NLTK Tutorial: Tagging

Edward Loper

Table of Contents

1. Introduction

When processing a text, it is often useful to associate auxiliary information with each token. For example, we might want to label each token with its part of speech; or we might want to disambiguate homonyms by associating them with "word sense" labels. This kind of auxiliary information is typically used in later stages of text processing. For example, part of speech labels could be used to derive the internal structure of a sentence; and "word sense" labels could be used to allow a question-answering system to distinguish homonyms.

The process of associating labels with each token in a text is called *tagging*, and the labels are called *tags*. The collection of tags used for a particular task is known as a *tag set*.

Part-of-speech tagging is the most common example of tagging, and it is the example we will examine in this tutorial. But you should keep in mind that most of the techniques we discuss here can also be applied to many other tagging problems.

2. Part-of-Speech Tagging

Part-of-speech tags divide words into categories, based on how they can be combined to form sentences. For example, articles can combine with nouns, but not verbs. Part-of-speech tags also give information about the semantic content of a word. For example, nouns typically express "things," and prepositions express relationships between "things."

Most part-of-speech tag sets make use of the same basic categories, such as "noun," "verb," "adjective," and "preposition." However, tag sets differ both in how finely they divide words into categories; and in how define their categories. For example, "is" might be tagged as a verb in one tag set; but as a form of "to be" in another tag set. This variation in tag sets is reasonable, since part-of-speech tags are used in different ways for different tasks.

In this tutorial, we will use the tag set listed in Table 1. This tag set is a simplification of the commonly used *Brown Corpus tag set*. The complete Brown Corpus tag set has 87 basic tags. For more information on tag sets, see *Foundations of Statistical Natural Language Processing* (Manning & Schutze), pp. 139-145.

Table 1. Tag Set

AT	Article
NN	Noun
VB	Verb
JJ	Adjective
IN	Preposition
CD	Number
END	Sentence-ending punctuation

3. The nltk.tagger Module

The `nltk.tagger` module defines the classes and interfaces used by NLTK to perform tagging.

3.1. TaggedType

NLTK defines a simple class, `TaggedType`, for representing the text type of a tagged token. A `TaggedType` consists of a *base type* and a *tag*. Typically, the base type and the tag will both be strings. For example, the tagged type for the noun "dog" would have the base type `'dog'` and the tag `'NN'`. A tagged type with base type *b* and tag *t* is written *b*`/`*t*. Tagged types are created with the `TaggedType` constructor:

```
>>> ttype1 = TaggedType('dog', 'NN')
'dog'/'NN'
>>> ttype2 = TaggedType('runs', 'VB')
'runs'/'VB'
```

A `TaggedType`'s base type is accessed via the `base` member function; and its tag is accessed via the `tag` member function:

```
>>> ttype1.base()
'dog'
>>> ttype2.tag()
'VB'
```

To construct a tagged token, simply use the `Token` constructor with a `TaggedType`:

```
>>> ttype = TaggedType('dog', 'NN')
'dog'/'NN'
>>> ttoken = Token(ttype, Location(5))
'dog'/'NN'@[5]
```

3.2. Reading Tagged Corpora

Several large corpora (such as the Brown Corpus and portions of the Wall Street Journal) have been manually tagged with part-of-speech tags. These corpora are primarily useful for testing taggers and for training statistical taggers. However, before we can use these corpora, we must read them from files and tokenize them.

Tagged texts are usually stored in files as a sequences of whitespace-separated tokens, where each token is of the form *base*`/`*tag*. Figure 1 shows an example of some tagged text, taken from the Brown corpus.

```
The/at grand/jj jury/nn commented/vbd on/in a/at number/nn of/in
other/ap topics/nns ,/, among/in them/ppo the/at Atlanta/np and/cc
Fulton/np-tl County/nn-tl purchasing/vbg departments/nns which/wdt
it/pps said/vbd “/“ are/ber well/ql operated/vbn and/cc follow/vb
generally/rb accepted/vbn practices/nns which/wdt inure/vb to/in
the/at best/jjt interest/nn of/in both/abx governments/nns ”/” ./.
```

