

M02S03- Information Extraction

NLP – Syntactic Processing

1. Introduction
2. Understanding the ATIS data
3. Information Extraction
4. POS Tagging
5. Rule-Based Models
6. Probabilistic Models for Entity Recognition
7. Naive Bayes Classifier for NER
8. Decision Tree Classifiers for NER
9. HMM and IOB labelling
10. Summary
11. Graded Questions
12. CRFs - Another Probabilistic Approach

Introduction

Even though textual data is widely available, the complexity of natural language makes it extremely difficult to extract useful information from text. This session will build an **Information Extraction (IE)** system that can extract structured data from unstructured textual data. A key component in information extraction systems is **Named-Entity-Recognition (NER)**. Various techniques and models for building NER systems in this session.

Make a conversational flight-booking system to show relevant flights when given a natural-language query such as “Please show me all morning flights from Bangalore to Mumbai on next Monday.”. To process this query, system must extract useful **named entities** from the unstructured text query and convert them to a structured format, such as the following dictionary/JSON object:

```
{source: 'Bangalore',  
destination: 'Mumbai',  
day: 'Monday',  
time-of-day: 'morning'}
```

Using these entities, query a database and get all relevant flight results. In general, named entities refer to names of people, organizations (Google), places (India, Mumbai), specific dates and time (Monday, 8 pm) etc.

This session will build such systems which can extract structured information from unstructured text data.

In this session

This session will introduce:

1. Named Entity Recognition
 - I-O-B labels
2. Building models for Entity Recognition
 - Rule-based techniques
 - Regular expression-based techniques
 - Chunking
 - Probabilistic models
 - Unigram & Bigram models
 - Naive Bayes Classifier
 - Decision trees
 - Conditional Random Fields (CRFs) -Optional

We'll use the **ATIS (Airline Travel Information Systems)** dataset to build an IE system. The ATIS dataset consists of English language queries for booking (or requesting information about) flights in the US. Download the ATIS dataset from [here](#).

Understanding the ATIS data

The ATIS (Airline Travel Information Systems) dataset consists of English language queries for booking (or requesting information about) flights in the US.

Each word in a query (i.e. a request by a user) is labelled according to its **entity-type**, for e.g. in the query 'please show morning flights from chicago to new york', 'chicago' and 'new york' are labelled as 'source' and 'destination' locations respectively while 'morning' is labelled as 'time-of-day' (the exact labelling scheme is a bit different, more on that later).

Some example queries taken from the dataset:

```
{
'what flights leave atlanta at about DIGIT in the afternoon and arrive in san francisco',
'what is the abbreviation for canadian airlines international',
'i 'd like to know the earliest flight from boston to atlanta',
'show me the us air flights from atlanta to boston',
'show me the cheapest round trips from dallas to baltimore',
'i 'd like to see all flights from denver to philadelphia"
}
```

Dataset has five zip files (or folds). Each zip file has three datasets: train, test and validation, and a dictionary. Available from <http://lisaweb.iro.umontreal.ca/transfert/lisa/users/mesnilgr/atis/>. All folds are structurally identical, so understanding one is enough to understand the entire set.

All the three datasets are in form of tuples containing three lists. The first list is the tokenized words of the queries, encoded by integers such as 554, 194, 268 ... and so on. For e.g. the first three integers 554, 194, 268 are encoded values of the words 'what', 'flights', 'leave' etc. Ignore the second list. The third list contains the (encoded) label of each word.

```
# import libraries
import numpy as np
import pandas as pd
import nltk, pprint
import matplotlib.pyplot as plt
import random

import gzip, os, pickle # gzip for reading the gz files, pickle to save/dump trained model
import _pickle as cPickle

import sklearn
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
```

Labels are similar to POS tags, where instead of using noun, verb, etc, we'll use **IOB (inside-outside-beginning) tags** of entities like flight-time, source-city, etc.

Let's now read the first fold of the dataset. The data is in .gz files, so we'll need the gzip library as well.

```
# read the first part of the dataset
# each part (.gz file) contains train, validation and test sets, plus a dict

filename = 'atis.fold0.pkl.gz'
f = gzip.open(filename, 'rb')
try:
    train_set, valid_set, test_set, dicts = pickle.load(f, encoding='latin1')
except:
    train_set, valid_set, test_set, dicts = pickle.load(f)
finally:
    f.close()
```

Thus, we have the train, validation and test sets each containing three lists of different lengths.

The first list contains the actual queries encoded by integers such as 554, 194, 268 ... and so on. For e.g. the first three integers 554, 194, 268 are encoded values of the words 'what', 'flights', 'leave' etc.

The second list is to be ignored.

The third list contains the (encoded) label of each word. Since the actual words are encoded by numbers, we have to decode them using the dicts provided.

```
# each list in the tuple is a numpy array (which us a complete sentence/query)
# printing first list in the tuple's first element
# each element represents a word of the query
# this translates to 'what flights leave atlanta ....'
train_x[0]

array([554, 194, 268, 64, 62, 16, 8, 234, 481, 20, 40, 58, 234,
       415, 205])

# labels are stored in the third list train_label
train_label[0]

array([126, 126, 126, 48, 126, 36, 35, 126, 126, 33, 126, 126, 126,
       78, 123])

To map the integers to words, we need to use the dictionaries provided.
The dicts ``words2idx`` and ``labels2idx`` map the numeric ids to
the actual words and labels respectively.

# dicts to map numbers to words/labels
print(type(dicts))
print(dicts.keys())

<class 'dict'>
dict_keys(['labels2idx', 'tables2idx', 'words2idx'])

# each key of 'words' is a word, each value its index
# printing some random key:value pairs of 'words'
random.sample(words.items(), 10)

[('go', 391),
 ('fare', 182),
 ('north', 343),
 ('goes', 214),
 ('thank', 479),
 ('listing', 277),
 ('weekdays', 550),
 ('makes', 294),
 ('those', 492),
 ('lunch', 290)]

# labels dict contains IOB (inside-out-beginning) labelled entities
# printing some random k:v pairs
random.sample(labels.items(), 25)

[('I-time', 120),
 ('I-flight_mod', 104),
 ('I-airline_name', 83),
 ('B-depart_time.end.time', 31),
 ('B-toloc.city_name', 78),
 ('B-meal', 51),
 ('B-booking_class', 16),
 ('I-today_relative', 121),
 ('I-toloc.airport_name', 122),
 ('I-class_type', 52),
 ('I-depart_time.period_of_day', 97),
 ('B-depart_time.start.time', 34),
 ('B-period_of_day', 57),
 ('B-arrive_date.day_number', 7),
 ('B-arrive_time.time_relative', 15),
 ('B-fromloc.airport_code', 46),
 ('I-fare_basis_code', 103),
 ('B-stoploc.city_name', 71),
 ('B-flight_mod', 42),
 ('B-arrive_date.date_relative', 5),
 ('B-fromloc.state_code', 49),
 ('B-aircraft_code', 0),
 ('B-city_name', 17),
 ('B-return_date.today_relative', 63),
 ('B-month_name', 55)]

There are 127 classes of labels (including the 'O' - tokens that do not fall into any entity).
```

