Automatic Part-of-Speech Tagging

Behrang QasemiZadeh

behrangatoffice@gmail.com

Automatic Part-of-Speech Tagging

- Reminder: Part-of-Speech (PoS) tagging is the process of classifying words into **lexical categories:**
  - We predict/guess the PoS category of a word.
  - The goal is to be correct inasmuch as possible.

Automatic Part-of-Speech Tagging

- How to make a part-of-speech tagger?
  - Using a single tag (the most frequent tag), e.g., tag everything as noun.
  - Rule-based methods: use *morphological* patterns, e.g., look into suffixes and prefixes, to determine the PoS of words.
  - Use a dictionary look-up.
  - Use statistical techniques.

Automatic Part-of-Speech Tagging

- In the remaining of this session we work with tagged sentences.

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = \
      brown.tagged_sents(categories='news', tagset='universal')
>>> brown_sents = brown.sents(categories='news')
```

The Default Tagger

- The simplest tagger can be implemented by assigning the same tag (e.g. the most frequent tag) to each token.

The Default Tagger

- The simplest tagger can be implemented by assigning the same tag (e.g. the most frequent tag) to each token.

```
>>> tags = [tag for (word, tag) in \
      brown.tagged_words(categories='news')]
>>> nltk.FreqDist(tags).max()
'NN‘
>>> default_tagger = nltk.DefaultTagger('NN')
```

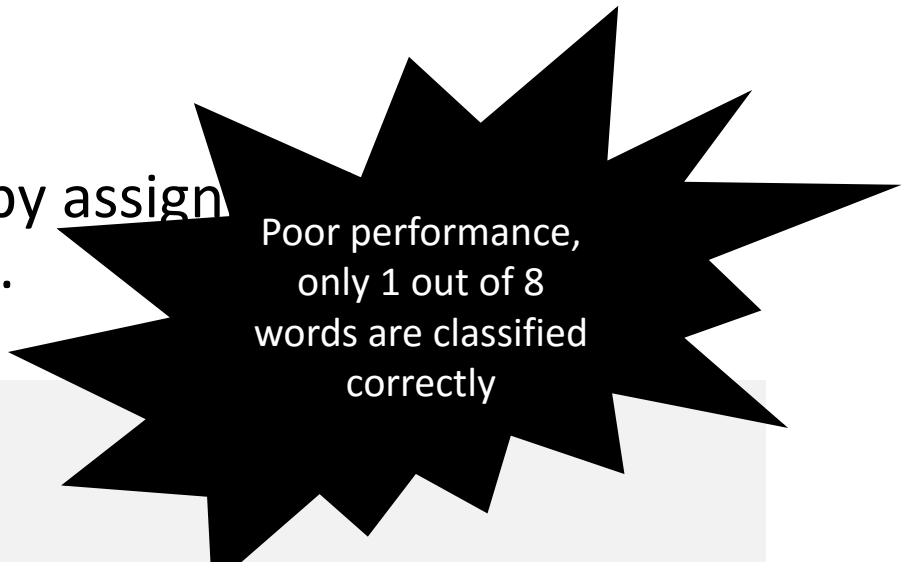
The Default Tagger

- The simplest tagger can be implemented by assigning the same tag (e.g. the most frequent tag) to each token.