Figure 1. An Example of Tagged Text (excerpted from the Brown Corpus)

To tokenize tagged texts of this form, the `nltk.tagger` module defines the `TaggedTokenizer` class:

```
>>> tagged_text_str = open('corpus.txt').read()
'John/NN saw/VB the/AT book/NN on/IN the/AT
 table/NN ./END  He/NN sighed/VB ./END'
>>> tokens = TaggedTokenizer().tokenize(tagged_text_str)
['John'/'NN'@[0w], 'saw'/'VB'@[1w], 'the'/'AT'@[2w],
 'book'/'NN'@[3w], 'on'/'IN'@[4w], 'the'/'AT'@[5w],
 'table'/'NN'@[6w], '.'/'END'@[7w], 'He'/'NN'@[8w],
 'sighed'/'VB'@[9w], '.'/'END'@[10w]]
```

If `TaggedTokenizer` encounters a word without a tag, it will assign it the default tag `None`.

3.3. The TaggerI Interface

The `nltk.tagger` module defines `TaggerI`, a general interface for tagging texts. This interface is used by all taggers. It defines a single method, `tag`, which assigns a tag to each token in a list, and returns the resulting list of tagged tokens.

```
>>> tokens = WSTokenizer().tokenize(text_str)
['John'@[0w], 'saw'@[1w], 'the'@[2w], 'book'@[3w],
 'on'@[4w], 'the'@[5w], 'table'@[6w], '.'@[7w],
 'He'@[8w], 'sighed'@[9w], '.'@[10w]]
>>> my_tagger.tag(tokens)
['John'/'NN'@[0w], 'saw'/'VB'@[1w], 'the'/'AT'@[2w],
 'book'/'NN'@[3w], 'on'/'IN'@[4w], 'the'/'AT'@[5w],
 'table'/'NN'@[6w], '.'/'END'@[7w], 'He'/'NN'@[8w],
 'sighed'/'VB'@[9w], '.'/'END'@[10w]]
```

4. Taggers

The `nltk.tagger` module currently defines four taggers; this list will likely grow in the future. This section describes the taggers currently implemented by `nltk.tagger`, and how they are used.

4.1. A Default Tagger

The simplest tagger defined by `nltk.tagger` is `NN_CD_Tagger`. This tagger assigns a tag to each token on the basis of its type. If its type appears to be a number, it assigns the type "CD." Otherwise, it assigns the type "NN."

```
>>> tokens = WSTokenizer().tokenize(text_str)
['John'@[0w], 'saw'@[1w], '3'@[2w],
 'polar'@[3w], 'bears'@[4w], '.'@[5w]]
>>> my_tagger.tag(tokens)
['John'/'NN'@[0w], 'saw'/'NN'@[1w], '3'/'CD'@[2w],
 'polar'/'NN'@[3w], 'bears'/'NN'@[4w], '.'/'NN'@[5w]]
```

This is a simple algorithm, but it yields quite poor performance when used by itself. On a typical corpus, it will tag only 20%-30% of the tokens correctly. However, it is a very reasonable tagger to use as a default, if a more advanced tagger fails to determine a token's tag. When used in conjunction with other taggers, `NN_CD_Tagger` can significantly improve performance.

4.2. Unigram Tagging

The `UnigramTagger` class implements a simple statistical tagging algorithm: for each token, it assigns the tag that is most likely for that token's type. For example, it will assign the tag "JJ" to any occurrence of the word "frequent," since "frequent" is used as an adjective (e.g. "a frequent word") more often than it is used as a verb (e.g. "I frequent this cafe").

Before a `UnigramTagger` can be used to tag data, it must be trained on a *training corpus*. It uses this corpus to determine which tags are most common for each word. `UnigramTaggers` are trained using the `train` method, which takes a tagged corpus:

```
# 'train.txt' is a tagged training corpus
>>> tagged_text_str = open('train.txt').read()
>>> train_toks = TaggedTokenizer().tokenize(tagged_text_str)
>>> tagger = UnigramTagger()
>>> tagger.train(train_toks)
```

Once a `UnigramTagger` has been trained, the `tag` can be used to tag untagged corpera:

```
>>> tokens = WSTokenizer().tokenize(text_str)
>>> tagger.tag(tokens)
['John'/'NN'@[0w], 'saw'/'VB'@[1w], 'the'/'AT'@[2w],
 'book'/'NN'@[3w], 'on'/'IN'@[4w], 'the'/'AT'@[5w], ...]
```

`UnigramTagger` will assign the default tag `None` to any token whose type was not encountered in the training data.