Decoding the list

Let's decode the lists of words and labels using the dictionaries provided in the ATIS data set. Decode using the dictionary provided in the dataset.

There are three dictionaries in ATIS dataset, out of which two are required '**words2idx**', (which will convert the first list to words (actual words of queries)), and '**labels2idx**' (which will convert the third list to labels)

The structure of each query (asking about flight information) is quite different. We'll build a machine learning model which could fit some structure to these queries and derive relevant entities from it.

IOB Labels

Decoding using the dictionary of labels and IOB (or BIO in some texts) labelling, a standard way of labelling named entities.

IOB (or BIO) method tags each token in the sentence with one of the three labels:

- I - inside (the entity),
- O- outside (the entity)
- B - beginning (of entity).

IOB labelling is especially helpful if the entities contain multiple words. We would want our system to read words like 'Air India', 'New Delhi', etc, as single entities.

Consider the following example for IOB labelling:

I	booked	a	table	at	Smoke	House	Deli	for	two	on	Wednesday	at	8:00	pm
O	O	B-NP	B-NP	O	B-restname	I- restname	I-restname	O	B-count	O	B-day	O	B-time	I-time

Any entity with more than 2 words such as 'Dallas Fort Worth', 'Smoke House Deli', the first word of the entity would be labelled as B-entity and other words in it would be labelled as I-entity, rest would be labelled as O.

IOB labels

Select whether the following IOB labeling for the given sentence is correct or not:

'Narendra Modi is Prime Minister of India'

IOB labels:

Narendra: B-Per

Modi: I-Per

is: - O

Prime: B-role

Minister: I-role

of: O

India: B-country,

☒ True

🔊 Feedback:

'Narendra', 'Modi' are words of the same entity. Similarly, 'Prime', 'Minister', 'India'

☐ False

NER

Named-entity recognition is the task of locating and classifying named entities into categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc.

Identify all named entities in the following sentence: "US President Donald Trump has announced that he is heading to Singapore on June 12 to hold talks with North Korean leader Kim Jong-un". More than one options may be correct.

☒ US ✓ Correct

🔊 Feedback:
Country name is classified into named entities

☒ Donald Trump ✓ Correct

🔊 Feedback:
Person name are classified into named entities

☐ that

☒ Singapore ✓ Correct

🔊 Feedback:
Country name is classified into named entities

☒ June 12 ✓ Correct

🔊 Feedback:
Date is classified into named entities

☒ North Korea ✓ Correct

🔊 Feedback:
Country name is classified into named entities

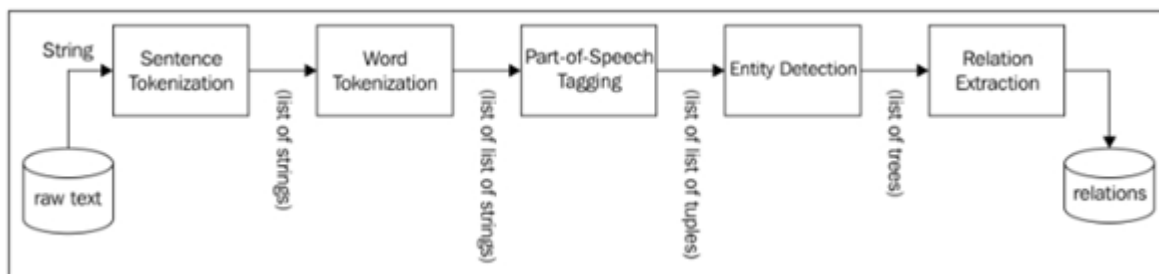
Information Extraction

Natural language is highly unstructured and complex, making it difficult for any system to process it. **Information Extraction (IE)** is the task of retrieving structured information from unstructured text data. IE is used in many applications such as **conversational** chatbots, extracting information from encyclopedias (such as Wikipedia), etc.

Information Extraction is used in a wide variety of NLP applications, such as extracting structured summaries from large corpora such as Wikipedia, conversational agents (chatbots), etc. In fact, modern virtual assistants such as Apple's Siri, Amazon's Alexa, Google Assistant etc. use sophisticated IE systems to extract information from large encyclopedias.

However, no matter how complex the IE task, there are some common steps (or subtasks) which form the pipeline of almost all IE systems.

Next, the major steps in an **IE pipeline** and build them one by one. Most IE pipelines start with the usual text pre-processing steps - sentence segmentation, word tokenisation and POS tagging. After pre-processing, the common tasks are **Named Entity Recognition (NER)**, and optionally relation recognition and record linkage. A generic IE pipeline looks something like this:



NER is arguably the most important and non-trivial task in the pipeline.

A generic IE pipeline is as follows:

1. Pre-processing

1. Sentence Tokenization: sequence segmentation of text.
2. Word Tokenization: breaks down sentences into tokens
3. POS tagging - assigning POS tags to the tokens. The POS tags can be helpful in defining what words could form an entity.

2. Entity Recognition

1. Rule-based models
2. Probabilistic models

In entity recognition, every token is tagged with an IOB label and then nearby tokens are combined basis their labels.