```
>>> raw = 'I do not like green eggs'
>>> tokens = word_tokenize(raw)
>>> default_tagger.tag(tokens)
[('I', 'NN'), ('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('green', 'NN'), ('eggs', 'NN')]
```

The Default Tagger

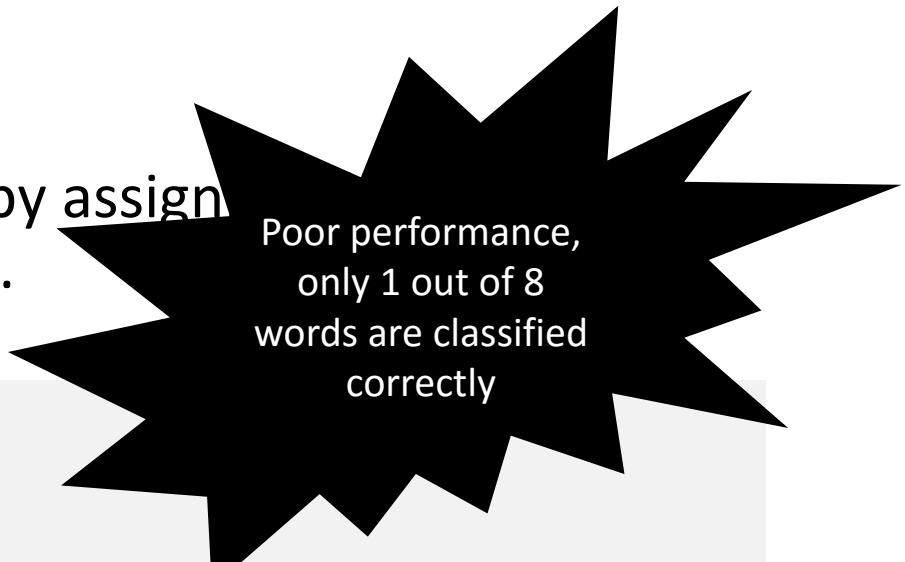
- The simplest tagger can be implemented by assign
  (e.g. the most frequent tag) to each token.




```
>>> raw = 'I do not like green eggs'
>>> tokens = word_tokenize(raw)
>>> default_tagger.tag(tokens)
[('I', 'NN'), ('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('green', 'NN'), ('eggs', 'NN')]
```

The Default Tagger

- The simplest tagger can be implemented by assign
  (e.g. the most frequent tag) to each token.




```
>>> raw = 'I do not like green eggs'
>>> tokens = word_tokenize(raw)
>>> default_tagger.tag(tokens)
[('I', 'NN'), ('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('green', 'NN'), ('eggs', 'NN')]
```

```
>>> default_tagger.evaluate(brown_tagged_sents)
0.13089484257215028
```

The Regular Expression Tagger

- Rule-based methods: use morphological patterns, e.g. look into suffixes and prefixes, to determine the PoS of words.

The Regular Expression Tagger

- Rule-based methods: use morphological patterns, e.g. look into suffixes and prefixes, to determine the PoS of words.

```
>>> patterns = [
      (r'.*ing$', 'VBG'), # gerunds
      (r'.*ed$', 'VBD'), # simple past
      (r'.*es$', 'VBZ'), # 3rd singular present
      (r'.*ould$', 'MD'), # modals
      (r'.*\'s$', 'NN$'), # possessive nouns
      (r'.*s$', 'NNS'), # plural nouns
      (r'^-?[0-9]+(.[0-9]+)?$', 'CD'), # cardinal numbers
      (r'.*', 'NN') # nouns (default)
]
```

The Regular Expression Tagger

- Rule-based methods: use morphological patterns, e.g. look into suffixes and prefixes, to determine the PoS of words.
  - Use the RE patterns to tag sentences!

The Regular Expression Tagger

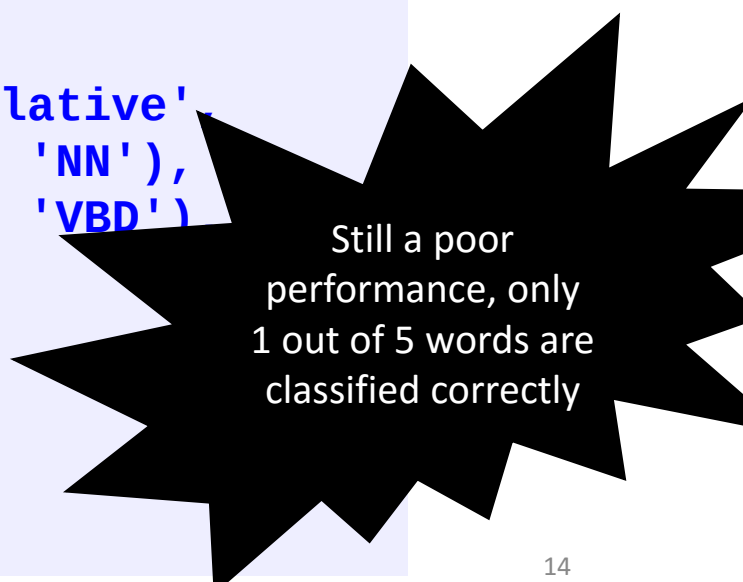
- Rule-based methods: use morphological patterns, e.g. look into suffixes and prefixes, to determine the PoS of words.
  - Use the RE patterns to tag sentences!