Note that, like almost all statistical taggers, the performance of `UnigramTagger` 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 than the texts we wish to tag.

4.3. Nth Order Tagging

The `NthOrderTagger` class implements a more advanced statistical tagging algorithm. In addition to considering the token's type, it also considers the part-of-speech tags of the n preceding tokens.

To decide which tag to assign to a token, `NthOrderTagger` first constructs a *context* for the token. This context consists of the token's type, along with the part-of-speech tags of the n preceding tags. It then picks the tag which is most likely for that context. Note that a 0th order tagger is equivalent to a unigram tagger, since the context used to tag a token is just its type. 1st order taggers are sometimes called *bigram taggers*, and 2nd order taggers are called *trigram taggers*.

`NthOrderTagger` uses a tagged training corpus to determine which part-of-speech tag is most likely for each context:

```
>>> train_toks = TaggedTokenizer().tokenize(tagged_text_str)
>>> tagger = NthOrderTagger(3)         # 3rd order tagger
>>> tagger.train(train_toks)
```

Once an `NthOrderTagger` has been trained, it can be used to tag untagged corpora:

```
>>> tokens = WSTokenizer().tokenize(text_str)
>>> tagger.tag(tokens)
['John'/'NN'@[0w], 'saw'/'VB'@[1w], 'the'/'AT'@[2w],
 'book'/'NN'@[3w], 'on'/'IN'@[4w], 'the'/'AT'@[5w], ...]
```

`NthOrderTagger` will assign the default tag `None` to any token whose context was not encountered in the training data.

Note that as n gets larger, the specificity of the contexts increases; and with it, the chance that the data we wish to tag will contain contexts that were not present in the training data. Thus, there is a trade-off between the accuracy and the coverage of our results. This is a common type of trade-off in natural language processing. It is closely related to the *precision/recall trade-off* that we'll encounter later when we discuss information retrieval.

4.4. Combining Taggers

One way to address the trade-off between accuracy and coverage is to use the more accurate algorithms when we can, but to fall back on algorithms with wider coverage when necessary. For example, we could combine the results of a 1st order tagger, a 0th order tagger, and an `NN_CD_Tagger`, as follows:

1. Try tagging the token with the 1st order tagger.
2. If the 1st order tagger is unable to find a tag for the token, try finding a tag with the 0th order tagger.
3. If the 0th order tagger is also unable to find a tag, use the `NN_CD_Tagger` to find a tag.

NLTK defines the `BackoffTagger` class for combining taggers in this way. A `BackoffTagger` is constructed from an ordered list of one or more *subtaggers*. For each token in the input, the `BackoffTagger` uses the result of the first tagger in the list that successfully found a tag. Taggers indicate that they are unable to tag a token by assigning it the special tag `None`. We can use a `BackoffTagger` to implement the strategy proposed above:

```
>>> train_toks = TaggedTokenizer().tokenize(tagged_text_str)

# Construct the taggers
>>> tagger1 = NthOrderTagger(1)         # 1st order tagger
>>> tagger2 = UnigramTagger()           # 0th order tagger
>>> tagger3 = NN_CD_Tagger()

# Train the taggers
>>> tagger1.train(train_toks)
```

```
>>> tagger2.train(train_toks)

# Combine the taggers
>>> tagger = BackoffTagger([tagger1, tagger2, tagger3])
```

Note that the order in which the taggers are given to `BackoffTagger` is important: the taggers should be listed in the order that they should be tried. This typically means that more specific taggers should be listed before less specific taggers.