1. **Relation Recognition** is the task of identifying relationships between the named entities. Using entity recognition, we can identify places (pl), organisations (o), persons (p). Relation recognition will find the relation between (pl,o), such that o is located in pl. Or between (o,p), such that p is working in o, etc.
2. **Record Linkage** refers to the task of linking two or more records that belong to the same entity. For example, Bangalore and Bengaluru refer to the same entity.

Next, we will implement the first few pre-processing steps on the airlines' dataset.

NER as a Sequence-Labelling Task

The task of **training an NER system**, i.e. assigning an IOB label to each word, is a **sequence labelling task** similar to POS tagging. For sequence labelling, one can try rule-based models such as writing **regular-expression based rules** to extract entities, **chunking** patterns of POS tags into an 'entity chunk' etc. (we'll try some of these below)

On the other hand, one can use **probabilistic sequence labelling models** such as **HMMs**, the **Naive Bayes** classifier (classifying each word into one label class), **Conditional Random Fields (CRFs)** etc.

Once the IOB tags of each word are predicted, we can **evaluate the model** using the usual metrics for multi-class classification models (num_classes = number of IOB tags).

Additional Reading

- **Relation Recognition:** Further on this topic from [here](#). (Refer to the 6th segment)
- **Record Linkage:** Refer to the [record linkage](#) toolkit in Python for further reading.

POS Tagging

Since the ATIS dataset is available in the form of individual tokens, the initial pre-processing steps (-tokenisation etc.) are not required. So, **POS tagging** is the first pre-processing task. POS tagging can give a good intuition of what words could form an entity.

The main objective of this session is to learn to accurately assign IOB labels to the tokens. It is like POS tagging in that it is a **sequence labelling task**, where instead of parts-of-speech tags, we want to assign IOB labels to words.

Named Entity Recognition task identifies ‘**entities**’ in the text. Entities could refer to names of people, organizations (e.g. Air India, United Airlines), places/cities (Mumbai, Chicago), dates and time points (May, Wednesday, morning flight), numbers of specific types (e.g. money - 5000 INR) etc. POS tagging won’t be able to identify such word entities. Therefore, IOB labelling is required. So, NER task is to **predict IOB labels of each word**.

NER is a sequence labelling task where the labels are the IOB labels. There are different approaches using which we can predict the IOB labels:

1. **Rule-based techniques:**
 - Regular Expression-Based Rules
 - Chunking
2. **Probabilistic models**
 - Unigram and Bigram models
 - Naive Bayes Classifier, Decision Trees, SVMs etc.
 - Conditional Random Fields (CRFs)

IOB labeling

Ashish mentioned that it might not be a good idea to model the NER task using a conventional ML model such as naive Bayes, Decision Trees etc. Rather, it is better to use models such as HMMs. What do you think is the main reason for this?

☐ The features are independent of each other in a sequence labeling task, and we want to exploit that information

☒ The labels of each word are not independent of each other in a sequence labeling task, and we want to exploit that information ✓ **Correct**

🔍 Feedback :

Features are not independent of each other. Example: POS tag of a word is dependent on POS tag of the previous word.

☐ Sequence labeling is a complex prediction problem, and HMMs generally perform better on complex problems than conventional models

☐ All of the above

<p>POS tagging</p> <p>You saw that the NLTK POS tagger is not performing accurately on the ATIS dataset - any word after 'to/TO' gets tagged as a verb, i.e. in all the queries of the form '... from city_1 to city_2', the word 'city_2' is getting tagged as a verb rather than a noun. Note that the POS tag of 'to' is 'to/TO'. Assume that the NLTK POS tagger is an HMM model. Why do you think this error is occurring? Choose the strongest hypothesis.</p> <p><input type="radio"/> The NLTK tagger is trained on a certain training corpus, which probably contains many words in the sequence 'to/TO word/VB'. In other words, the model's emission probabilities are providing an incorrect signal</p> <p><input checked="" type="radio"/> The NLTK tagger is trained on a certain training corpus, which probably contains many words in the sequence 'to/TO word/VB'. In other words, the model's transition probabilities are providing an incorrect signal ✔ Correct</p> <p>🔗 Feedback :</p> <p><i>POS tagging highly depends on corpus on which it is trained. Also, POS of the word is dependent on the previous POS tag (transition) and on the word (emission).</i></p> <p><input type="radio"/> The NLTK tagger is trained on a certain training corpus, which probably contains names of cities tagged as city/VB. In other words, the model's emission probabilities are providing an incorrect signal</p> <p><input type="radio"/> The NLTK tagger is trained on a certain training corpus, which probably contains names of cities tagged as city/VB. In other words, the model's transition probabilities are providing an incorrect signal</p>	<p>Problems with the NLTK Tagger</p> <p>Note that almost all city/airport names that come after 'to/TO' are tagged as verbs 'VB', which is clearly incorrect. This is because NLTK's built-in tagger is trained using the penntreebank dataset, and it takes 'to/TO' as a strong signal for a 'VB'.</p> <p>In general, the performance of a POS tagger depends a lot on the data used to train it. There are alternatives to it - one, you can try using an alternative tagger such as the Stanford tagger, Spacy etc. (though note that getting them up and running in python may take a bit of time in installing dependencies/debugging etc.).</p> <p>The other alternative (recommended as a quick fix) is to use a backup tagger within NLTK, i.e. manually specify a unigram/bigram tagger to be used, and backed up by the standard NLTK tagger. You can learn how to do that here.</p>
--	--

To correct the POS tags manually, use the **backoff** option in the `nltk.tag()` method. The backoff option allows to chain multiple taggers together. If one tagger doesn't know how to tag a word, it can back off to another one.

It is difficult to get 100% accuracy in POS tagging. Therefore, in this exercise, we stick to the NLTK POS tagger and use it for predicting the IOB labels. Also, while building classifiers for NER in the next sections, we see that POS tags form just one 'feature' for prediction, we use other features as well (such as morphology of words, the words themselves, other derived features etc.).