```
>>> regexp_tagger = nltk.RegexpTagger(patterns)
>>> regexp_tagger.tag(brown_sents[3])
[('``', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative',
'NN'), ('handful', 'NN'), ('of', 'NN'), ('such', 'NN'),
('reports', 'NNS'), ('was', 'NNS'), ('received', 'VBD'),
("''", 'NN'), (',', 'NN'), ('the', 'NN'), ...]
```

The Regular Expression Tagger

- Rule-based methods: use morphological patterns, e.g. look into suffixes and prefixes, to determine the PoS of words.
  - Use the RE patterns to tag sentences!

```
>>> regexp_tagger = nltk.RegexpTagger(patterns)
>>> regexp_tagger.tag(brown_sents[3])
[('``', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative',
'NN'), ('handful', 'NN'), ('of', 'NN'), ('such', 'NN'),
('reports', 'NNS'), ('was', 'NNS'), ('received', 'VBD'),
("''", 'NN'), (',', 'NN'), ('the', 'NN'), ...]
```



The Regular Expression Tagger

- Rule-based methods: use morphological patterns, e.g. look into suffixes and prefixes, to determine the PoS of words.
  - Use the RE patterns to tag sentences!

```
>>> regexp_tagger = nltk.RegexpTagger(patterns)
>>> regexp_tagger.tag(brown_sents[3])
[('``', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative',
'NN'), ('handful', 'NN'), ('of', 'NN'), ('such', 'NN'),
('reports', 'NNS'), ('was', 'NNS'), ('received', 'VBD'),
("''", 'NN'), (',', 'NN'), ('the', 'NN'), ...]
>>> regexp_tagger.evaluate(brown_tagged_sents)

0.20326391789486245
```

The Unigram (Look-up) Tagger

- Find the hundred most frequent words and store their most likely tag.
- Use this information as the model for a "lookup tagger"
  - i.e. an NLTK **`UnigramTagger`**

The Lookup Tagger

- Find the hundred most frequent words and store their most likely tag.
- Use this information as the model for a "lookup tagger"
  - i.e. an NLTK **`UnigramTagger`**

```
>>> fd = nltk.FreqDist(brown.words(categories='news'))
>>> cfd = nltk.ConditionalFreqDist(\
      brown.tagged_words(categories='news', tagset='universal'))
>>> most_freq_words = [word[0] for word in fd.most_common(100)]
>>> likely_tags = dict((word, cfd[word].max()) for \
      word in most_freq_words)
>>> baseline_tagger = nltk.UnigramTagger(model=likely_tags)
```

The Lookup Tagger

- Find the hundred most frequent words and store their most likely tag.
- Use this information as the model for a "lookup tagger"
  - i.e. an NLTK **`UnigramTagger`**

```
>>> fd = nltk.FreqDist(brown.words(categories='news'))
>>> cfd = nltk.ConditionalFreqDist(\
      brown.tagged_words(categories='news', tagset='universal'))
>>> most_freq_words = [word[0] for word in fd.most_common(100)]
>>> likely_tags = dict((word, cfd[word].max()) for \
      word in most_freq_words)
>>> baseline_tagger = nltk.UnigramTagger(model=likely_tags)
```