Having defined a combined tagger, we can use it to tag new corpora:

```
>>> tokens = TaggedTokenizer().tokenize(tagged_text_str)
>>> tagger.tag(tokens)
['John'/'NN'@[0w], 'saw'/'VB'@[1w], 'the'/'AT'@[2w],
 'book'/'NN'@[3w], 'on'/'IN'@[4w], 'the'/'AT'@[5w], ...]
```

5. Tagging: A Closer Look

In the next five sections, we will discuss how each of the taggers introduced in the previous section are implemented. This discussion serves several purposes:

- It demonstrates how to write classes implementing the interfaces defined by NLTK.
- It provides you with a better understanding of the algorithms and data structures underlying each approach to tagging.
- It gives you a chance to see some of the code used to implement NLTK. We have tried hard to ensure that the implementation of every class in NLTK is easy to understand.

Before you read this section, you may wish to read the tutorial " Writing Classes For NLTK", which describes how to create classes that interface with the toolkit.

6. Sequential Taggers

The four taggers discussed in this tutorial are implemented as sequential taggers. A *sequential tagger* is a tagger that:

1. Assigns tags to one token at a time, starting with the first token of the text, and proceeding in sequential order.
2. Decides which tag to assign a token on the basis of that token, the tokens that preceed it, and the predicted tags for the tokens that preceed it.

To capture this commonality, we define a common base class, `SequentialTagger`. This base class defines `tag` using a new method, `tag_next`, which returns the appropriate tag for the next token. However, `SequentialTagger` does not implement this new method itself. Instead, each tagger subclass provides its own implementation.

In addition to capturing the commonality between the four taggers, the `SequentialTagger` class has another advantage: it will allow us to define `BackoffTagger` in such a way that each subtagger can use the predictions made by

the other taggers as context for deciding which tags to assign. See BackoffTagger for more details.

6.1. SequentialTagger.tag_next

The `tag_next` method decides which tag to assign a token, given the list of tagged tokens that preceeds it. It takes two arguments: a list of tagged tokens preceeding the token to be tagged, and the token to be tagged; and it returns the appropriate tag for that token.

6.2. SequentialTagger.tag

The implementation of the `tag` method is relatively streight forward. It simply loops through the untagged text, calling `tag_next` for each token. It uses the result of each call to `tag_next` to create a tagged version of that token, and collects these together to form the tagged text.

```
    def tag(self, text):
        tagged_text = []

        for token in text:
            tag = self.next_tag(tagged_text, token)
            tagged_token = Token(TaggedType(token.type(), tag), token.loc())
            tagged_text.append(tagged_token)

        return tagged_text
```

6.3. The SequentialTagger Implementation

The complete listing for `SequentialTagger` is:

```
class SequentialTagger(TaggerI):

    def next_tag(self, tagged_tokens, next_token):
        assert 0, "next_tag not defined by SequentialTagger subclass"

    def tag(self, text):
        tagged_text = []

        # Tag each token, in sequential order.
        for token in text:
            # Get the tag for the next token.
            tag = self.next_tag(tagged_text, token)

            # Use tag to build a tagged token, and add it to tagged_text.
            tagged_token = Token(TaggedType(token.type(), tag), token.loc())
            tagged_text.append(tagged_token)

        return tagged_text
```

Figure 2. The SequentialTagger Implementation

Note that SequentialTagger requires that subclasses define the `_tag_next` method; otherwise, the `assert` statement will raise an exception when the user tries to tag a text.

6.4. Subclasses

The next four sections show how the `SequentialTagger` base class can be used to define `NN_CD_Tagger`, `UnigramTagger`, `NthOrderTagger`, and `BackoffTagger`.

7. NN_CD_Tagger

`NN_CD_Tagger` assigns the tag `"CD"` to any token whose type appears to be a number; and `"NN"` to any other token. It uses a simple regular expression to test whether a token's type is a number:

```
r'^[0-9]+(.[0-9]+)?$'
```

This regular expression matches one or more digits, followed by an optional period and one or more digits (e.g., `"12"` or `"732.42"`). Note the use of `"^"` (which matches the beginning of a string) and `"$"` (which matches the end of a string) to ensure that the regular expression will only match complete token types.