Creating 3-tuples of (word, pos, IOS_label)

To train a model, we need the entity labels of each word along with the POS tags, for e.g. in this format:

```
('show', 'VB', 'O'),
('me', 'PRP', 'O'),
('the', 'DT', 'O'),
('cheapest', 'JJS', 'B-cost_relative'),
('round', 'NN', 'B-round_trip'),
('trips', 'NNS', 'I-round_trip'),
('from', 'IN', 'O'),
('dallas', 'NN', 'B-fromloc.city_name'),
('to', 'TO', 'O'),
('baltimore', 'VB', 'B-toloc.city_name')

# function to create (word, pos_tag, iob_label) tuples for a given dataset
def create_word_pos_label(pos_tagged_data, labels):
    iob_labels = [] # initialize the list of 3-tuples to be returned
```

```

for sent in list(zip(pos_tagged_data, labels)):
    pos = sent[0]
    labels = sent[1]
    zipped_list = list(zip(pos, labels)) # [(word, pos), label]

    # create (word, pos, label) tuples from zipped list
    tuple_3 = [(word_pos_tuple[0], word_pos_tuple[1], id_to_labels[label])
               for word_pos_tuple, label in zipped_list]
    iob_labels.append(tuple_3)
return iob_labels

```

Now, for each word, we have the POS tag. The final dataset is in a 3-tuple form (word, POS tag, IOB label). But NLTK doesn't process the data in form of tuples. So, these tuples are converted to trees using the method `conlltags2tree()` in NLTK.

Read more <https://stackoverflow.com/questions/40879520/nltk-convert-a-chunked-tree-into-a-list-iob-tagging>

POS Tagger

While POS tagging the datasets, you used the following code:

```
# pos tagging train, validation and test sets

train_pos = pos_tag(train_x)
```

Let's use a combination of POS taggers in the following precedence order:

Bigram -> Unigram -> Default tagger

Which of the following is the correct option for training:

☒

```
Unigram_tagger = nltk.UnigramTagger(train_set, backoff=DefaultTagger)

Bigram_tagger = nltk.BigramTagger(train_set, backoff=Unigram_tagger)
```

🔔 Feedback:

Unigram Tagger is backed by Default Tagger. And Bigram Tagger is tagged by Unigram Tagger

☐

```
Bigram_tagger = nltk.BigramTagger(train_set, backoff=DefaultTagger)

Unigram_tagger = nltk.UnigramTagger(train_set, backoff=BigramTagger)
```

☐

```
DefaultTagger = nltk.BigramTagger(train_set, backoff=DefaultTagger)

Unigram_tagger = nltk.UnigramTagger(train_set, backoff=DefaultTagger)
```

☐ all of the above

Rule-Based Models (For NER)

NER is a sequence prediction task. There are broadly two types of models for NER - **rule-based techniques** and **probabilistic models**.

Simpler rule-based models for entity recognition or Rule-based taggers use the commonly observed rules in the text to identify the tag of each word. They are like the rule-based POS taggers which use rules such as these - VBG mostly ends with '-ing', VBD is likely to end with 'ed' etc.

Chunking

Rule-based models for NER tasks are based on **chunking**. Chunking is a commonly used **shallow parsing technique** used to chunk words that constitute some meaningful phrase in the sentence. Chunks are **non-overlapping** subsets of words in a sentence that form a meaningful 'entity'. For example, a **noun phrase chunk** (NP chunk) is commonly used in NER tasks to identify groups of words that correspond to some 'entity'. For example, in the following sentence, there are two noun phrase chunks:

Sentence: He bought a new car from the Maruti Suzuki showroom.

***Noun phrase chunks** - a new car, the Maruti Suzuki showroom*

A key difference between a noun phrase (NP) used in constituency parsing and a **noun phrase chunk** is that a chunk does not include any other noun phrase chunk within it, i.e. NP chunks are non-overlapping. This is also why chunking is a **shallow parsing** technique which falls somewhere between POS tagging and constituency parsing.

In general, the idea of chunking in the context of entity recognition is simple - since we know that most entities are nouns and noun phrases, we can write rules to extract these noun phrases and hopefully extract a large number of named entities (e.g. Maruti Suzuki, a new car, as shown above).

Concept of **chunking** in detail and how regular expressions can be used to identify chunks in the sentence.

Rule-Based Models for Entity Recognition

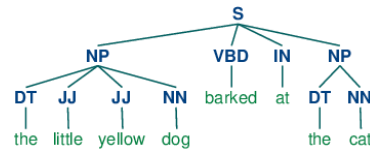
The most basic rule-based system can be written using regular expressions. The idea is to manually identify patterns which indicate occurrence of entities we are interested in, such as source and destination cities, mentions of dates and time, names of organisations (in this case airlines such as united airlines, American airlines etc.) and write regular expressions to match them.

Chunking

Chunking is a way to identify meaningful sequences of tokens called chunks in a sentence. It is commonly used to identify sequences of nouns, verbs etc. For example, in the example sentence taken from the NLTK book:

S = "We saw the yellow dog"

there are two **noun phrase chunks** as shown below. Each outer box represents a chunk.



The corresponding **IOB representation** of the same is as follows:

W	e	s	a	w	t	h	e	y	e	l	l	o	w	d	o	g
PRP		VBD			DT		JJ						NN			
B-NP		O			B-NP		I-NP						I-NP			

Similarly, in our dataset, the following sentence contains chunks such as `fromloc.city_name` (san francisco), `class_type` (first class), `depart_time.time` (DIGITDIGIT noon) etc.

Let's take some more examples of chunking done using regular expressions:

Sentence: Ram booked the flight.

Noun phrase chunks: 'Ram', 'the flight'

One possible grammar to chunk the sentence is as follows:

- Grammar: 'NP_chunk: {<DT>?<NN>}'

In NLTK, to get the IOB labels of the parsed sentence, use the 'tree2conlltags()' method.

There are various techniques for building chunkers, such as regex based, unigram and bigram chunkers etc.

Regular Expression Based Chunkers ¶

Regex based chunkers define what is called a **chunk grammar**. A **chunk grammar** is a **pattern of POS tags** which are likely to form a particular chunk (and thus POS tagging is a necessary preprocessing step for such chunkers).