```
>>> baseline_tagger.evaluate(brown_tagged_sents)
0.45578495136941344
```

Much better performance

The Lookup Tagger (backoff)

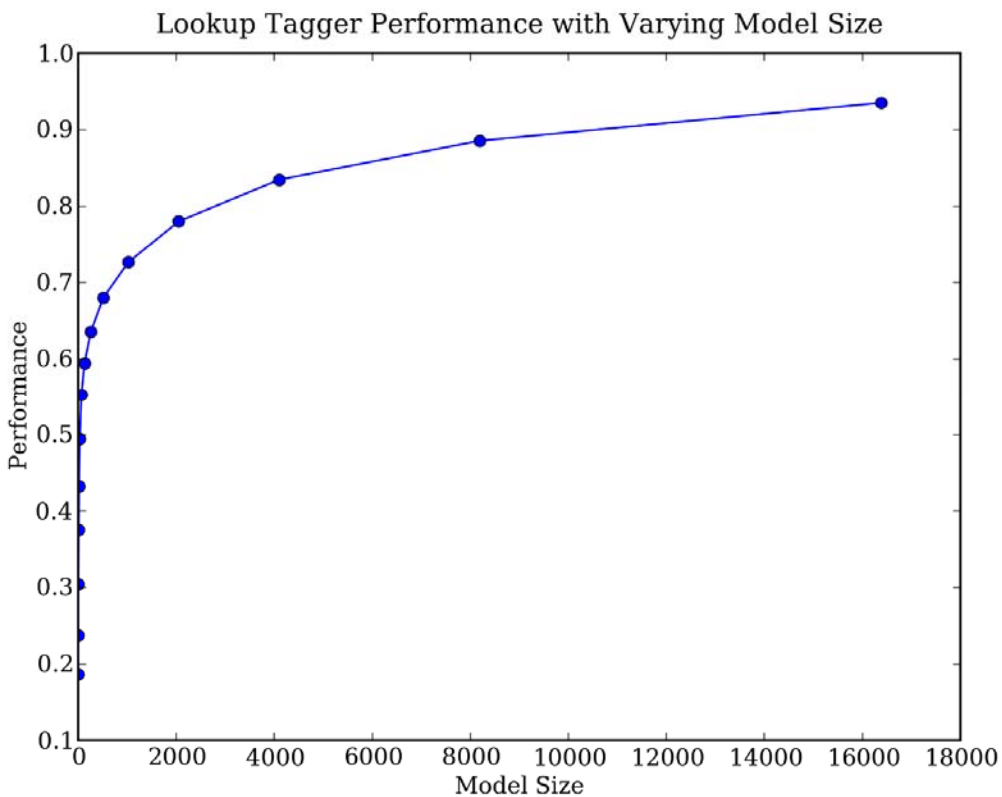
- The output of the previous tagger is full of “None” tags:
  - Words that are not among the 100 most frequent words.
- In these cases we can assign words to a default tag:
  - First, use the lookup table
  - If a word is not found, then use the default tagger
- The above process known as **backoff.**

The Lookup Tagger (backoff)

- The output of the previous tagger is full of “None” tags:
  - Words that are not among the 100 most frequent words.
- In these cases we can assign words to a default tag:
  - First, use the lookup table
  - If a word is not found, then use the default tagger
- The above process known as **backoff.**