Since `NN_CD_Tagger` is a subclass of `SequentialTagger`, it just needs to define the `next_tag` method. In the case of `NN_CD_Tagger`, the `next_tag` method is quite simple:

```
def next_tag(self, tagged_tokens, next_token):
    if re.match(r'^[0-9]+(.[0-9]+)?$', next_token.type()):
        return 'CD'
    else:
        return 'NN'
```

Since `NN_CD_Taggers` are stateless, and have no customization parameters, the `NN_CD_Tagger constructor` is empty:

```
def __init__(self): pass
```

The complete listing for the `NN_CD_Tagger` class is:

```
class NN_CD_Tagger(SequentialTagger):

    def __init__(self): pass

    def next_tag(self, tagged_tokens, next_token):
        # Assign the 'CD' tag for numbers; and 'NN' for anything else.
        if re.match(r'^[0-9]+(.[0-9]+)?$', next_token.type()):
            return 'CD'
        else:
            return 'NN'
```

Figure 3. The NN_CD_Tagger Implementation

Note that `NN_CD_Tagger` does *not* define `tag`. When the `tag` method is called, the definition given by `SequentialTagger` will be used.

8. UnigramTagger

`UnigramTagger` tags each token with the tag that is most likely to go with the token's type. It uses a training corpus to decide which tag is most likely for each type. In particular, it assumes that the tag that occurs most frequently with a type is the most likely tag for that type. For example, if the training corpus contains the word "track" as a noun 18 times, and as a verb 7 times, then it will assign the noun tag to any tokens whose type is "track." [1]

UnigramTagger uses a `ConditionalFreqDist` to record the most likely tag for each type. [2] The `train` method constructs this conditional frequency distribution from a training corpus.

8.1. Training the Unigram Tagger

Tagging is a prediction problem. In particular, the outcome we are interested in is the tag; and the context that we will use to predict the outcome is the token's type. So we will construct a `ConditionalFreqDist` whose samples are tags, and whose conditions are token types:

```
def train(self, tagged_tokens):
    for token in tagged_tokens:
        outcome = token.type().tag()
        context = token.type().base()
        self._freqdist[context].inc(outcome)
```

8.2. Tagging with the Unigram Tagger

To find the most likely tag for a given token, we can use the the indexing operator to access the `FreqDist` for the appropriate context; and use the `max` method to find the most likely outcome for that frequency distribution. For example, we could find the most likely tag for the base type "bank" as follows:

```
>>> freqdist['bank'].max()
'NN'
```

The `next_tag` method must decide which tag is most likely for a given token. It simply consults the tagger's conditional frequency distribution to find the tag that is most likely for the tokens's type.

```
def next_tag(self, tagged_tokens, next_token):
    context = next_token.type()
    return self._freqdist[context].max()
```

Note: If a context was not encountered in the training corpus, then the frequency distribution for that context will be empty; so `max()` will return `None`. Thus, `next_tag` will return `None` as for any token whose type was not encountered in the training corpus.

8.3. Initializing the Unigram Tagger

The constructor for `UnigramTagger` simply initializes `self._freqdist` with a new conditional frequency distribution.

```
    def __init__(self):
        self._freqdist = probability.ConditionalFreqDist()
```

8.4. The UnigramTagger Implementation

The complete listing for the `UnigramTagger` class is:

```
class UnigramTagger(TaggerI):
class UnigramTagger(SequentialTagger):
    def __init__(self):
        self._freqdist = ConditionalFreqDist()

    def train(self, tagged_tokens):
        for token in tagged_tokens:
            context = token.type().base()
            feature = token.type().tag()
            self._freqdist[context].inc(feature)

    def next_tag(self, tagged_tokens, next_token):
        context = next_token.type()
        return self._freqdist[context].max()
```

Figure 4. The UnigramTagger Implementation

9. NthOrderTagger

The `NthOrderTagger` is a generalization of the `UnigramTagger`. Instead of using the token's base type as a context, it uses a tuple consisting of the token's base type and the tags of the n preceding tokens. This generalization creates two new issues.