The example from the NLTK book defined a simple grammar to identify noun phrase chunks:

```
1: # chunking example sentence
sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
            ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
```

Now, we **define a chunk grammar** to identify noun phrase chunks as follows: *A noun phrase chunk occurs when an optional determiner (DT) is followed by any number of adjectives (JJ) and then a noun (NN).*

```
1: # define chunk grammar to identify noun phrase chunks
# an optional determiner (DT), followed by any number of adjectives (JJ)
# and then a noun (NN)
grammar = "NP_chunk: {<DT>?<NN><JJ>+}"
```

We now use the `nltk.RegexpParser` to parse the given sentence. The output will be a tree which identifies the NP chunks.

```
1: # parse the sentence
cp = nltk.RegexpParser(grammar)
result = cp.parse(sentence)
print(result)

(S
  the/DT
  little/JJ
  yellow/JJ
  (NP_chunk dog/NN)
  barked/VBD
  at/IN
  (NP_chunk the/DT cat/NN))
```

The first NP chunk is 'the little yellow dog' and the second one is 'the cat'. One can also print the list representation of this tree as follows:

Chunking

Consider the sentence: The evening flight from Kolkata could be delayed

The final tree of IOB labels looks like following:

```
[('the', 'DT', 'B-NP_chunk'),
 ('evening', 'NN', 'I-NP_chunk'),
 ('flight', 'NN', 'I-NP_chunk'),
 ('from', 'NN', 'O'),
 ('Kolkata', 'NN', 'B-NP_chunk'),
 ('landed', 'O')]
```

Which of the grammar passed in RegexpParser could result in the above tree:

- ☒ Grammar = "NP_chunk: {<DT>?<NN>+}"
- ☐ Grammar = "NP_chunk: {<DT>?<NN>}"
- ☐ Grammar = "NP_chunk: {<NN>+}"
- ☐ All of the above

Applying Chunking to the Flight Reservation Dataset

Now, let's apply the concept of chunking to the ATIS dataset.

In the training set, we have the word, the entity and the POS tag of each word. Let's understand how we can use chunk grammar to extract chunks. Note that our dataset contains far more types of chunks than just noun phrase chunks, such as 'round trip', 'cost-relative', 'fromloc-city-name' etc. A sample query from the training set looks like the following:

```
('show', 'VB', 'O'),  
( 'me', 'PRP', 'O'),  
( 'the', 'DT', 'O'),  
( 'cheapest', 'JJS', 'B-cost_relative'),  
( 'round', 'NN', 'B-round_trip'),  
( 'trips', 'NNS', 'I-round_trip'),  
( 'from', 'IN', 'O'),  
( 'baltimore', 'NN', 'B-fromloc.city_name'),  
( 'to', 'TO', 'O'),  
( 'Denver', 'VB', 'B-toloc.city_name')
```

The word after 'to' is getting tagged as a verb, whereas it is actually a noun. You'll see this issue in many queries in the dataset. Let's write a regular expression such that is able to identify:

1. Denver as '**toloc.city_name**' chunk,
2. Baltimore as '**fromloc.city_name**' chunk,
3. round trips as '**round_trip**' chunk,
4. Cheapest as '**cost_relative**' chunk

The following regular expression rules may be used to do that:

```
Grammar = "toloc.city_name: {<VB><NN>?}  
fromloc.city_name: {<JJ>?<NN>}  
round_trip: {<NN><NNS>}  
cost_relative: {<JJS>?<JJR>?}"
```

How to write simple rule-based chunkers. Our NER task is far more complex than what simple rule-based chunkers can accomplish, and thus we need more sophisticated models.

```
# grammar for source and destination city chunks
grammar = '''
fromloc.city_name: {<JJ>?<NN>}
toloc.city_name: {<VB><NN>?}
'''

cp = nltk.RegexpParser(grammar)
result = cp.evaluate(train_trees)
print(result)
```

```
ChunkParse score:
  IOB Accuracy: 65.2%%
  Precision: 31.7%%
  Recall: 40.9%%
  F-Measure: 35.7%%
```

The results are although better than the baseline model, they are still quite unimpressive. Notice that now precision, recall and f-score are non-zero, indicating that the chunker is able to identify at least some chunks correctly (in this case `fromloc.city_name` and `toloc.city_name`).

We can add more regex patterns for other chunk types as well.

We can see that in this dataset, queries are quite complex (large variety of labels, sentence structures etc.) and thus it is extremely hard write hand-written rules to extract useful entities.

Thus, we need to train probabilistic models such as CRFs, HMMs etc. to tag each word with its corresponding entity label.

Rule-based systems become more complex and difficult to maintain as we keep on increasing the rules.

The alternative, in this case, is to use probabilistic machine learning models.

IOB labeling-Regex

Following is one of the queries from the ATIS dataset.

('show', 'VB', 'O'),
('me', 'PRP', 'O'),
('the', 'DT', 'O'),
('least', 'JJ\$', 'B-cost_relative'),
('expensive', 'JJ', 'B-cost'),
('single', 'NN', 'B-round_trip'),
('trips', 'NNS', 'I-round_trip'),
('from', 'IN', 'O'),
('baltimore', 'NN', 'B-fromloc.city_name'),
('to', 'TO', 'O'),
('Dallas', 'VB', 'B-toloc.city_name'),
('Fort', 'NN', 'I-fromloc.city_name'),
('Worth', 'NN', 'I-fromloc.city_name')

Select the regular expression that is able to identify:

Dallas Fort Worth as 'toloc.city_name' chunk,

Baltimore as 'fromloc.city_name' chunk,

round trips as 'round_trip' chunk,

Cheapest as 'cost_relative' chunk

☒ toloc.city_name: {<VB><NN>?<NN>?}
fromloc.city_name: {<JJ>?<NN>}
round_trip: {<NN><NNS>}
cost_relative: {<JJ\$>?<JJR>?}

Feedback:

Dallas Fort Worth is VB/NN/NN, but in general if the city is one word, then NN is optional

☐ fromloc.city_name: {<JJ>?<NN>}
toloc.city_name: {<VB><NN>?}
round_trip: {<NN><NNS>}
cost_relative: {<JJ\$>?<JJR>?}

☐ fromloc.city_name: {<JJ>?<NN>}
toloc.city_name: {<VB><NN><NN>?}
round_trip: {<NN><NNS>}
cost_relative: {<JJ\$>?<JJR>?}

☐ All of the above

Probabilistic Models for Entity Recognition

Two probabilistic models to get the most probable IOB tags for words. Recall that you have studied the **unigram** and **bigram** models for POS tagging earlier:

1. **Unigram chunker** computes the unigram probabilities $P(\text{IOB label} \mid \text{pos})$ for each word and assigns the label that is most likely for the POS tag.
2. **Bigram chunker** works similar to a unigram chunker, the only difference being that now the probability of a POS tag having an IOB label is computed using the current and the previous POS tags, i.e. $P(\text{label} \mid \text{pos}, \text{prev_pos})$.