```
>>> baseline_tagger = nltk.UnigramTagger(model=likely_tags,
      backoff=nltk.DefaultTagger('NN'))
```

The Lookup Tagger



- Performance initially increases rapidly as the model size grows, eventually reaching a plateau, when large increases in model size yield little improvement in performance.

Evaluation

- In natural language processing, evaluation is a central theme.
- As you witnessed**,** we use **gold standard** test data for the evaluation:
  - **gold standard** test data: a corpus which has been manually annotated and it is accepted as a standard against which the guesses of an automatic system are assessed.
  - In pervious examples, the tagged sentences from the Brown corpus.
- **Separating the Training and Testing Data:** to avoid over fitting!
  - A tagger may simply memorized its training data and has no generalization ability, i.e. it may be useless for tagging new text.

N-Gram Tagging

- Unigram taggers are based on a simple statistical algorithm:
  - For each token, assign the tag that is most likely for that particular token.

N-Gram Tagging

- Unigram taggers are based on a simple statistical algorithm:
  - For each token, assign the tag that is most likely for that particular token.
- For unigram taggers, the training process involves storing the most likely tag for any word in a dictionary:
  - model = a dictionary (as we developed few moments ago)

N-Gram Tagging

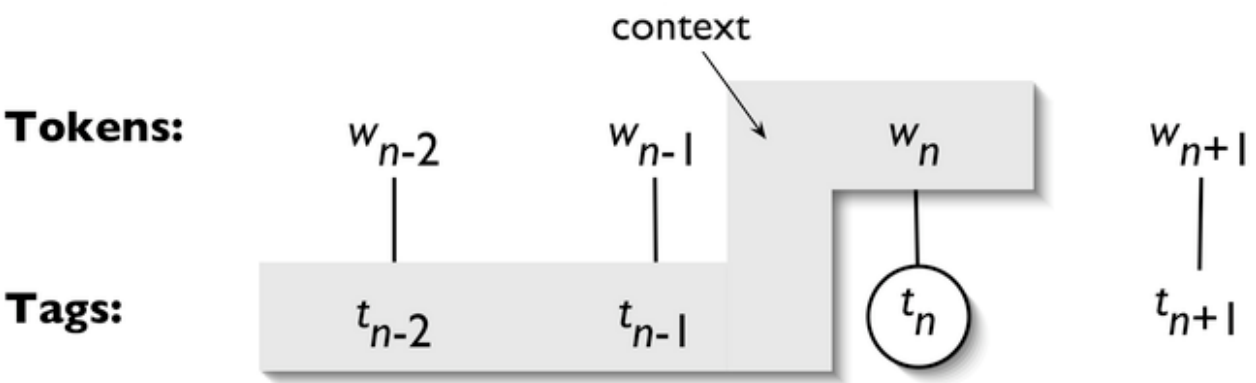
- Unigram taggers are based on a simple statistical algorithm:
  - For each token, assign the tag that is most likely for that particular token.
- For unigram taggers, the training process involves storing the most likely tag for any word in a dictionary:
  - model = a dictionary (as we developed few moments ago)
- When using unigrams, we limit the context to one word:
  - That is, current token in isolation.
  - We tag a word like “book” always as known no matter the context is “to book” or “the book”.

N-Gram Tagging

- Unigram taggers are based on a simple statistical algorithm:
  - For each token, assign the tag that is most likely for that particular token.
- For unigram taggers, the training process involves storing the most likely tag for any word in a dictionary:
  - model = a dictionary (as we developed few moments ago)
- When using unigrams, we limit the context to one word:
  - That is, current token in isolation.
  - We tag a word like “book” always as known no matter the context is “to book” or “the book”.
- An **n-gram tagger** is a generalization of a unigram tagger.

N-Gram Tagging

- The context in an **n-gram tagger** is the current word together with the part-of-speech tags of the *n*-1 preceding tokens.



- The **`NgramTagger`** class uses a tagged training corpus to determine which part-of-speech tag is most likely for each context.

N-Gram Tagging: bigram tagging

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
```

N-Gram Tagging: bigram tagging

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
```

N-Gram Tagging: bigram tagging

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
```

```
>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments',
'NNS'), ('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace',
'NN'),...,('.', '.')]
```

