part-of-speech tagged word corpus* recipe in *Chapter 3*, *Creating Custom Corpora*.

Analyzing a tagger against a tagged corpus

Every previous example in this chapter has been about training and evaluating a tagger on a single corpus. But how do you know how well that tagger will perform on a different corpus? The `analyze_tagger_coverage.py` script gives you a simple way to test the performance of a tagger against another tagged corpus. Here's how to test NLTK's built-in tagger against the `treebank` corpus:

```
$ python analyze_tagger_coverage.py treebank --metrics
```

The output has been omitted for brevity, but I encourage you to run it yourself to see the results. It's especially useful for evaluating a tagger's performance on a corpus that it was not trained on, such as `conll2000` or `brown`.

If you only provide a corpus argument, this script will use NLTK's built-in tagger. To evaluate your own tagger, you can use the `--tagger` argument, which takes a path to a pickled tagger. The path can be absolute or relative to a `nltk_data` directory. For example:

```
$ python analyze_tagger_coverage.py treebank --metrics --tagger path/to/
tagger.pickle
```

You can also use a custom corpus just like we did earlier with `train_tagger.py`, but if your corpus is not tagged, then you must omit the `--metrics` argument. In that case, you will only get tag counts, with no notion of accuracy, because there are no tags to compare to.

Analyzing a tagged corpus

Finally, there is a script called `analyze_tagged_corpus.py`, which, as the name implies, will read in a tagged corpus and print out stats about the number of words and tags. You can run it as follows:

```
$ python analyze_tagged_corpus.py treebank
```

The results are available in *Appendix*, *Penn Treebank Part-of-speech Tags*. As with the other commands, you can pass in a custom corpus path and reader to analyze your own tagged corpus.

See also

The previous recipes in this chapter cover the details of the classes and methods that power the functionality of `train_tagger.py`. The *Training a chunker with NLTK-Trainer* recipe at the end of *Chapter 5*, *Extracting Chunks*, will introduce NLTK-Trainer's chunking-related scripts, and classification-related scripts will be covered in the *Training a classifier with NLTK-Trainer* recipe at the end of *Chapter 7*, *Text Classification*.