Unigram Chunker

A unigram chunker assigns the IOB label that is most likely for each POS tag.

Class `UnigramChunker` on initialisation, first converts the tree form of a sentence to the list form (word, pos, label), extracts the (pos, label) pairs and computes the unigram probabilities $P(\text{label} \mid \text{pos})$ for each POS tag. It then simply assigns the label that is most likely for the POS tag.

The `parse()` method of the class takes a sentence in the form (word, pos) as the input, extracts only the pos tag from it, and uses the unigram tagger to assign the IOB label to each word. It then returns the sentence after converting it to a tree format.

Note that the unigram tagger, like the previous regex-based chunkers, *does not make use of the word itself but only the word's POS tag*.

```
# unigram chunker

from nltk import ChunkParserI

class UnigramChunker(ChunkParserI):
    def __init__(self, train_sents):
        # convert train sents from tree format to tags
        train_data = [(t, c) for w, t, c in nltk.chunk.tree2conlltags(sent)]
                        for sent in train_sents]
        self.tagger = nltk.UnigramTagger(train_data)

    def parse(self, sentence):
        pos_tags = [pos for (word, pos) in sentence]
        tagged_pos_tags = self.tagger.tag(pos_tags)
        chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]

        # convert to tree again
        conlltags = [(word, pos, chunktag) for ((word, pos), chunktag) in zip(sentence, chunktags)]
        return nltk.chunk.conlltags2tree(conlltags)
```

```
# unigram chunker
unigram_chunker = UnigramChunker(train_trees)
print(unigram_chunker.evaluate(valid_trees))
```

```
# printing the most likely IOB tags for each POS tag

# extract the list of pos tags
postags = sorted(set([pos for sent in train_trees for (word, pos) in sent.leaves()]))

# for each tag, assign the most likely IOB label
print(unigram_chunker.tagger.tag(postags))

[('CC', 'O'), ('CD', 'B-round_trip'), ('DT', 'O'), ('EX', 'O'), ('FW', 'B-fromloc.city_name'), ('IN', 'O'), ('JJ', 'O'), ('JJR', 'B-cost_relative'), ('JJS', 'B-cost_relative'), ('MD', 'O'), ('NN', 'O'), ('NNP', 'B-depart_time.time'), ('NNS', 'O'), ('PDT', 'O'), ('POS', 'O'), ('PRP', 'O'), ('PRP$', 'O'), ('RB', 'O'), ('RBR', 'B-cost_relative'), ('RBS', 'B-cost_relative'), ('RP', 'O'), ('TO', 'O'), ('VB', 'B-tooloc.city_name'), ('VBD', 'O'), ('VBG', 'O'), ('VBN', 'O'), ('VBP', 'O'), ('VBZ', 'O'), ('WDT', 'O'), ('WP', 'O'), ('WRB', 'O')]
```

The unigram tagger has learnt that most pos tags are indeed an 'O', i.e. don't form an entity. Some interesting patterns it has learnt are:

- JJR, JJS (relative adjectives), are most likely B-cost_relative (e.g. cheapest, cheaper)
- NNP is most likely to be B-depart_time.time

Bigram Chunker

Let's try a bigram chunker as well - we just need to change the `UnigramTagger` to `BigramTagger`. This works exactly like the unigram chunker, the only difference being that now the probability of a pos tag having a label is computed using the current and the previous POS tags, i.e. $P(\text{label} \mid \text{pos}, \text{prev_pos})$.

```
# bigram tagger

class BigramChunker(ChunkParserI):
    def __init__(self, train_sents):
        # convert train_sents from tree format to tags
        train_data = [[(t, c) for w, t, c in nltk.chunk.tree2conlltags(sent)]
                       for sent in train_sents]
        self.tagger = nltk.BigramTagger(train_data)

    def parse(self, sentence):
        pos_tags = [pos for (word, pos) in sentence]
        tagged_pos_tags = self.tagger.tag(pos_tags)
        chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]

        # convert to tree again
        conlltags = [(word, pos, chunktag) for ((word, pos), chunktag) in zip(sentence, chunktags)]
        return nltk.chunk.conlltags2tree(conlltags)

# unigram chunker
bigram_chunker = BigramChunker(train_trees)
print(bigram_chunker.evaluate(valid_trees))
```

The metrics have improved significantly from unigram to bigram, which is expected. However, there are still some major flaws in this approach to build chunkers, the main drawback being that the model *uses only the POS tag to assign the label, not the actual word itself*.

It is likely that if a model can make use of the word itself apart from the POS tag, it should be able to learn more complex patterns needed for this task.

In fact, apart from the word, we can extract many other features, such as previous word, previous tag, whether the word is a numeric, whether the word is a city or an airline company etc.

Following few sections extract a variety of features and build classifiers such as Naive Bayes using those features.

First step in the direction of feature extraction is to extract a feature which indicates whether a word is a city, state or county etc. Such features can be extracted by simply **looking up a gazetteer**.

Another way to identify named entities (like cities and states) is to look up a dictionary or a **gazetteer**. A gazetteer is a geographical directory which stores data regarding the names of geographical entities (cities, states, countries) and some other features related to the geographies. An example gazetteer file for the US is given below.