N-Gram Tagging: bigram tagging

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
```

```
>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments',
'NNS'), ('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace',
'NN'),...,('.', '.')]
```

N-Gram Tagging: bigram tagging

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
```

```
>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments',
'NNS'), ('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace',
'NN'),...,('.', '.')]
```

```
>>> bigram_tagger.evaluate(test_sents)
0.102063...
```

N-Gram Tagging: bigram tagging

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
```

```
>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments',
'NNS'), ('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace',
'NN'),...,('.', '.')]
```

```
>>> bigram_tagger.evaluate(test_sents)
0.102063...
```

N-Gram Tagging: bigram tagging

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
```

```
>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments',
'NNS'), ('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace',
'NN'),...,('.', '.')]
```

```
>>> bigram_tagger.evaluate(test_sents)
0.102063...
```

This is due to the problem known as the *data sparsity*:

- As *n* gets larger, the specificity of the contexts increases, as does the chance that the data we wish to tag contains contexts that were not present in the training data.

Combining Taggers

- When developing a tool like PoS tagger, there is a trade-off between the accuracy and the coverage of our results.

Combining Taggers

- When developing a tool like PoS tagger, there is a trade-off between the accuracy and the coverage of our results.
- One way to address the trade-off between accuracy and coverage is to combine algorithms:
  - use the more accurate algorithms when possible, but when it is necessary use algorithms with wider coverage.

Combining Taggers

- When developing a tool like PoS tagger, there is a trade-off between the accuracy and the coverage of our results.
- One way to address the trade-off between accuracy and coverage is to combine algorithms:
  - use the more accurate algorithms when possible, but when it is necessary use algorithms with wider coverage.
- In the PoS tagging scenario:
  - Try tagging a token with the bigram tagger.
  - If the bigram tagger fails to find a tag for the token, try the unigram tagger.
  - If the unigram tagger is also unable to find a tag, use a default tagger.

Combining Taggers

```
>>> t0 = nltk.DefaultTagger('NN')
>>> t1 = nltk.UnigramTagger(train_sents, backoff=t0)
>>> t2 = nltk.BigramTagger(train_sents, backoff=t1)
>>> t2.evaluate(test_sents)
0.844513...
```

Combining Taggers

```
>>> t0 = nltk.DefaultTagger('NN')
>>> t1 = nltk.UnigramTagger(train_sents, backoff=t0)
>>> t2 = nltk.BigramTagger(train_sents, backoff=t1)
>>> t2.evaluate(test_sents)
0.844513...
```

- The tagger with a smaller context backs off the tagger with the context of larger size.

Combining Taggers

```
>>> t0 = nltk.DefaultTagger('NN')
>>> t1 = nltk.UnigramTagger(train_sents, backoff=t0)
>>> t2 = nltk.BigramTagger(train_sents, backoff=t1)
>>> t2.evaluate(test_sents)
0.844513...
```

- Exercise: extend the above example by defining a **`TrigramTagger`**, which backs off to **`t2`**.

Tagging Unknown (out-of-vocabulary) Words

- In the previous slide, the proposed regular-expression tagger or a default tagger are unable to make use of context.

Tagging Unknown (out-of-vocabulary) Words

- In the previous slide, the proposed regular-expression tagger or a default tagger are unable to make use of context.
- One useful method is to limit the the vocabulary of a tagger to the most frequent *n* words, and to replace every other word with a special word *UNK:*

Tagging Unknown (out-of-vocabulary) Words

- In the previous slide, the proposed regular-expression tagger or a default tagger are unable to make use of context.
- One useful method is to limit the the vocabulary of a tagger to the most frequent *n* words, and to replace every other word with a special word *UNK:*
  - During training, a unigram tagger will probably learn that *UNK* is often a noun.
  - However, the n-gram taggers will detect contexts in which it has some other tag, e.g. that a verb follows the preposition *to*.