First, we must decide how to handle the first n tokens, since they do not have n preceding tokens. `NthOrderTagger` simply uses the tags that are available. For example, in a 3rd order tagger, the context of the second token will contain only the token's type and the first token's tag. Another option would be to simply ignore the first n tokens. As it turns out, which approach we take will not have much of an impact, since n (the order of the tagger) is generally much less than n_{train} (the number of training samples).

The second issue is that, when tagging a text, we do not have access the the actual tags of the n preceding tokens. However, we do have access to our predicted values for these tags. `NthOrderTagger` uses these predicted tags, since they are likely to be

correct. Assuming that our predictions are good, the use of predicted tags instead of actual tags will have a relatively minor impact on performance.

9.1. Initializing the Nth Order Tagger

Having addressed these two issues, we can examine the implementation of the `NthOrderTagger`. The constructor simply records *n*, and constructs a new conditional frequency distribution:

```
def __init__(self, n):
    self._n = n
    self._freqdist = probability.ConditionalFreqDist()
```

9.2. Training the Nth Order Tagger

To train the `NthOrderTagger`, we examine each token, and increment the count of the tag for the appropriate context. For contexts, we use a tuple consisting of the *n* previous tags and the current token's base type. We use a variable called `prev_tags` to record the rpevious *n* tags; and update it after examining each token.

```
def train(self, tagged_tokens):
    # prev_tags is a list of the previous n tags that we've assigned.
    prev_tags = []

    for token in tagged_tokens:
        context = tuple(prev_tags + [token.type().base()])
        feature = token.type().tag()
        self._freqdist[context].inc(feature)

        # Update prev_tags
        prev_tags.append(token.type().tag())
        if len(prev_tags) == (self._n+1):
            del prev_tags[0]
```

9.3. Tagging with the Nth Order Tagger

As with the `UnigramTagger`, we can find the most likely tag for each token by using the `max` method for the frequency distribution with the appropriate context. But instead of using each token's base type as a context, we use a tuple consisting of the *n* previous predicted tags and the token's base type.

```
def next_tag(self, tagged_tokens, next_token):
    # Find the tags of the n previous tokens.
    prev_tags = []
    start = max(len(tagged_tokens) - self._n, 0)
    for token in tagged_tokens[start:]:
        prev_tags.append(token.type().tag())

    # Return the most likely tag for the token's context.
    context = tuple(prev_tags + [next_token.type()])
    return self._freqdist[context].max()
```

The complete listing for the `NthOrderTagger` class is:

```
class NthOrderTagger(SequentialTagger):
    def __init__(self, n):
        self._n = n
        self._freqdist = CFFreqDist()

    def train(self, tagged_tokens):
        # prev_tags is a list of the previous n tags that we've assigned.
        prev_tags = []

        for token in tagged_tokens:
            context = tuple(prev_tags + [token.type().base()])
            feature = token.type().tag()
            self._freqdist[context].inc(feature)

            # Update prev_tags
            prev_tags.append(token.type().tag())
            if len(prev_tags) == (self._n+1):
                del prev_tags[0]

    def next_tag(self, tagged_tokens, next_token):
        # Find the tags of the n previous tokens.
        prev_tags = []
        start = max(len(tagged_tokens) - self._n, 0)
        for token in tagged_tokens[start:]:
            prev_tags.append(token.type().tag())

        # Return the most likely tag for the token's context.
        context = tuple(prev_tags + [next_token.type()])
        return self._freqdist[context].max()
```

Figure 5. The NthOrderTagger Implementation

10. BackoffTagger

The `BackoffTagger` is used to combine the results of a list of *subtaggers*. For each token to be tagged, the `BackoffTagger` consults each subtagger, in order. Each token is assigned the first non-`None` tag returned by a subtagger for that token. If all of the subtaggers return the tag `None` for a token, then `BackoffTagger` will assign it the tag `None`.