Data download URL: https://raw.githubusercontent.com/grammakov/USA-cities-and-states/master/us_cities_states_counties.csv

Data download URL: https://raw.githubusercontent.com/grammakov/USA-cities-and-states/master/us_cities_states_counties.csv

We'll write a simple function which takes a word as input and returns a tuple indicating **whether the word is a city, state or a county**.

```
: # reading a file containing list of US cities, states and counties
us_cities = pd.read_csv("us_cities_states_counties.csv", sep="|")
us_cities.head()

: # storing cities, states and counties as sets
cities = set(us_cities['City'].str.lower())
states = set(us_cities['State full'].str.lower())
counties = set(us_cities['County'].str.lower())

: print(len(cities))
print(len(states))
print(len(counties))

: # define a function to look up a given word in cities, states, county
def gazetteer_lookup(word):
    return (word in cities, word in states, word in counties)

: # sample lookups
print(gazetteer_lookup('washington'))
print(gazetteer_lookup('utah'))
print(gazetteer_lookup('philadelphia'))
```

Naive Bayes Classifier for NER

Till now, we have covered rule-based models, unigram and bigram models for predicting the IOB labels. This segment will **build a machine learning model to predict IOB tags of the words**.

Just like machine learning classification models, we can have features for sequence labelling task. Features could be the morphology (or shape) of the word such as whether the word is upper/lowercase, POS tags of the words in the neighbourhood, whether the word is present in the gazetteer (i.e. `word_is_city`, `word_is_state`), etc.

Features- IOB labeling

Sequence prediction tasks such as NER can be modelled by classifiers such as Naive Bayes, Decision Trees etc. The task is as follows - you are given a list of words whose IOB tags are to be predicted. Typically, you assign POS tags to the words as well. The target label sequence is a vector of IOB labels.

To build classifiers, you create features for each word in the sequence. Think about at least five features which you think can be useful predictors of IOB labels.

Position in sentences,
Likelihood of being chunked,
Percentage occurrences in documents.

Words 16

Note: Once submitted, answer is not editable.



Suggested Answer

The word itself, previous word, next word, pos tag, previous pos, next pos, previous IOB label, `word_is_city`, `word_is_uppercase` etc.

In this segment, we'll take up **Naive Bayes Classifier** to predict labels of the words. We'll take into account features such as the word itself, POS tag of the word and POS tag of the previous word, whether the word is the first or last word of the sentence, whether the word is in the gazetteer etc.

IOB tagging is a **sequence classification task** - given a sequence of words and pos tags, predict the IOB label of the word.

One of the main advantages of classifier based chunkers is that we can use a variety of features which we think will be strong indicators of a word's IOB tag.

For e.g. if a word is a state/city name such as 'boston', it is very likely an `B-fromloc.city_name` or `B-toloc.city_name`.

Similarly, we can expect that the **previous word and the previous POS tag** could help predict the IOB labels of a word; that **if a word is the first or the last in the sentence** may strongly help predict the IOB label, etc.

Also, in all sequence classification tasks, one can use the **predicted labels of the previous words** as features (recall that HMMs compute transition probabilities).

<p>Naive Bayes classifier which uses a variety of word features for classification.</p> <p>It takes in a sentence and the word whose features are to be extracted (defined by its index <i>i</i> in the sentence) as input and returns a dictionary of word features as output. It also takes a <code>history</code> argument - a list of already predicted previous tags to the left of the target word, which is useful if you are using them as features.</p>	<pre># extracts features for the word at index i in a sentence def npchunk_features(sentence, i, history): word, pos = sentence[i] # the first word has both previous word and previous tag undefined if i == 0: prevword, prevpos = "<START>", "<START>" else: prevword, prevpos = sentence[i-1] # gazetteer lookup features (see section below) gazetteer = gazetteer_lookup(word) return {"pos": pos, "prevpos": prevpos, 'word':word, 'word_is_city': gazetteer[0], 'word_is_state': gazetteer[1], 'word_is_county': gazetteer[2]}</pre>
---	---

Function `npchunk_features()` will return the following features for each word in a dictionary:

1. POS word of the tag
2. Previous POS tag
3. Current word
4. Whether the word is in gazetteer as 'City'
5. Whether the word is in gazetteer as 'State'
6. Whether the word is in gazetteer as 'County'

Now, each word contains all the above features. We'll build a **Naive Bayes classifier** using these features. We are using the Naive Bayes implementation of the NLTK library rather than sklearn.

Now, we define two classes `ConsecutiveNPChunkTagger` and `ConsecutiveNPChunker`.

ConsecutiveNPChunkTagger

`__init__` method creates `train_set` - a list of labelled training sentences, a list of tuples (featureset, tag) - each tuple is featureset (dict) of a word and its label.

List `history` contains list of previously predicted IOB tags, (left of the target word). We can only use IOB tags to the left of the target word since that's all the tags we have at the time of prediction.

`__init__` method loops through an IOB tagged list of `train_sents`. It first untags the IOB tags to generate (word, pos_tag) tuples stored in `untagged_sent`. These tuples are used to compute the

word features. For each (word, IOB_tag) in tagged_sent, it computes the word features and appends feature dict and tag to train_sents. It further appends the IOB tag to history.

tag() method takes in a sentence as a list of words and predicts IOB label of each word in sentence.

ConsecutiveNPChunker

This class does uninteresting work of converting between tree-list-tree formats (NLTK's builtin classifiers need the list format). It takes in a list of sentences as trees, converts each sentence to the list form, and then initialises its tagger using methods already defined in the ConsecutiveNPChunkTagger class. The parse method tags the sentence and returns it in the tree format since it is easier to print and read.

<pre> class ConsecutiveNPChunkTagger(nltk.TaggerI): def __init__(self, train_sents): train_set = [] for tagged_sent in train_sents: untagged_sent = nltk.tag.untag(tagged_sent) history = [] # compute features for each word for i, (word, tag) in enumerate(tagged_sent): featureset = npchunk_features(untagged_sent, i, history) train_set.append((featureset, tag)) history.append(tag) self.classifier = nltk.NaiveBayesClassifier.train(train_set) def tag(self, sentence): history = [] for i, word in enumerate(sentence): featureset = npchunk_features(sentence, i, history) tag = self.classifier.classify(featureset) history.append(tag) return zip(sentence, history) </pre>	<pre> class ConsecutiveNPChunker(nltk.ChunkParserI): def __init__(self, train_sents): tagged_sents = [((w,t),c) for (w,t,c) in nltk.chunk.tree2conlltags(sent)] for sent in train_sents] self.tagger = ConsecutiveNPChunkTagger(tagged_sents) def parse(self, sentence): tagged_sents = self.tagger.tag(sentence) conlltags = [(w,t,c) for ((w,t),c) in tagged_sents] return nltk.chunk.conlltags2tree(conlltags) </pre>
---	--

```

# training the chunker
chunker = ConsecutiveNPChunker(train_trees)

```

```

# evaluate the chunker
print(chunker.evaluate(valid_trees))

```

```

ChunkParse score:
IOB Accuracy:   91.7%%
Precision:      75.3%%
Recall:         81.8%%
F-Measure:     78.4%%

```

The results have improved significantly compared to the basic unigram/bigram chunkers, and they may improve further if we create better features.