Storing Taggers

- Training a tagger on a large corpus may take a significant time.
- Instead of training a tagger every time we need one, it is convenient to save a trained tagger in a file for later re-use.

Storing Taggers

- Training a tagger on a large corpus may take a significant time.
- Instead of training a tagger every time we need one, it is convenient to save a trained tagger in a file for later re-use.

To save a model

```
>>> from pickle import dump
>>> output = open('t2.pkl', 'wb')
>>> dump(t2, output, -1)
>>> output.close()
```

To load a model

```
>>> from pickle import load
>>> input = open('t2.pkl', 'rb')
>>> tagger = load(input)
>>> input.close()
```

Performance Limitations

- What is the upper limit to the performance of an n-gram tagger?
  - How many cases of PoS ambiguity does a trigram tagger encounter?

Performance Limitations

- What is the upper limit to the performance of an n-gram tagger?
  - How many cases of PoS ambiguity does a trigram tagger encounter?

```
>>> cfd = nltk.ConditionalFreqDist(
      ((x[1], y[1], z[0]), z[1])
      for sent in brown_tagged_sents
      for x, y, z in nltk.trigrams(sent))
>>> ambiguous_contexts = [c for c in cfd.conditions() if len(cfd[c]) > 1]
>>> float(sum(cfd[c].N() for c in ambiguous_contexts) )/ cfd.N()
0.04929770206802929 6
```

Performance Limitations

- What is the upper limit to the performance of an n-gram tagger?
  - How many cases of PoS ambiguity does a trigram tagger encounter?

```
>>> cfd = nltk.ConditionalFreqDist(
      ((x[1], y[1], z[0]), z[1])
      for sent in brown_tagged_sents
      for x, y, z in nltk.trigrams(sent))
>>> ambiguous_contexts = [c for c in cfd.conditions() if len(cfd[c]) > 1]
>>> float(sum(cfd[c].N() for c in ambiguous_contexts) )/ cfd.N()
0.04929770206802929 6
```

In 5% of cases there is more than one tag that could be legitimately assigned to the current word according to the training data!

Performance Limitations

- What is the upper limit to the performance of an n-gram tagger?
  - How many cases of PoS ambiguity does a trigram tagger encounter?
  - Assuming we always pick the most likely tag in ambiguous contexts, we can derive a lower bound on the performance of a tagger.

Performance Limitations

- What is the upper limit to the performance of an n-gram tagger?
  - How many cases of PoS ambiguity does a trigram tagger encounter?
  - Assuming we always pick the most likely tag in ambiguous contexts, we can derive a lower bound on the performance of a tagger.
- Another way to investigate the performance of a tagger is to study its mistakes.
  - Some tags may be harder than others to assign
  - It might be possible to treat them specially by pre or post-processing the data.

Performance Limitations

- What is the upper limit to the performance of an n-gram tagger?
  - How many cases of PoS ambiguity does a trigram tagger encounter?
  - Assuming we always pick the most likely tag in ambiguous contexts, we can derive a lower bound on the performance of a tagger.
- Another way to investigate the performance of a tagger is to study its errors.
  - Some tags may be harder than others to assign
  - It might be possible to treat them specially by pre or post-processing the data.
  - A convenient way to look at tagging errors is the **confusion matrix (**expected tags (the gold standard) against actual tags generated by a tagger).

Performance Limitations

- **Confusion matrix**

```
>>> test_tags = [tag for sent in brown.sents(categories='editorial')
                 for (word, tag) in tagger.tag(sent)]
>>> gold_tags = [tag for (word, tag) in
      brown.tagged_words(categories='editorial')]
>>> print(nltk.ConfusionMatrix(gold_tags, test_tags))
```

Some observations

1. Definition of the tagset is an important that influences the performance of a tagger.

Some observations

1. The definition of a tagset is an important factor that influences the performance of a tagger.

- Tagging process collapses distinctions:
  - e.g. lexical identity is usually lost when all personal pronouns are tagged PRP.
- Tagging process introduces new distinctions and removes ambiguities:
  - e.g. *deal* tagged as *VB* or *NN*.
- These collapsing distinctions and introducing new distinctions are valuable in classification tasks but also in the overall performance of a tagger.