10.1. Initializing a Backoff Tagger

The `BackoffTagger` constructor simply records the list of subtaggers.

```
    def __init__(self, subtaggers):
        self._taggers = subtaggers
```

The implementation of `BackoffTagger` is relatively straight-forward. Its `next_tag` method simply calls each subtagger's `next_tag` method, in order; and returns the first non-`None` tag produced by a subtagger.

```
    def next_tag(self, tagged_tokens, next_token):
        for subtagger in self._subtaggers:
            tag = subtagger.next_tag(tagged_tokens, next_token)
            if tag is not None:
                return tag

        # Default to None if all subtaggers return None.
        return None
```

10.3. The BackoffTagger Implementation

The complete listing for the `BackoffTagger` class is:

```
class BackoffTagger(SequentialTagger):
    def __init__(self, subtaggers):
        self._subtaggers = subtaggers

    def next_tag(self, tagged_tokens, next_token):
        for subtagger in self._subtaggers:
            tag = subtagger.next_tag(tagged_tokens, next_token)
            if tag is not None:
                return tag

        # Default to None if all subtaggers return None.
        return None
```

Figure 6. The BackoffTagger Implementation

11. Exercises

11.1. Combining Taggers with BackoffTagger

There is typically a trade-off between the accuracy and coverage for taggers: taggers that use more specific contexts usually produce more accurate results, when they have seen those contexts in the training data; but because the training data is limited, they are less likely to encounter each context. The `BackoffTagger` addresses this problem by trying taggers with more specific contexts first; and falling back to the more general taggers when necessary. In this exercise, we examine the effects of using `BackoffTagger`.

1. Create an `NN_CD_Tagger`, a `UnigramTagger`, and a `NthOrderTagger`. Train the `UnigramTagger`, and the `NthOrderTagger` using a tagged section of the Brown corpus.

Test the performance of each tagger, using a tagged section of the Brown corpus. Record the *accuracy* of the tagger (the percentage of tokens that are correctly tagged). Be sure to use a different section of the corpus for testing than you used for training.

3. Use `BackoffTagger` to create three different combinations of the basic taggers. Test the accuracy of each combined tagger. Which combinations give the most improvement?
4. Try repeating steps 1-3 with a different sized training corpus. How does it affect your results?

11.2. Tagger Context

`NthOrderParser` chooses a tag for a token based on its type and the tags of the n preceeding tokens. This is a common context to use for parsing, but ceratinly not the only possible context.

Construct a new tagger, subclassed from `SequentialTagger`, that uses a different context. If your tagger's context contains multiple elements, then you should combine them in a `tuple` (see NthOrderTagger for an example of this). Some possibilities for elements to include are:

- The base type of the current token, or of a previous token.
- The length of the current token's type, or of a previous token's type.
- The first letter of the current token's type, or of a previous token's type.
- The tag a previous token.

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.

Use `BackoffTagger` to combine your tagger with other taggers. How does the combined tagger's accuracy compare to the basic tagger? How does the combined tagger's accuracy compare to the combined taggers you created in the previous exercise?

11.3. Reverse Sequential Taggers

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 can provide more information about what tag should be used. A *reverse sequential tagger* is a tagger that:

1. Assigns tags to one token at a time, starting with the last token of the text, and proceeding in right-to-left order.
2. Decides which tag to assign a token on the basis of that token, the tokens that follow it, and the predicted tags for the tokens that follow it.

There is no need to create new classes to perform reverse sequential tagging. By reversing texts at appropriate times, we can use sequential tagging classes to perform reverse sequential tagging. In particular, we should reverse the training text before we train the tagger; and reverse the text that we wish to tag both before and after we use the sequential tagger.

Use this technique to create a first order reverse sequential tagger. 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. How does its accuracy compare to a first order sequential tagger, using the same training data and test data?

11.4. Processing Individual Sentences

[to be written] Write a modified nth order tagger, that ignores tags that are in a previous sentence. E.g., for a 3nd order tagger, if the previous 3 words were "dog/NN ./. A/DT", then just use "DT" and the current token as context.

11.5. Alternatives to Backoff

[to be written] Create a new kind of tagger that combines 2 or more subtaggers.

Index

Notes

1. We considered using a conditional probability distribution, instead of a conditional frequency distribution. However, for most probability distributions, the maximum probability sample is always equal to the maximum frequency sample in the underlying frequency distributions. We decided that the additional complexity involved in using `ConditionalProbDist was not justified.`
2. See the probability tutorial for information about constructing and using frequency distributions.