For example, if the word is 'DIGIT' (numbers are labelled as 'DIGIT' in this dataset), we can have a feature which indicates that (see example below). In this dataset, 4-digit numbers are encoded as 'DIGITDIGITDIGITDIGIT'.

So, the Naive Bayes classifier performed better than the rule-based, unigram or bigram chunker models. These results improved marginally when more features are created.

NB classifier

'Expensive' had an I-label for cost entity. What could be the possible B-label for cost? (Select all that apply)

☒ Least

✓ Correct

🗨 Feedback :

least expensive.

☐ Longest

☐ Flight

☐ All of the above

Additional Reading

- You can also use the Naive Bayes classifier to predict POS tags for the word in a sentence. Refer to section 1.6 of the following link:
<https://www.nltk.org/book/ch06.html>
- Read more on [why Naive Bayes perform well](#) even when the features are dependent on each other.

Decision Tree Classifiers for NER

Use Decision Trees as the classifier for the NER task. The features extraction process is exactly the same as that used in the Naive Bayes classifier.

Decision Trees

Compare the results of Naive Bayes and Decision Trees for NER task. Is there overfitting in Decision Trees

☐ Yes

☒ No



Correct

Feedback :

Overfitting takes place when only training results show considerable good accuracy and not the validation and test set

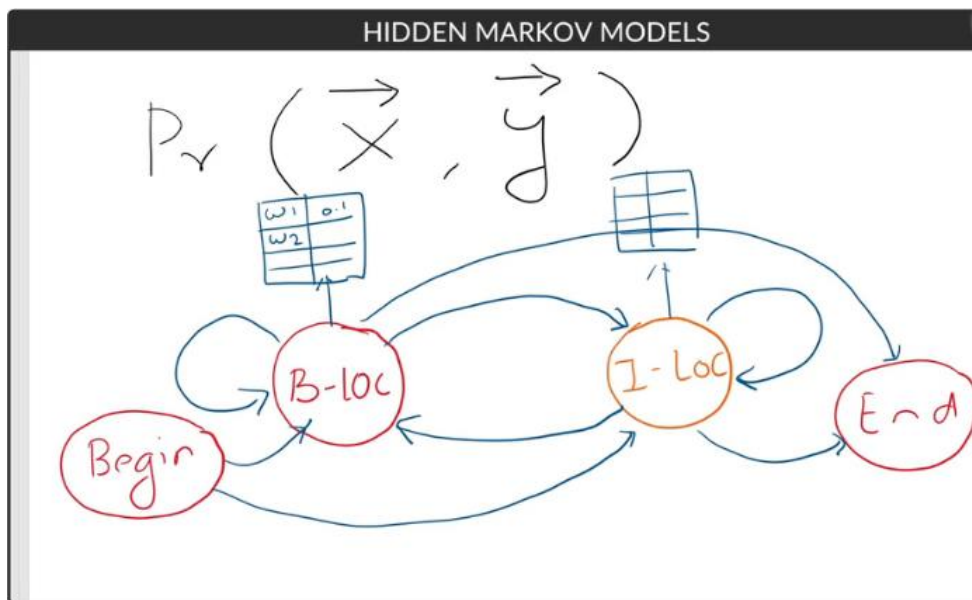
Additional Reading

- To experiment with the hyperparameters of the decision tree classifier, refer to the NLTK documentation [here](#).

HMM and IOB labelling

We studied HMMs in detail and used them for building POS taggers. In general, HMMs can be used for any sequence classification task, such as NER. However, many NER tasks and datasets are far more complex than tasks such as POS tagging, and therefore, more sophisticated sequence models have been developed and widely accepted in the NLP community.

In this segment, study some limitations of HMMs and understand why they are not preferred for the NER task we are trying to solve.



Summary

This session built an **information extraction (IE) system** which can extract entities relevant for booking flights (such as source and destination cities, time, date, budget constraints etc.) in a structured format from unstructured user-generated queries.

How to use different models for **Named Entity Recognition** and their Python implementation.

Following techniques and models for NER:

- Rule-based techniques
 1. Chunking
 2. Regular expression-based techniques
- Probabilistic models
 1. Unigram & Bigram models
 2. Naive Bayes Classifier
 3. Decision trees
 4. *Conditional Random Fields (CRFs)* - (Optional)

CRFs - Another Probabilistic Approach

Information Extraction (IE) extracted entities from unstructured user queries in a structured format using two types of techniques:

1. **Rule-based** techniques (by chunking phrases and using regular expressions)
2. **Probabilistic models** (Unigram, Bigram, Naive Bayes and Decision Trees)

Next **optional session** has another type of probabilistic model widely used in NLP for sequence modelling - **Conditional Random Fields (CRFs)**.

The concepts covered in the optional session are explained below briefly. Highly recommended to go through the session on CRFs since they are used heavily in the industry. CRFs are also used in the upcoming chatbot project. Although no need to build CRFs from scratch (rather use a library), but an understanding of how they work will be helpful.

CRFs Overview

CRFs are used in a wide variety of **sequence labelling tasks** such as POS tagging, speech recognition, NER, and even in computational biology for modelling genetic patterns etc. These are commonly used as an alternative to HMMs, and, in some applications, have empirically proven to be significantly better than HMMs. Next session shows that CRFs achieve much better accuracy (on the ATIS flight booking dataset) than any of the models tried in the previous session.

In the session

- CRFs architecture
- Feature functions
- Training CRFs
- Python implementation of CRF