Some observations

1. Definition of the tagset is an important that influences the performance of a tagger.

2. In n-gram tagging, a potential issue is the size of their n-gram table (language model):

- It is important to strike a balance between model size and tagger performance.

Some observations

1. Definition of the tagset is an important that influences the performance of a tagger.

2. In n-gram tagging, a potential issue is the size of their n-gram table (language model):
   - It is important to strike a balance between model size and tagger performance.

3. The information used in our implemented n-gram tagger is limited to the prior context:
   - words themselves might be a useful source of information!

Transformation-Based Tagging

- Brill tagger, is a kind of *transformation-based learning*, named after Eric Brill its inventor:
  - An inductive tagging method which performs very well using models that are only a tiny fraction of the size of n-gram taggers.

Transformation-Based Tagging

- Brill tagger, is a kind of *transformation-based learning*, named after Eric Brill its inventor:
  - An inductive tagging method which performs very well using models that are only a tiny fraction of the size of n-gram taggers.
- The general idea is very simple but effective:
  - guess the tag of each word, then go back and fix the mistakes.
- A Brill tagger successively transforms a bad tagging of a text into a better one.

Transformation-Based Tagging

- Brill tagger, is a kind of *transformation-based learning*, named after Eric Brill its inventor:
  - An inductive tagging method which performs very well using models that are only a tiny fraction of the size of n-gram taggers.
- The general idea is very simple but effective:
  - guess the tag of each word, then go back and fix the mistakes.
- A Brill tagger successively transforms a bad tagging of a text into a better one.
- It is a supervised learning technique (similar to n-gram tagging):
  - i.e. annotated data is required.

Transformation-Based Tagging

- In contrast to n-gram tagging, Brill tagging does not count observations but it compiles a list of transformational correction rules.
- Rules are in the general form of "replace T_1 with T_2 in the context *C.*"

Transformation-Based Tagging

- In contrast to n-gram tagging, Brill tagging does not count observations but it compiles a list of transformational correction rules.
- Rules are in the general form of "replace T_1 with T_2 in the context *C.*"
- During its training phase, the tagger guesses values for T_1, T_2 and *C*.
- Examples of such rules are:
  - Replace *NN* with *VB* when the previous word is *TO.*
  - Replace *TO* with *IN* when the next tag is *NNS.*

Transformation-Based Tagging

- In contrast to n-gram tagging, Brill tagging does not count observations but it compiles a list of transformational correction rules.
- Rules are in the general form of "replace T_1 with T_2 in the context *C.*"
- During its training phase, the tagger guesses values for T_1, T_2 and *C*.
- Examples of such rules are:
  - Replace *NN* with *VB* when the previous word is *TO.*
  - Replace *TO* with *IN* when the next tag is *NNS.*
- Each rule is scored according to its net benefit: the number of incorrect tags that it corrects.

Transformation-Based Tagging

- In contrast to n-gram tagging, Brill tagging does not count observations but it compiles a list of transformational correction rules.
- Rules are in the general form of "replace T_1 with T_2 in the context C."
- During its training phase, the tagger guesses values for T_1, T_2 and C.
- Examples of such rules are:
  - Replace *NN* with *VB* when the previous word is *TO.*
  - Replace *TO* with *IN* when the next tag is *NNS.*
- Each rule is scored according to its net benefit: the number of incorrect tags that it corrects.

```
nltk.tag.brill.demo()
```

Further Reading

- Chapters 4 and 5 of (Jurafsky & Martin, 2008) contain more advanced material on n-grams and part-of-speech tagging.
- Look at HOWTO sections of NLTK website (http://nltk.org/howto).
  - http://www.nltk.org/howto/probability.html